Processos i fils

José Antonio Leo Megías

Índex

Introducció

Contínuament veiem persones i màquines fent accions de forma simultàniament. Els ordinadors intenten simular aquesta realitat, on moltes coses passen a la vegada.

Un dels avantatges de realitzar diverses coses a la vegada és que es podem acabar més ràpid. Aquest seria el paradigma que s'intenta seguir quan a un ordinador executen diverses tasques alhora.

El multiprocés, ha estat molt lligat al sistema operatiu. De fet, els primers programes concurrents van ser els sistemes operatius, on ordinadors amb un únic processador en el sistema operatiu s'encarregava de repartir el temps del processador entre les diferents tasques.

Amb l'avanç i l'accessibilitat de la tecnologia, els ordinadors s'estan dotant de més d'un processador i per tant la programació en paral·lel és un fet. Al millorar les comunicacions també és més viable implementar sistemes distribuïts, on cada ordinador executa part d'una aplicació.

Durant els darrers temps, la programació concurrent s'està implementant a una gran part dels sistemes informàtics. L'aparició del fil o *thread* i de llenguatges de programació que donen suport a la programació concurrent amb primitives pròpies del llenguatge, com Java, han fet que l'execució d'una aplicació sigui més ràpida i una mica més simple de programar. Malgrat tot la programació multifil, on diversos fils estan en execució continua sent una mica més complexa.

Un altre element que ha propiciat el desenvolupament d'aplicacions concurrents és Internet, ja que qualsevol aplicació que executem sobre Internet (programes de missatgeria, navegadors, etc.) estarà lligada a la programació concurrent.

La unitat està dividida en dues activitats, la primera "Programació multiprocés" veurem les característiques de la programació concurrent, paral·lela i distribuïda, veurem que són els processos, quins són els seus estats, els problemes i les solucions que es plantegen quan s'apliquen tècniques de concurrència, com ara la comunicació i la sincronització.

La segona unitat "programació multifil", veurem com s'implementen aplicacions amb diversos fils en execució. Veurem els diferents estats d'un fil, els problemes que pot plantejar la seva programació i com es poden solucionar.

És molt important seguir la teoria i entendre els exemples que es proposen, anar fent els exercicis d'autoavaluació, i realitzar les activitats per a una bona compressió de la unitat.

Resultats d'aprenentatge

1. Desenvolupa aplicacions compostes per diversos processos reconeixent i aplicant principis de programació paral·lela.

- Reconeix les característiques de la programació concurrent i els seus àmbits d'aplicació.
- Identifica les diferències entre programació paral·lela i programació distribuïda, els seus avantatges i inconvenients.
- Analitza les característiques dels processos i de la seva execució a càrrec del sistema operatiu.
- Caracteritza els fils d'execució i descriu la seva relació amb els processos.
- Utilitza classes per programar aplicacions que creïn subprocessos.
- Utilitza mecanismes per sincronitzar i obtenir el valor retornat per als subprocessos iniciats.
- Desenvolupa aplicacions que gestionin i utilitzin processos per a l'execució de diverses tasques en paral·lel.
- Depura i documenta les aplicacions desenvolupades.

2. Desenvolupa aplicacions compostes per diversos fils d'execució analitzant i aplicant llibreries específiques del llenguatge de programació.

- Identifica situacions en què sigui útil la utilització de diversos fils en un programa.
- Reconeix els mecanismes per a crear, iniciar i finalitzar fils.
- Programa aplicacions que implementen diversos fils.
- Identifica els possibles estats d'execució d'un fil i programa aplicacions que els gestionen.
- Utilitza mecanismes per compartir informació entre diversos fils d'un mateix procés.
- Desenvolupa programes formats per diversos fils sincronitzats mitjançant tècniques específiques.
- Estableix i controla la prioritat de cadascun dels fils d'execució.
- Depura i documenta els programes desenvolupats.

1. Programació multiprocés i paral·lela

El dia a dia demana cada cop més potència de càlcul a les aplicacions informàtiques, més i més càlculs numèrics en enginyeria, ciència, astrofísica, meteorologia... Volem que els ordinadors reaccionin amb un raonament humà. Així tenim ordinadors que juguen partides d'escacs, que parlen com humans o que prenen decisions a partir de dades poc precises...

Malgrat que no és l'únic factor determinant, els processadors juguen un paper fonamental a l'hora d'assolir aquesta potència. Cada cop són més ràpids i eficients i els sistemes operatius permeten coordinar diversos processadors per incrementar el nombre de càlculs per unitat de temps. L'ús de diversos processadors dins d'un sistema informàtic s'anomena multiprocés. Ara bé, la coordinació de diversos processadors pot provocar situacions d'error que haurem d'evitar.

1.1 Programació multiprocés

Un cuiner que tingui intenció de fer un plat molt elaborat, amb salses i guarnicions diferents, probablement farà servir diferents paelles per cuinar cada element que necessiti utilitzar per elaborar el plat. Tindrà una paella amb la salsa, una altra preparant la guarnició i en una altra cuinarà el plat principal. Tot això ho farà alhora i, finalment, quan acabi totes aquestes tasques les ajuntarà en un únic plat.

Aquesta analogia ens serveix per introduir el terme **multiprocés**. El cuiner està executant diferents processos a la vegada: cuina la salsa, la guarnició i el plat principal, seguint un ordre d'execució per arribar a un resultat que esperava, un bon plat.

1.1.1 Procés, sistemes monoprocessador-multiprocessador

Seguint amb l'analogia del cuiner, un procés seria l'activitat realitzada pel cuiner (processador) d'elaborar algun dels complements o el plat principal (resultats) a partir d'una recepta (programa). Cada procés de creació requereix agafar els ingredients (les dades), posar-los a la paella (recurs associat) i cuinar-los amb independència de la resta de processos.

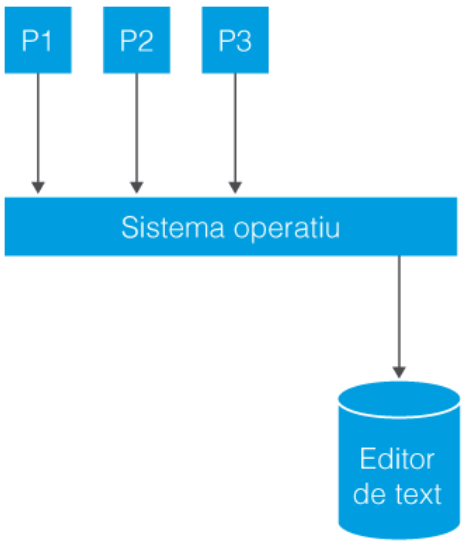
Un programa és un element estàtic, un conjunt d'instruccions, unes línies de codi escrites en un llenguatge de programació, que descriuen el tractament que cal donar a unes dades inicials (d'entrada) per aconseguir el resultat esperat (una sortida concreta). En canvi, **un procés** és dinàmic, és una instància d'un programa en execució, que realitza els canvis indicats pel programa a les dades inicials i obté una sortida concreta. El procés, a més de les instruccions, requerirà també de recursos específics per a l'execució com ara el comptador d'instruccions del programa, el contingut dels registres o les dades.

El sistema operatiu és l'encarregat de gestionar els processos.

El sistema operatiu és l'encarregat de la gestió de processos. Els crea, els elimina i els proveeix d'instruments que en permetin l'execució i també la comunicació entre ells. Quan s'executa un programa, el sistema operatiu crea una instància del programa: el procés. Si el programa es tornés a executar es crearia una nova instància totalment independent a l'anterior (un nou procés) amb les seves pròpies variables, piles i registres.

Imaginem un servidor d'aplicacions en el qual tenim instal·lat un programa d'edició de text. Posem per cas que existeixen diversos usuaris que volen escriure els seus textos executant l'editor. Cada instància del programa és un procés totalment independent als altres. Cada procés té unes dades d'entrada i per tant diferents sortides. Cada usuari escriurà a la seva execució de l'editor (el procés), el seu text. A la figura 1.1 es reflecteix aquest exemple.

Figura 1.1. Execució de processos



Quan un procés es troba en execució es troba completament en memòria i té assignats els recursos que necessita. Un procés no pot escriure en zones de memòria assignada a altres processos, la memòria no és compartida. Cada procés té una estructura de dades en la qual es guarda la informació associada a l'execució del procés. Aquesta zona s'anomena Bloc de Control de Procés (BCP). Els processos competeixen amb altres processos executats a la vegada pels recursos

del sistema. Opcionalment el sistema operatiu permet també que els processos puguin comunicar-se i col·laborar entre ells.

Fins ara hem parlat d'elements de programari, com ara programes i processos, però no del maquinari utilitzat per executar-los. L'element de maquinari en el qual s'executen els processos és el processador. Un **processador** és el component de maquinari d'un sistema informàtic encarregat d'executar les instruccions i processar les dades. Quan un dispositiu informàtic està format per un únic processador parlarem de sistemes monoprocessadors, per contra, si està format per més d'un processador parlarem de sistemes multiprocessador.

Un **sistema monoprocessador** és aquell que està format únicament per un processador.
Un **sistema multiprocessador** està format per més d'un processador.

Actualment, la majoria dels sistemes operatius aprofiten els temps de repòs dels processos, quan esperen, per exemple, alguna dada de l'usuari o es troben pendents d'alguna operació d'entrada/sortida, per introduir al processador un altre procés, simulant així una execució paral·lela.

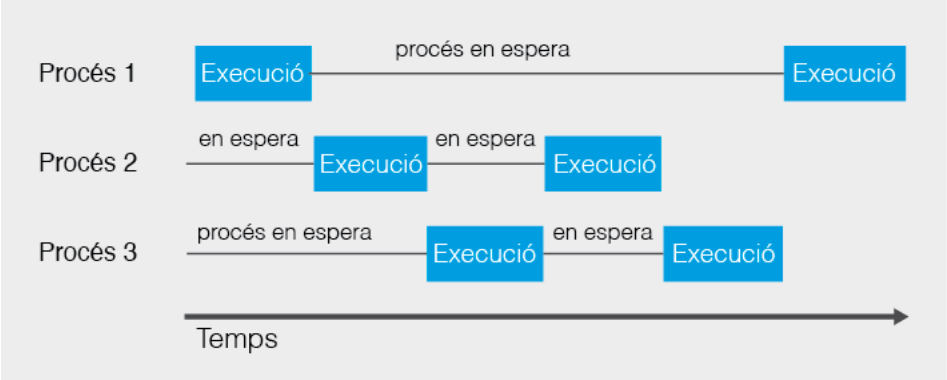
Placa Base Multiprocessador

De forma genèrica anomenarem els processos que s'executin a la vegada, ja sigui de forma real o simulada, **processos concurrents**. L'execució de processos concurrents, encara que sigui de forma simulada, fa augmentar el rendiment del sistema informàtic ja que aprofita més el temps del processador. És el sistema operatiu l'encarregat de gestionar l'execució concurrent de diferents processos contra un mateix processador i sovint ens hi referim amb el nom de **multiprogramació**.

Parlem de **multiprogramació** quan el sistema operatiu gestiona l'execució de processos concurrents a un sistema monoprocessador.

La figura 1.2 correspon a una imatge de processos concurrents en la qual hi ha tres processos en execució i es van intercalant repartint-se el temps del processador.

Figura 1.2. Execució de processos concurrents



El sistema operatiu Windows 95 únicament permet treballar amb un únic processador. Els sistemes operatius a partir de Win2000/NT i Linux/Unix són multiprocés, poden treballar amb més d'un processador

Els sistemes informàtics monoprocessadors actuals tenen una gran velocitat i si han d'executar més d'un procés a la vegada aniran alternant la seva execució simulant un multiprocés. Quan en un ordinador monoprocessador s'apliquen tècniques de multiprogramació els beneficis en rendiment no són tan evidents com en sistemes informàtics multiprocessadors.

Processador Intel amb dos nuclis

En un sistema informàtic **multiprocessador** existeixen dos o més processadors, per tant es poden executar simultàniament diversos processos. També es poden qualificar de multiprocessadors aquells dispositius el processador dels quals té més d'un nucli (*multicore*) , és a dir, tenen més d'una CPU al mateix circuit integrat del processador. Evidentment l'arquitectura és diferent, però aquest sistema, de la mateixa manera que un equip multiprocessador, ens permet executar de manera simultània processos diferents.

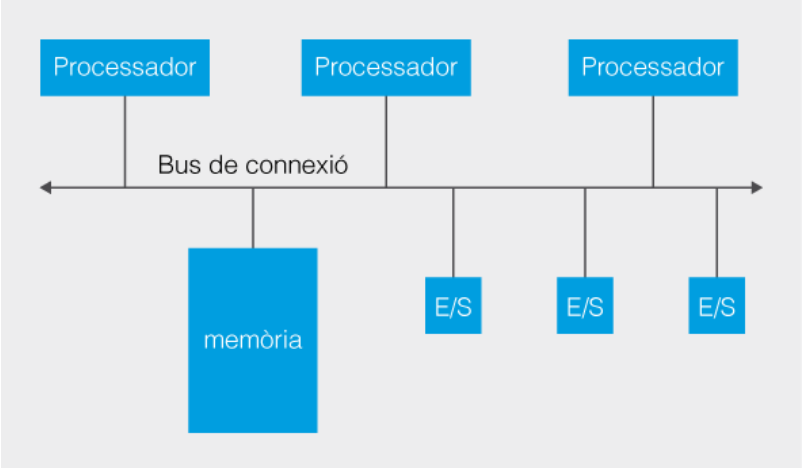
Els sistemes multiprocessadors es poden classificar depenent de la seva arquitectura en *sistemes multiprocessador fortament acoblats* i *sistemes multiprocessador dèbilment acoblats*.

Sistemes multiprocessadors fortament acoblats: en aquesta arquitectura els diferents processadors comparteixen una mateixa memòria i estan interconnectats a ella a través d'un bus. També poden tenir una petita memòria cau a cada processador. Hi ha una forta col·laboració entre processadors compartint variables a la memòria comuna a mode de resultats parcials o també per comunicar-se entre ells. Actualment, la majoria de sistemes operatius suporten aquest tipus de sistemes multiprocessador.

Els sistemes fortament acoblats depenen del pes que té cada processador a l'hora d'executar els processos. Es poden dividir en **sistemes multiprocés simètrics**, en els quals els processadors del sistema són de característiques similars i competeixen entre iguals per executar processos, i **sistemes multiprocés asimètrics** en els quals un dels processadors del sistema, el *màster*, controla la resta de processadors. Aquest sistema també s'anomena *master-slave*.

A la figura 1.3 es mostra gràficament com és un sistema multiprocessador fortament acoblat.

FIGURA 1.3. Sistema informàtic multiprocessador fortament acoblat

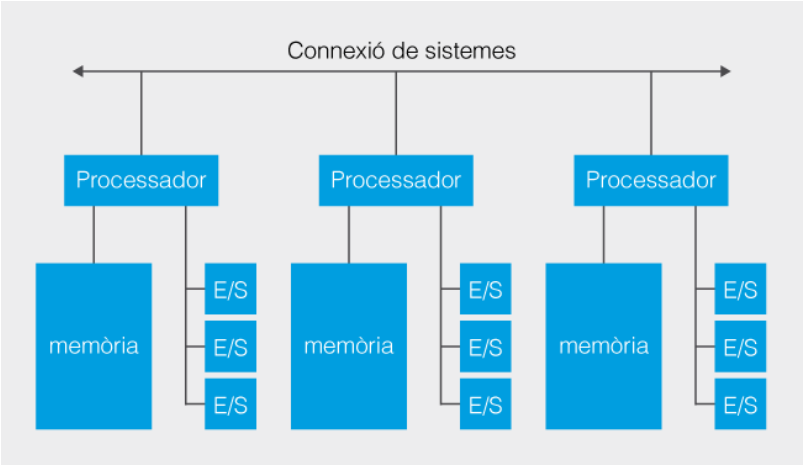


Sistemes multiprocessadors dèbilment acoblats: aquests sistemes no comparteixen memòria, cada processador té una memòria associada. A les execucions que necessiten col·laboració entre processos els cal l'intercanvi de missatges a través d'enllaços de comunicacions, per habilitar la comunicació entre processos. Un tipus d'aquests sistemes poc acoblats són els sistemes distribuïts, en els quals

cada dispositiu monoprocessador (o multiprocessador) podria estar situat a llocs físicament distants. Una xarxa d'ordinadors o Internet podria ser un exemple d'un sistema dèbilment acoblat distribuït.

A la figura 1.4 es mostra gràficament com és un sistema multiprocessador dèbilment acoblat.

Figura 1.4. Sistema informàtic multiprocessador dèbilment acoblat



1.1.2 Programació concurrent, paral·lela i distribuïda

Simula, creat al 1966 va ser el primer llenguatge de programació orientat a objectes concurrent.

Actualment, una gran quantitat d'aplicacions utilitzen la programació concurrent. En els sistemes operatius actuals existeixen moltes aplicacions executant-se al mateix temps i compartint informació. Per exemple, podem escriure en un editor de text, tenir posat un reproductor de música i descarregar un vídeo al mateix temps. També podem tenir en execució un navegador capaç de carregar en diverses pestanyes diferents pàgines web. Són exemples que il·lustren la idea de programació concurrent.

Deep Blue. Supercomputador d'IBM amb 256 processadors treballant en paral·lel

Gràcies a l'evolució que ha sofert el maquinari durant els últims anys s'han creat sistemes operatius que poden optimitzar els recursos dels processadors i poden executar diferents processos de forma simultània. L'execució simultània de processos s'anomena també **concurrència**. No vol dir que s'hagin d'executar exactament al mateix moment, l'intercalat de processos també es considera execució concurrent. La majoria dels llenguatges de programació actuals poden fer ús d'aquest recurs i dissenyar aplicacions en les quals els processos s'executin de manera concurrent.

> Parlem de **programació concurrent** quan s'executen en un dispositiu informàtic de forma simultània diferents tasques (processos).

L'execució concurrent pot comportar certs problemes a l'hora d'accedir a les dades que no trobarem mai a l'execució seqüencial i que caldrà tenir en compte. Bàsicament usarem dues tècniques per evitar-los: el boqueig i la comunicació.

Si la programació concurrent es dóna en un computador amb un únic processador parlarem de multiprogramació. En canvi, si el computador té més d'un processador i, per tant, els processos es poden executar de forma realment simultània, parlarem de programació paral·lela.

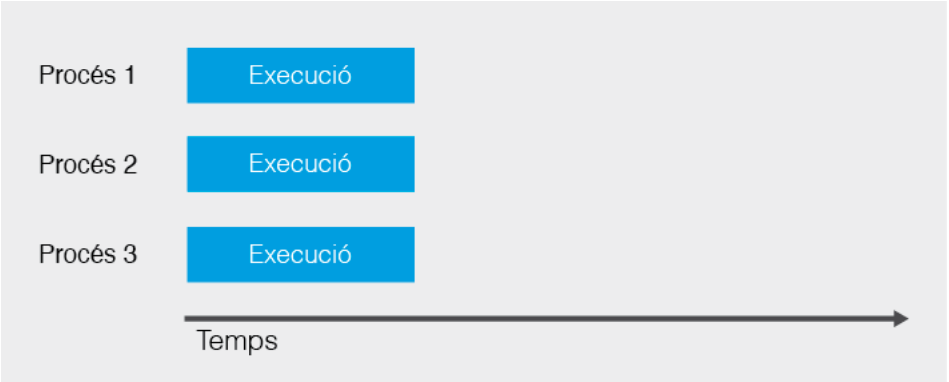
En un dispositiu multiprocessador la concurrència és real, els processos són executats de forma simultània en diferents processadors del sistema. És habitual que el número de processos que s'estigui executant de forma concurrent sigui major que el número de processadors. Per tant serà obligatori que alguns processos s'executin sobre el mateix processador.

Quan la programació concurrent es realitza en un sistema multiprocessador parlem de **programació paral·lela**.

En els ordinadors multiprocessadors la concurrència és real, per tant el temps d'execució acostuma a ser menor que en dispositius monoprocessadors.

A la figura 1.5 veiem un esquema del resultat d'una execució de tres processos en paral·lel en un ordinador multiprocessador amb tres processadors. En contraposició tenim la figura 1.2 amb l'execució en un ordinador monoprocessador.

Figura 1.5. Execució en paral·lel



El principal desavantatge de la programació paral·lela són els controls que hem d'afegir per tal que la comunicació i sincronització dels processos que s'executen concurrentment siguin correctes. Dos processos s'han de sincronitzar o comunicar quan un procés necessita alguna dada que està processant l'altre o bé un d'ells ha d'esperar la finalització de l'altre per poder continuar la seva execució.

El llenguatge de programació Java té integrat el suport a la concurrència en el propi llenguatge i no a través de llibreries.

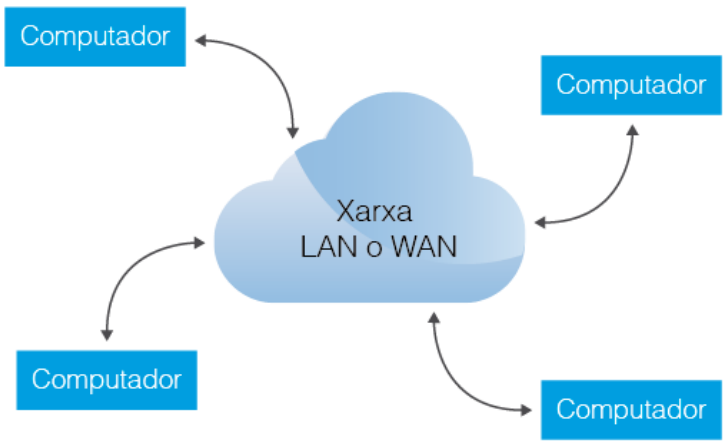
La programació paral·lela està molt lligada a l'arquitectura del sistema de computació. Però quan programem en llenguatges d'alt nivell ens hem d'abstreure de la plataforma en la qual s'executarà. Els llenguatges d'alt nivell ens proveeixen de llibreries que ens faciliten aquesta abstracció i ens permeten utilitzar les mateixes operacions per programar de forma paral·lela que acabaran implementades per usar-se un una plataforma concreta.

Un tipus especial de programació paral·lela és l'anomenada programació distribuïda. Aquesta programació es dóna en sistemes informàtics distribuïts. Un sistema distribuït està format per un conjunt d'ordinadors que poden estar situats en llocs

geogràfics diferents units entre ells a través d'una xarxa de comunicacions. Un exemple de sistema distribuït pot ser el d'un banc amb moltes oficines al món, amb un ordinador central per oficina per guardar els comptes locals i fer les transaccions locals. Aquest ordinador es pot comunicar amb els altres ordinadors centrals de la xarxa d'oficines. Quan es fa una transacció no importa on es troba la compte o el client.

En la figura 1.6 podem veure l'execució d'una programació distribuïda.

Figura 1.6. Execució de programació distribuïda



La **programació distribuïda** és un tipus de programació concurrent en la qual els processos són executats en una xarxa de processadors autònoms o en un sistema distribuït. És un sistema de computadors independents que des del punt de vista de l'usuari del sistema es veu com una sola computadora.

En els sistemes distribuïts, a diferència dels sistemes paral·lels, no existeix memòria compartida, per tant si necessiten la utilització de variables compartides es crearan rèpliques d'aquestes variables als diferents dispositius de la xarxa i caldrà controlar la coherència de les dades.

La comunicació entre processos per intercanviar dades o sincronitzar-se entre ells es fa amb missatges que s'envien a través de la xarxa de comunicacions que comparteixen.

Els principals avantatges són que es tracta de sistemes altament escalables i per tant reconfigurables fàcilment i d'alta disponibilitat. Com a desavantatges més importants, trobarem que són sistemes heterogenis, complexos de sincronitzar. Les comunicacions es fan a través de missatges que utilitzen la xarxa de comunicació compartida i això pot provocar-ne la saturació.

1.1.3 Avantatges i desavantatges del multiprocés

Hem definit programació concurrent o concurrència com la tècnica per la qual múltiples processos s'executen alhora i poden comunicar-se entre ells. La majoria dels sistemes intenten aprofitar aquesta concurrència per incrementar la capacitat d'execució.

Incrementar la potència de càlcul i el rendiment és un dels principals avantatges. Quan s'executen diversos processos a l'hora, la velocitat d'execució global es pot incrementar. Cal tenir en compte, però, que no sempre és així. A vegades, depenent de la complexitat de l'aplicació, les tècniques de sincronisme o comunicació són més costoses en temps que l'execució dels processos.

Un sistema multiprocessador és **flexible**, ja que, si augmenten el processos que s'estan executant és capaç de distribuir la càrrega de treball dels processadors, i pot també reassignar dinàmicament els recursos de memòria i els dispositius per ser més eficients. És de **fàcil creixement**. Si el sistema ho permet, es poden afegir nous processadors de forma senzilla i així augmenten la seva potència.

Els sistemes multiprocessador poden ser sistemes redundants. El fet de disposar de diversos processadors, pot permetre un augment de la disponibilitat dels recursos (més processadors al servei de l'usuari) o bé l'ús de processadors especialitzats en tasques de verificació i control. En el darrer cas parlarem de sistemes amb una alta **tolerància a fallades**. Una fallada d'un processador no fa que el sistema s'aturi.

Els sistemes multiprocés ens permeten diferenciar processos per la seva **especialització** i, per tant, reservar processadors per operacions complexes, aprofitar-ne d'altres per processaments paral·lels i per avançar l'execució.

Els inconvenients del multiprocessament vénen provocats sobretot pel control que s'ha de realitzar quan hi ha diversos processos en execució i han de compartir informació o comunicar-se. Això incrementa la complexitat de la programació i penalitza el temps d'execució.

Si el multiprocessament és en un entorn multiprocessador paral·lel o distribuït, el tràfic dels busos de comunicació s'incrementa paral·lel al número de processadors o computadors que incorporem al sistema, i això pot arribar a ser un crític coll d'ampolla.

1.2 Processos i serveis

La majoria de nosaltres tenim a l'ordinador un antivirus instal·lat. Quan posem en marxa el nostre ordinador no arranquem l'antivirus i no interactuem amb ell a no ser que trobi un virus i ens avisi. A l'iniciar el sistema, el programa d'antivirus s'executa. Llavors es crea un procés que es manté en execució fins que apaguem el nostre ordinador. Aquest procés que controla les infeccions de tots els fitxers que entren al nostre ordinador és un **servei**.

Un servei és un tipus de procés que no té interfície amb l'usuari. Poden ser, depenent de la seva configuració, inicialitzats pel sistema de forma automàtica, en el qual van realitzant les seves funcions sense que l'usuari se n'assabenti o bé es poden mantenir a l'espera que algú els faci una petició per fer una tasca en concret.

Depenent de com s'estan executant, els processos es classificaran com a processos en primer pla (*foreground*, en anglès) i processos en segon pla (*background*). Els processos en primer pla mantenen una comunicació amb l'usuari, ja sigui informant de les accions realitzades o esperant les seves peticions per mitjà d'alguna interfície d'usuari. En canvi, els que s'executen en segon pla no es mostren explícitament a l'usuari, bé perquè no els cal la seva intervenció o bé perquè les dades requerides no s'incorporen a través d'una interfície d'usuari.

El nom *deamon* prové de les sigles angleses DAEMON (*Disk And Execution Monitor*). Sovint a la literatura tècnica s'ha estès aquest concepte usant la paraula "dimoni", pel que podeu trobar aquest mot fent referencia als programes que s'executen en segon pla.

Els serveis són processos executats en segon pla. *Servei* és una nomenclatura utilitzada en Windows. En sistemes Linux s'anomenen també processos *deamon*.

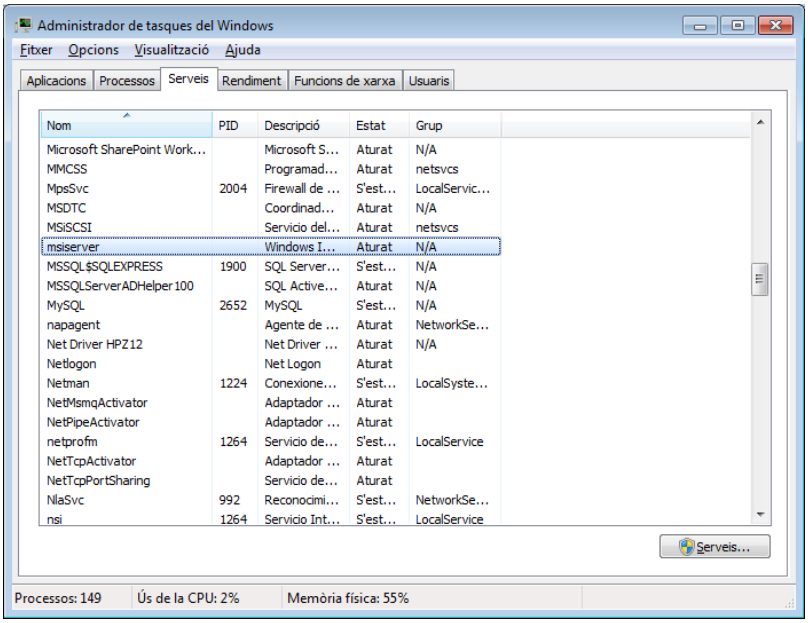
A Linux el nom dels *daemons* sempre acaben amb la lletra *d*, com ara httpd, el *daemon* de l'http.

Com que els serveis no disposen d'interfície d'usuari directa, emmagatzemen la informació generada o les possibles errades que vagin produint, en fitxers de registre habitualment coneguts com a *logs*.

Alguns serveis poden estar a l'espera de ser cridats per realitzar les seves funcions. Per exemple, un servidor web té un servei actiu. A Linux és un httpd el que està escoltant un port de la xarxa i quan l'usuari fa una petició a través de la xarxa per obtenir una plana web, és el dimoni el que s'encarrega de processar la petició i enviar la plana de retorn, si té accés autoritzat i la plana existeix, o bé fer arribar a través de la xarxa el missatge informatiu indicant la impossibilitat del lliurament.

A la figura 1.7 podem veure els serveis executats en un ordinador Windows.

FIGURA 1.7. Serveis Windows



Els serveis poden estar en execució local, al nostre ordinador, com ara el Firewall, l'antivirus, la connexió Wi-Fi, o els processos que controlen el nostre ordinador, o bé poden executar-se en algun ordinador d'un sistema informàtic distribuït o a Internet, etc. Les crides als serveis distribuïts es realitzen sobre protocols de comunicació. Per exemple, els serveis d'http, DNS (Domain Name System), FTP (File Transfer Protocol) són serveis que ens ofereixen servidors que són a Internet.

1.2.1 Fils i processos

Recordem el cuiner que treballa simultàniament preparant un plat per diferents comensals. El processador d'un sistema informàtic (cuiner) està executant diferents instàncies del mateix programa (la recepta). A banda, el cuiner va fent diferents parts del plat. Si la recepta és botifarra amb patates, dividirà la seva tasca en preparar la botifarra per una banda i les patates per l'altra. Ha dividit el procés en dos subprocessos. Cada un d'aquests subprocessos s'anomenen **fils**.

El sistema operatiu pot mantenir en execució diversos processos a la vegada fent servir concurrència o paral·lelisme. A més, dins de cada procés poden executar-se diversos fils, de manera que blocs d'instruccions que presentin certa independència puguin executar-se a la vegada. Els fils comparteixen els recursos del procés (dades, codi, memòria, etc). En conseqüència, els fils, a diferència dels processos, comparteixen memòria i si un fil modifica una variable del procés, la resta de fils podran veure la modificació quan accedeixin a la variable.

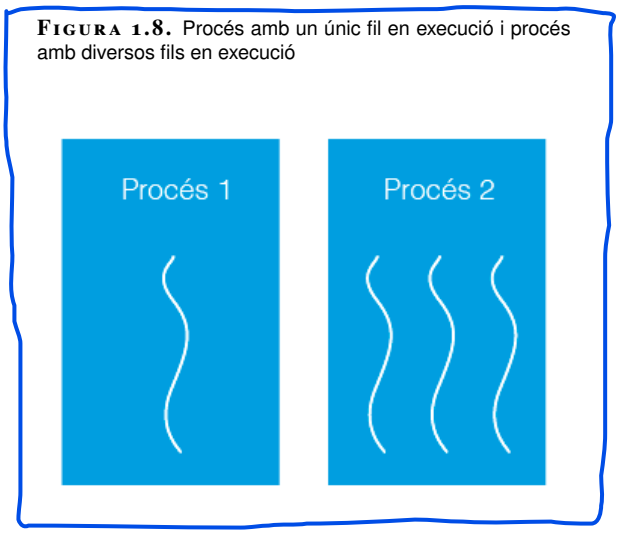
Els processos són anomenats **entitats pesades** perquè estan a espais de direccionament de memòria independents, de creació i de comunicació entre processos, cosa que consumeix molts recursos de processador. En canvi, ni la creació de fils ni la comunicació consumeixen gaire temps de processador. Per aquest motiu els fils s'anomenen **entitats lleugeres**.

Un procés estarà en execució mentre algun dels seus fils estigui actiu. Tot i així, també és cert que si finalitzem un procés de forma forçada, els seus fils també acabaran l'execució.

Podem parlar de dos nivells de fils: els fils de nivell d'usuari, que són els que creem quan programem amb un llenguatge de programació com ara Java; i els fils de segon nivell que són els que crea el sistema operatiu per donar suport als primers. Nosaltres programarem els nostres fils utilitzant unes llibreries que ens proporciona el llenguatge de programació. Són les llibreries amb les instruccions pertinents que indicaran al sistema els fils que han de crear i com gestionar-los.

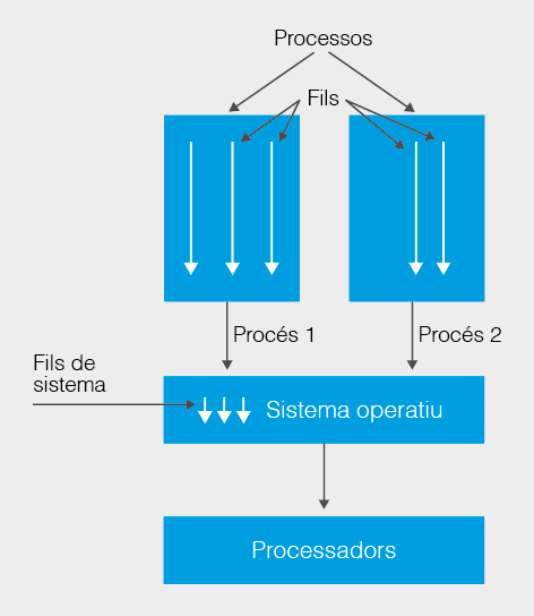
A la figura 2.8 es mostren dos processos un amb l'execució d'un fil i l'altre amb l'execució de tres fils concurrents.

FIGURA 1.8. Procés amb un únic fil en execució i procés amb diversos fils en execució



La figura 1.9 il·lustra el tractament que el sistema operatiu fa dels fils. En referència al processament, cada fil es processa de forma independent malgrat que pertanyi o no a un mateix procés. L'única diferència entre el tractament de fils i processos la trobem a nivell de memòria. Per a fils d'un mateix procés el sistema operatiu ha de mantenir les mateixes dades en memòria. Per a fils de diferents processos en canvi cal disposar de dades independents. En el cas que sigui necessari que un mateix processador alterni l'execució de diversos processos, caldrà restaurar la memòria específica del procés en cada canvi. D'això s'anomena també canvi de context. Si l'alternança de processament es fa entre fils d'un mateix procés, no caldrà restaurar la memòria i per tant el procés serà molt més eficient.

FIGURA 1.9. Dos processos amb diferents fils en execució. Cada fil s'executa de forma independent. Els fils del mateix procés comparteixen memòria.



1.3 Gestió de processos i desenvolupament d'aplicacions amb finalitat de computació paral·lela

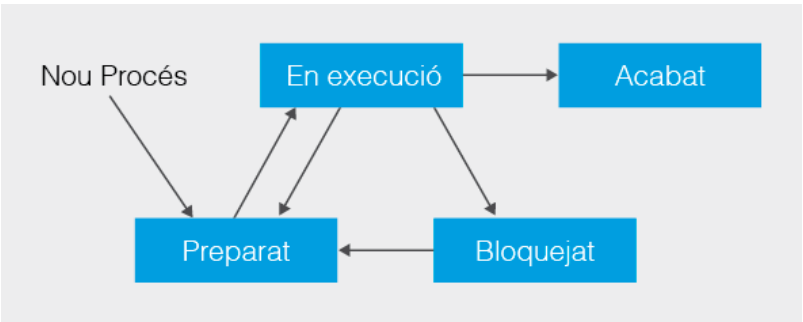
Un del objectius del sistema operatiu és proporcionar als processos els recursos necessaris per la seva execució. Ha d'aconseguir una política de planificació que determini quin procés farà ús del processador. El sistema operatiu és, doncs, l'encarregat de decidir quin procés cal executar i durant quant de temps cal fer-ho. Sortosament, els processos contemplen moltes estones de repòs en què no els cal fer servir el processador, per exemple esperant dades. Si en iniciar l'espera el procés avisa al sistema operatiu, aquest podrà reemplaçar el procés actiu per un altre en disposició de fer servir el processador. En acabar el temps de repòs, els processos hauran d'avisar de nou al sistema per tal que planifiqui una futura execució quan el processador quedi lliure. Quan finalment els processos finalitzin l'execució, el sistema operatiu alliberarà els recursos utilitzats per a l'execució i deixarà lliure el processador. El temps des que un procés inicia l'execució fins que la finalitza s'anomena cicle de vida del procés. Per tal que el sistema pugui gestionar les necessitats dels processos durant tot el seu cicle de vida, els identificarà etiquetant-los amb una estat que n'indiqui les necessitats de processament.

1.3.1 Estats d'un procés

Tots els sistemes operatius disposen d'un **planificador de processos** encarregat de repartir l'ús del processador de la forma més eficient possible i assegurant que tots els processos s'executin en algun moment. Per realitzar la planificació, el sistema operatiu es basarà en l'estat dels processos per saber quins necessitaran l'ús del processador. Els processos en disposició de ser executats s'organitzaran en una cua esperant el seu torn.

Depenent del sistema operatiu i de si s'està executant en un procés o un fil, el nombre d'estats pot variar. El diagrama d'estats de la figura 1.10 mostra els estats més habituals.

FIGURA 1.10. Estats d'un procés



El primer estat que ens trobem és **nou**, és a dir, quan un procés és creat. Un cop creat, passa a l'estat de **preparat**. En aquest moment el procés està preparat per fer

ús del processador, està competint pel recurs del processador. El planificador de processos del sistema operatiu és el que decideix quan entra el procés a executar-se. Quan el procés s'està executat, el seu estat s'anomena **en execució**. Un altre cop, és el planificador l'encarregat de decidir quan abandona el processador.

Assignar un temps d'execució fix a cada procés i un cop acabat aquest temps canviar el procés de l'estat d'execució a l'estat de preparat a tot esperant de ser assignat de nou al processador, pot ser una política del planificador de processos per assignar els diferents torns d'execució.

> El planificador de processos segueix una política que indica quan un procés ha de canviar d'estat, quan ha de deixar lliure o entrar a fer ús del processador.

Quan un procés abandoni el processador, perquè el planificador de processos així ho haurà decidit, canviarà a l'estat de preparat i es quedarà esperant que el planificador li assigni el torn per aconseguir de nou el processador i tornar a l'estat d'execució.

Des de l'estat d'execució, un procés pot passar a l'estat de **bloquejat o en espera**. En aquest estat, el procés estarà a l'espera d'un esdeveniment, com pot ser una operació d'entrada/sortida, o l'espera de la finalització d'un altre procés, etc. Quan l'esdeveniment esperat succeeixi, el procés tornarà a l'estat de preparat, guardant el torn amb la resta de processos preparats.

L'últim estat és **acabat**. És un estat al qual s'hi arriba un cop el procés ha finalitzat tota la seva execució i estarà a punt per tal que el sistema n'alliberi quan pugui els recursos associats.

Les transicions de l'estat d'un procés poden ser provocades per algun dels motius següents:

- D'execució a bloquejat: el procés realitza alguna operació d'entrada i sortida o el procés ha d'esperar que un altre procés modifiqui alguna dada o alliberi algun recurs que necessita.
- D'execució a preparat: el sistema operatiu decideix treure un procés del processador perquè ha estat un temps excessiu fent-ne ús i el passa a l'estat de preparat. Assigna a un altre procés l'accés al processador.
- De preparat a execució: és el planificador de processos, depenent de la política que practiqui, l'encarregat de fer aquest canvi.
- De bloquejat a preparat: el recurs o la dada pel la qual havia estat bloquejat està disponible o ha acabat l'operació d'entrada i sortida.
- D'execució a acabat: el procés finalitza les seves operacions o bé un altre procés o el sistema operatiu el fan acabar.

Cada cop que un procés canvia a l'estat d'execució, s'anomena **canvi de context**, perquè el sistema operatiu ha d'emmagatzemar els resultats parcials aconseguits durant l'execució del procés sortint i cal que carregui els resultats parcials de la darrera execució del procés entrant en els registres del processador, assegurant l'accessibilitat a les variables i al codi que toqui seguir executant.

1.3.2 Planificació de processos

La planificació de processos és un conjunt de protocols o polítiques que decideixen en quin ordre s'executaran els processos al processador. Aquestes polítiques segueixen alguns criteris de prioritat segons la importància del procés, el temps d'utilització del processador, el temps de resposta del procés, etc.

La funció de planificació (*scheduler* en anglès) del nucli del sistema operatiu, és l'encarregada de decidir quin procés entra al processador, pot dividir-se en tres nivells:

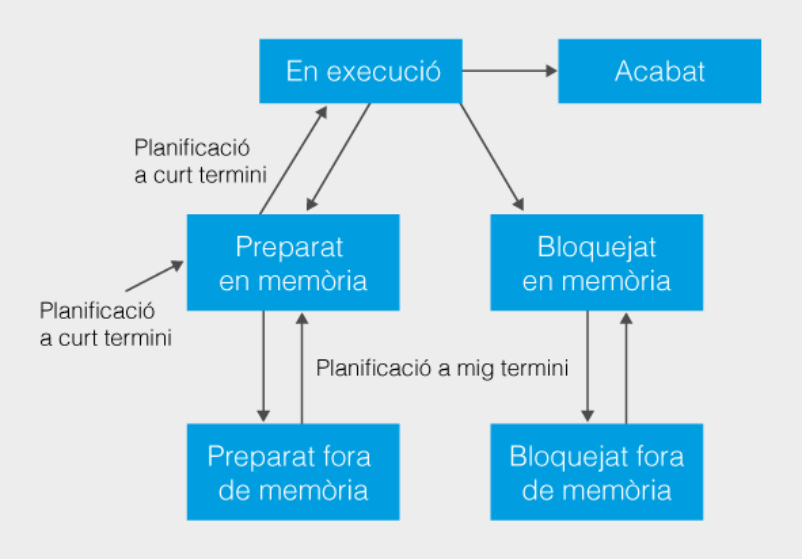
1. **Planificació a llarg termini**: en el moment que es crea un procés es decideixen alguns criteris de planificació, com ara la unitat de temps màxim que un procés podrà romandre en execució (que s'anomena **quantum**) o la prioritat inicial que se li assignarà a cada procés de forma dinàmica per la utilització del processador.

2. **Planificació a curt termini**: cada vegada que un procés abandona el processador cal prendre la decisió de quin serà el nou procés que entrarà a fer-ne ús. En aquest cas les polítiques intenten minimitzar el temps necessari per aconseguir un canvi de context. Els processos recentment executats solen tenir prioritat però cal assegurar que mai s'exclogui permanentment cap procés de l'execució.

3. **Planificació a mig termini**: existeixen altres parts del sistema operatiu que també són importants a la planificació de processos. En el cas del SWAP de memòria, si es treu un procés de la memòria per problemes d'espai fa que no pugui ser planificable de forma immediata. El planificador a mig termini serà l'encarregat de decidir quan cal portar a la memòria principal les dades dels processos emmagatzemats fora i quines caldrà traslladar de la memòria principal a l'espai d'intercanvi.

La figura 1.11 mostra gràficament els nivells de planificació de processos.

FIGURA 1.11. Planificació de processos



Els sistemes multiprocessadors presenten a més dos components de planificació específics.

El component de **planificació temporal** en el qual es troba definida la política de planificació que actua sobre cada processador de forma individual, com si fos un sistema monoprocessador. El planificador temporal decideix quan de temps cal dedicar a processar cada procés, quan cal fer canviar, per a quin motiu, etc.

El component de **planificació espacial**, en el qual es defineix com es reparteixen els processos entre els processadors. És a dir, organitza quin processador executa quin procés.

Aquesta planificació en els multiprocessadors utilitza alguns criteris per aplicar les polítiques d'assignació. El planificador va memoritzant quins són el processos executats sobre cada processador i decideix si serà adequat que torni a executar-se sobre el mateix. Una política d'**afinitat al processador** pot provocar una sobrecàrrega en alguns processadors i fer que l'execució no sigui massa eficient. Per contra una política de **repartiment de càrrega** evita aquesta suposada infrautilització d'alguns processadors, però augmenta les sobrecàrregues a la memòria compartida.

1.3.3 Programació paral·lela transparent

Quan busquem resultats ràpidament o molta potència de càlcul podem pensar en augmentar el número de processadors de la nostra computadora i executar una aplicació de forma paral·lela als diferents processadors. Però l'execució en paral·lel no és sempre possible. En primer lloc perquè hi ha aplicacions que no es poden paral·lelitzar, sinó que la seva execució ha de ser seqüencial. I en segon lloc, per la dificultat per desenvolupar entorns de programació a sistemes paral·lels, ja que es requereix un esforç important pels programadors.

Però, i si en lloc d'aplicar la concurrència explícita, en la qual el programador aporta a l'algoritme les pautes del paral·lelisme, s'aplica una programació de concurrència implícita, en la qual és el sistema operatiu el que decideix quins processos es poden dividir i quins no, aplicant certes regles genèriques i comuns a la majoria de problemes? El programador només hauria d'escollir les regles.

S'han generalitzat els sistemes informàtics multinuclis a servidors, ordinadors de sobretaula i també a sistemes mòbils, *smartphones*, tauletes, etc. Tots aquests dispositius ajuden a potenciar la programació concurrent i paral·lela. Però la programació paral·lela és complexa, cal sincronitzar processos i controlar dades compartides, cosa que afegeix complexitat a la tasca pel programador.

La plataforma Java a Java SE 6 introdueix un conjunt de paquets que proporcionen suport a la programació concurrent i Java SE 7 millora encara més el suport a la programació en paral·lel.

Java proporciona suport a la programació concurrent amb llibreries de baix nivell a través de la classe `java.lang.Thread` i de la interfície `java.lang.Runnable`. Hi ha problemes, però, que poden programar-se de forma paral·lela seguint patrons semblants. L'ús de `Thread` i `Runnable` comporta un esforç addicional del programador, obligant-lo a afegir proves extres per verificar el comportament correcte del codi, evitant lectures i escriptures errònies, sincronitzant les execucions dels processos i evitant els problemes derivats dels bloquejos.

Una aproximació relativament senzilla a alguns d'aquests problemes que poden resoldre's de forma paral·lela amb implantacions típiques que presenten un cert nivell de patronatge són els marcs predefinits de programació paral·lela del llenguatge a Java: **Executor** i **fork-join**. Són llibreries que encapsulen bona part de la funcionalitat i amb ella de la complexitat addicional que la programació de més baix nivell (`Threads`) presenta.

Ús de patrons per resoldre problemes similars

En enginyeria, anomenem patrons a aquelles solucions genèriques que ens permeten resoldre diversos problemes similars canviant petites coses. L'enginyeria informàtica fa servir aquest concepte per referir-se a biblioteques complexes que permeten un alt grau de configuració malgrat que per això sigui necessari escriure aquelles parts específiques que adaptaran el patró al problema que s'intenti resoldre.

Java fa servir aquest concepte en moltes de les seves biblioteques tot definint una estructura base a través d'interfícies i classes genèriques. El programador pot reutilitzar aquesta estructura i el codi ja implementat, usant-la en diferents situacions i adaptant-la amb codi específic per aquelles definicions abstractes i classes que restin indefinides en la implementació de la biblioteca.

La biblioteca `Executor` és una d'aquestes que implementa diversos patrons amb l'objectiu de solucionar de la forma més transparent possible bona part de les situacions en les quals es necessita programació paral·lela. Cada patró s'adapta

a diferents situacions per la qual cosa cal saber distingir les situacions i aplicar correctament el patró que calgui.

Executor

Java ens dóna eines per la simplificació de la programació paral·lela. La idea és dividir un problema en problemes més petits i cadascun d'aquests subproblemes enviar-los a executar per separat. Per tant, d'una tasca complicada en tenim diverses de més simples. Per poder executar en paral·lel totes aquestes subtasques i així augmentar el rendiment, es creen diferents subprocessos o fils, però aquesta creació és totalment transparent pel programador, és la màquina virtual de Java l'encarregada de gestionar la creació de fils per executar en paral·lel les subtasques.

Executor és una millora sobre la creació de fils, ja que s'abstreu de la creació i la gestió dels subprocessos, De fet només cal indicar la tasca o tasques a realitzar de forma paral·lela (instant objectes de tipus Runnable o Callable), escollir el gestor adequat i deixar que aquest s'encarregui de tot.

A grans trets, podem dir que disposem de tres tipus de gestors, un de genèric (`ThreadPoolExecutor`), un capaç de seguir pautes temporals d'execució (`ScheduledThreadPoolExecutor`) i una darrer que per la seva especificitat i complexitat l'estudiarem a banda. És l'anomenat marc Fork-Join.

`ThreadPoolExecutor`, és la classe base que permet implementar el gestor genèric. Els gestors genèrics són adequats per solucionar problemes que es puguin resoldre executant tasques independents entre si. Imaginem que volem saber la suma d'una llista gran de nombres. Podem optar per sumar seqüencialment la llista o bé per dividir-la en dos o tres trossos i sumar cada tros per separat. Al final només caldrà afegir el valor de cada tros al resultat. La suma de cada tros és totalment independent i per tant pot realitzar-se amb independència de la resta. És a dir, només caldria crear tantes tasques com trossos a sumar i passar-les al gestor perquè les executi.

`ThreadPoolExecutor`, no crea pas un fil per cada tasca (instància de `Callable` o `Runnable`). De fet, és habitual assignar-li més tasques que no pas fils. Les tasques pendents romanen en una cua de tasques i un nombre de fils determinat s'encarrega d'anar-les executant. Quants fils cal crear? Segur que aquesta és un decisió difícil i dependrà de cada cas. Per això JAVA ens ofereix 3 formes ràpides de configurar les instàncies de `ThreadPoolExecutor` fent servir la classe `Executors`. Es tracta d'una classe que aglutina un conjunt d'utilitats en forma de mètodes estàtics. Els tres tipus s'obtenen a partir dels mètodes estàtics següents:

- `newCachedThreadPool()`: crea un *pool* que va creant fils a mesura que els necessita i reutilitza els fils inactius.

- `newFixedThreadPool(int numThreads)`: crea un *pool* amb un número de fils fix. L'indicat en el paràmetre. Les tasques s'emmmagatzemen en una cua de tasques i es van associant als fils a mida que aquest quedin lliures.

- `newSingleThreadExecutor()`: crea un *pool* amb un sol fil que serà el que processi totes les tasques.

Qualsevol d'aquestes configuracions obtindrà una instància de `ThreadPoolExecutor` que es comportarà com indiquen els comentaris. Per poder afegir noves tasques, iniciar el processament paral·lel i mantenir-lo actiu fins acabar haver processat totes les tasques, la classe `ThreadPoolExecutor` disposa de mètodes com `execute` per anar introduint una tasca (instància de `Runnable` o `Callable`) cada cop, `invokeAll` per afegir, tot d'una, una llista de diverses tasques, `getPoolSize()` per obtenir el nombre fils en execució o també `getCompleteTaskCount()` per saber el nombre de tasques finalitzades.

`ScheduledThreadPoolExecutor` és un cas específic de *pool*. Es tracta d'un gestor de fils, amb una política d'execució associada a una seqüència temporal. És a dir executa les tasques programades cada cert temps. Pot ser d'una sola vegada o de forma repetitiva. Per exemple si volem consultar el nostre correu cada 5 minuts, mentre realitzem altres feines, podem programar un fil independent amb aquesta tasca usant `ScheduledThreadPoolExecutor`. La tasca podria comprovar el correu i avisar només en cas que hagin arribat nous. D'aquesta manera podem concentrar-nos en la nostra feina deixant que el fil periòdic treballi per nosaltres. Podem obtenir instàncies de `ScheduledThreadPoolExecutor` usant una utilitat de la classe `Executor`, el mètode *static* `newScheduledThreadPool(int numThreads)`.

La classe `ScheduledThreadPoolExecutor` disposa dels mètodes següents per gestionar el retard i cadència:

- `schedule (Callable task, long delay, TimeUnit timeunit)`: crea i executa una tasca (*task*) que és invocada després d'un temps concret (*delay*) expressat en les unitats indicades (*timeunit*).
- `schedule (Runnable task, long delay, TimeUnit timeunit)`: crea i executa una acció (*task*), una sola vegada, després que hagi transcorregut un temps concret (*delay*) expressat en les unitat indicades (*timeunit*).
- `scheduleAtFixedRate (Runnable task, long initialDelay, long period, TimeUnit timeunit)`: crea i executa una tasca (*task*) de forma periòdica. La primera vegada s'invocarà després d'un retard inicial (*initialDelay*) i posteriorment cada període de temps determinat (*period*). El temps es mesura en les unitats indicades (*timeunit*).
- `scheduleWithFixedDelay (Runnable task, long initialDelay, long delay, TimeUnit timeunit)`: crea i executa una tasca (*task*) de forma periòdica. La primera vegada s'invocarà després d'un retard inicial (*initialDelay*) i posteriorment, en acabar la darrera tasca llançada, no es començarà la següent, fins que no hagi passat el temps indicat a *delay*. El temps es mesura en les unitats indicades (*timeunit*).

El mètodes que reben tasques de tipus `Callable` retornaran el resultat obtingut durant el càlcul.

Exemple amb ThreadPoolExecutor

Anem ara a veure un exemple per calcular deu multiplicacions aleatòries d'enters usant un `Executor` amb un màxim de 3 fils. Per a la instanciació del gestor caldrà:

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.
    newFixedThreadPool(3);
```

Això significa que si hi ha més de 3 tasques per executar, únicament n'enviarà 3 a executar i la resta es quedaran bloquejades fins que un fil acabi el seu processament.

La classe interior `Multiplicacio` implementa la interfície `Callable`. Es podria fer servir el mètode *run* i implementar `Runable`. La diferència és que `Callable` retorna valors del resultat en forma de qualsevol objecte. La concurrència es realitza dins del mètode `call()`.

Es crea una llista de deu multiplicacions a l'atzar. I després crida al mètode d'executor `invokeAll()`. Aquest rep una col·lecció de tasques de tipus `Callable`, les executa i en retorna el resultat dins d'un objecte `Future`. La classe `java.util.concurrent.Future` és una interfície dissenyada per donar suport a les classes que mantinguin dades calculades de forma asíncrona. Les instàncies de `Future` obliguen a fer servir els mètodes `get` per accedir als resultats. Això assegura que el valor real només serà accessible quan els càlculs hagin acabat. Mentre durin els càlculs la invocació del mètode `get` implicarà un bloqueig del fil que l'estigui invocant.

Els objectes Future es generen com a conseqüència d'executar el mètode `call` de les instàncies `Callable`. Per cada `Callable` invocada es genera un objecte Future que romandrà bloquejat fins que la instància Callable associada finalitzi.

```
package multiplicallista;

import java.util.*;
import java.util.concurrent.*;

public class MultiplicaLlista {

static class Multiplicacio implements Callable<Integer> {
 private int operador1;
 private int operador2;
 public Multiplicacio(int operador1, int operador2) {
   this.operador1 = operador1;
   this.operador2 = operador2;
 }
 @Override
 public Integer call() throws Exception {
     return operador1 * operador2;
 }
}
  public static void main(String[] args) throws
       InterruptedException, ExecutionException {

     ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors
         .newFixedThreadPool(3);
     List<Multiplicacio> llistaTasques= new ArrayList<
         Multiplicacio>();
     for (int i = 0; i < 10; i++) {
       Multiplicacio calcula = new Multiplicacio((int)(Math.
           random()*10), (int)(Math.random()*10));
       llistaTasques.add(calcula);
     }
     List <Future<Integer>> llistaResultats;
     llistaResultats = executor.invokeAll(llistaTasques);

     executor.shutdown();
```

```
32      for (int i = 0; i < llistaResultats.size(); i++) {
33            Future<Integer> resultat = llistaResultats.get(i);
34            try {
35              System.out.println("Resultat tasca "+i+ " és:" +
                     resultat.get());
36            } catch (InterruptedException | ExecutionException e)
                  {
37            }
38        }
39
40    }
41  }
```

`ThreadPoolExecutor` disposa també del mètode `invokeAny(tasques)` que rep una col·lecció d'instàncies de `Callable` que anirà executant fins que una d'elles finalitzi sense error. En aquest moment retornarà el resultat obtingut per la tasca finalitzada en un objecte `Future`.

Un altre mètode de `ThreadPoolExecutor` que cal conèixer és `shutdown()`, que impedeix executar noves tasques però deixa que els fils que estan en execució puguin acabar.

Exemple amb ScheduledThreadPoolExecutor

En el següent exemple utilitzarem l'executor `ScheduledThreadPoolExecutor` per a programar tasques que s'executin periòdicament, després d'un retard programat.

Primer creem un *pool* de dos fils del `ScheduledThreadPoolExecutor` i utilitzem `scheduleWithFixedDelay`, que ens permet programar i executar l'inici de la tasca (`initialDelay`) i programar les tasques successives després de la primera finalització (*long period*).

Executa el mètode *run* d' `ExecutaFil`. La primera vegada al cap de 2 segons i posteriorment cada 3 segons.

Per últim, el mètode `awaitTermination`es queda a l'espera abans del *shutdown* (aturada), fins que totes les tasques hagin acabat, es produeixi el *timeout* (temps d'espera) o els fils siguin finalitzats abruptament, el que passi primer.

```
1   package tascaprogramada;
2
3   import java.util.Calendar;
4   import java.util.GregorianCalendar;
5   import java.util.concurrent.ExecutionException;
6   import java.util.concurrent.Executors;
7   import java.util.concurrent.ScheduledExecutorService;
8   import java.util.concurrent.TimeUnit;
9
10  public class TascaProgramada {
11
12      public static void main(final String... args) throws
            InterruptedException, ExecutionException {
13         //mostrem hora actual abans d'execució
14          Calendar calendario = new GregorianCalendar();
15         System.out.println("Inici: "+ calendario.get(Calendar.
               HOUR_OF_DAY) + ":" + calendario.get(Calendar.MINUTE) +
                ":" + calendario.get(Calendar.SECOND));
16
17          // Crea un pool de 2 fils
18          final ScheduledExecutorService schExService = Executors.
                newScheduledThreadPool(2);
19          // Crea objecte Runnable.
20          final Runnable ob = new TascaProgramada().new ExecutaFil
                ();
21          // Programa Fil, s'inicia als 2 segons i després es va
                executant cada 3 segons
22          schExService.scheduleWithFixedDelay(ob, 2, 3, TimeUnit.
                SECONDS);
```

```
        // Espera per acabar 10 segons
        schExService.awaitTermination(10, TimeUnit.SECONDS);
        // shutdown .
        schExService.shutdownNow();
        System.out.println("Completat");
    }
    // Fil Runnable
    class ExecutaFil implements Runnable {
        @Override
        public void run() {
            Calendar calendario = new GregorianCalendar();
            System.out.println("Hora execució tasca: "+
                calendario.get(Calendar.HOUR_OF_DAY) + ":" +
                calendario.get(Calendar.MINUTE) + ":" +
                calendario.get(Calendar.SECOND));
            System.out.println("Tasca en execució");

            System.out.println("Execució acabada");
        }
    }


}
```

Quan executem el codi anterior obtindrem per la pantalla uns resultats semblants als següents:

run:

Inici: 11:27:10

Hora execució tasca: 11:27:12

Tasca en execució

Execució acabada

Hora execució tasca: 11:27:15

Tasca en execució

Execució acabada

Hora execució tasca: 11:27:18

Tasca en execució

Execució acabada

Completat

BUILD SUCCESSFUL (total time: 10 seconds)

Fixeu-vos com la primera execució es produeix després de 2 segons, a continuació la tasca s'executa cada 3 segons.

Fork-Join

El marc Fork-Join és una evolució dels patrons Executor. De fet aquí trobem també un gestor de fils (la classe `ForkJoinPool`), però una mica més sofisticat. Processa les tasques amb l'algoritme per robatori (*work-stealing*). Això vol dir que el gestor del *pool* busca fils poc actius i hi intercanvia tasques en espera. A més les tasques poden crear noves tasques que s'incorporaran a la cua de tasques pendents per a

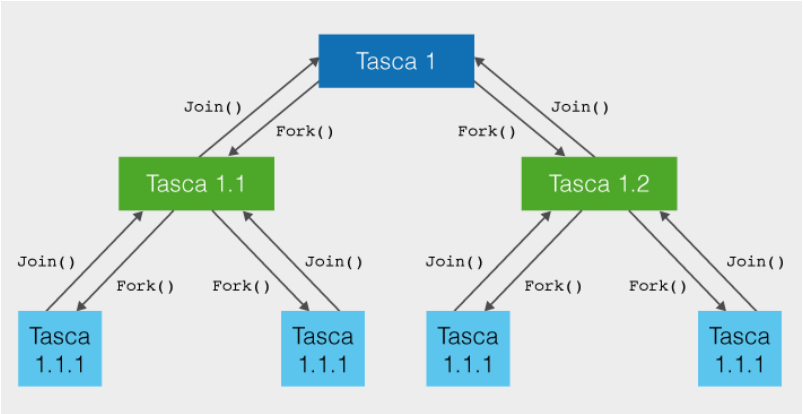
ser executades. L'eficiència aconseguida és força alta, ja que s'aprofita al màxim el processament paral·lel. Per aquest motiu Fork_Join és una alternativa ideal per algoritmes que puguin resoldre's de forma recursiva ja que s'aconseguirà la màxima eficiència intercanviant aquelles tasques que es trobin esperant resultats per d'altres que permetin avançar en els càlculs.

La classe `ForkJoinTask` és una classe abstracta per les tasques que s'executen a `ForkJoinPool` i conté els mètodes `fork` i `join`. El mètode `fork()` situa la tasca invocada a la cua d'execucions en qualsevol moment per tal que sigui planificada. Això permet que a una tasca crear-ne de noves i deixar-les a punt per ser executades quan el gestor ho consideri.

El mètode `join()` aturarà l'execució del fil invocador a l'espera que la tasca invocada finalitzi l'execució i retorni si fos el cas els resultats. El bloqueig del fil posarà en alerta el gestor `ForkJoinPool` que podrà intercanviar la tasca aturada per una altra que resti en espera.

A la figura 1.12 es mostra com cooperen les tasques a través dels mètodes `fork()` i `join()`.

FIGURA 1.12. Mètodes fork-join



`ForkJoinTask` està implementada per dues classes que l'especialitzen, la classe `RecursiveTask` i la classe `RecursiveAction`. La primera és adequada per a tasques que necessitin calcular i retornar un valor. La segona en canvi representa procediments o accions que no necessiten retornar cap resultat.

Quan fem servir `RecursiveAction` resulta molt pràctic fer servir el mètode `invokeAll`, en comptes de cridar, per cada tasca generada els mètodes `fork` i `join`. Concretament el mètode `invokeAll` fa un invocació de `fork` per cada una de les tasques rebudes com a paràmetre i seguidament s'espera a la finalització de cada fil amb una crida al mètode `join` de les tasques engegades. `invokeAll` no retorna cap resultat, per tant no resulta gaire útil d'utilitzar amb objectes de tipus `RecursiveTask`.

Algoritmes recursius i algoritmes iteratius

Els algoritmes recursius tenen la peculiaritat de poder-se resoldre realitzant diverses vegades una mateixa operació però usant valors diferents. Cada execució s'encarrega de realitzar un càlcul parcial fent servir les dades d'una o més

execucions anteriors de la mateixa operació. El darrer resultat parcial calculat coincidirà amb la solució final. Per tal que el càlcul recursiu sigui factible és necessari que existeixi un solució trivial per algun dels valors a operar. Per exemple l'agoritme que calcula el valor factorial d'un número és recursiu perquè podem dir que el valor factorial d'un enter positiu és: *n! = n* (n-1)!* excepte quan *n* val 1 (el cas trivial) ja que sabem que *1! = 1*.

Usant un llenguatge de programació podem expressar aquest algoritme recursiu fent:

```
long factoria(long n){
    if(n==1){
       return n;
    }else{
       return n*factorial(n-1);
    }
}
```

Cal recordar que els algoritmes recursius consumeixen molta memòria i que en un processament seqüencial, poden resultar poc eficients si el seu càlcul requereix moltes crides recursives. Però qualsevol algoritme recursiu pot transformar-se sempre en iteratiu. En un processament seqüencial, els algoritmes iteratius són més eficients que els recursius. Ara bé, és possible aprofitar el processament paral·lel i les característiques recursives d'alguns algoritmes per aconseguir executar en paral·lel les crides recursives de manera que incrementem l'eficiència final. Malauradament, l'increment d'eficiència només serà efectiu si el càlcul a realitzar té prou entitat.

Per donar l'entitat idònia als càlculs i agilitzar l'eficiència, es combina el càlcul iteratiu amb el recursiu. És a dir, es defineix un llindar o condició a partir de la qual seria preferible resoldre el problema iterativament. Mentre no s'arribi al llindar el càlcul es derivarà a la versió recursiva executada en paral·lel, fent servir el marc Fork-Join. Quan arribem al llindar, en canvi, realitzarem el càlcul seqüencialment. L'elecció del llindar no és trivial perquè d'ella dependrà en bona part disposar d'un algoritme eficient o no.

Així doncs, l'esquema d'execució de les tasques a implementar a les instàncies de ForkJoinTask seria similar al codi següent:

```
if (el que queda per resoldre surt més a compte resoldre-ho directament ){ //
    lindar
  execució seqüencial
}else{
  divisió recursiva de la tasca.
  crida/planificació de l'execució paral·lela de les tasques creades
  recollida dels resultats i combinació adequada per obtenir el calcul desitjat
      .
}
```

Veiem el que acabem d'explicar amb un exemple. Buscarem dins d'un *array* d'enters el número més gran. La recursió es basarà en dividir la col·lecció d'enters en dos i demanar per cada tros l'obtenció del valor mínim i el càlcul seqüencial en un algoritme de cerca iteratiu.

Usarem RecursiveTask perquè ens cal retornar el valor mínim. Els valors de partida els passarem sempre a través del constructor de la tasca i els emmagat-

zemarem com atributs per tenir-los disponibles en el mètode `compute` que és l'equivalent específic de les tasques `ForkJoinTask` al `run` o `call` ja vistos en parlar de l'`Executor`.

```
package ioc.dam.m09.u1.forkjoin.maxim;

import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class MaximTask extends RecursiveTask<Short> {
    //private static final int LLINDAR=10000000;
    private static final int LLINDAR=10000000;
    private short[] arr ;
    private int inici, fi;

    public MaximTask(short[] arr, int inici, int fi) {
            this.arr = arr;
            this.inici = inici;
            this.fi = fi;
    }

    private short getMaxSeq(){
        short max = arr[inici];
        for (int i = inici+1; i < fi; i++) {
            if (arr[i] > max) {
                    max = arr[i];
            }
        }
        return max;
    }

    private short getMaxReq(){
        MaximTask task1;
        MaximTask task2;
        int mig = (inici+fi)/2+1;
        task1 = new MaximTask(arr, inici, mig);
        task1.fork();
        task2 = new MaximTask(arr, mig, fi);
        task2.fork();
        return (short) Math.max(task1.join(), task2.join());
    }

    @Override
    protected Short compute() {
        if(fi – inici <= LLINDAR){
            return getMaxSeq();
        }else{
            return getMaxReq();
        }
    }

    public static void main(String[] args) {

        short[] data = createArray(100000000);

        // Mira el número de processadors
        System.out.println("Inici càlcul");
        ForkJoinPool pool = new ForkJoinPool();

        int inici=0;
        int  fi= data.length;
        MaximTask tasca = new MaximTask(data, inici, fi);

        long time = System.currentTimeMillis();
        // crida la tasca i espera que es completin
        int result1 = pool.invoke(tasca);
```

```
63          // màxim
64          int result=  tasca.join();
65          System.out.println("Temps utilitzat:" +(System.
                currentTimeMillis()-time));
66
67          System.out.println ("Màxim es " + result);
68      }
69
70
71      private static short [] createArray(int size){
72          short[] ret = new short[size];
73          for(int i=0; i<size; i++){
74              ret[i] = (short) (1000 * Math.random());
75              if(i==((short)(size*0.9))){
76                  ret[i]=1005;
77              }
78          }
79          return ret;
80      }
81  }
```

Proveu diferents llindars per veure que l'actual és el més eficient.

1.4 Sincronització i comunicació entre processos

Comunicar-se i sincronitzar-se són les dues accions més importants en l'execució de processos concurrents. Quan diversos processos estan en execució a la vegada no podem controlar l'ordre en què acabaran executant-se, ja que això dependrà de la política de planificació escollida i dels processos que es trobin en execució a cada moment. Però l'ordre de processament, en algunes ocasions, pot arribar a ser fonamental per aconseguir un resultat correcte. Les tècniques que ens ajuden a controlar l'ordre d'execució dels diferents processos s'anomenen tècniques de comunicació i sincronització.

1.4.1 Competència de recursos i sincronització

Els processos concurrents es poden classificar en independents o cooperants segons el seu grau de col·laboració. Un procés **independent** no necessita ajuda ni cooperació d'altres processos. En canvi, els processos **cooperants** estan dissenyats per treballar conjuntament amb altres processos i, per tant, s'han de poder comunicar i interactuar entre ells.

Aquestes col·laboracions impliquen una coordinació de tots el processos implicats. A vegades la coordinació requereix de cert nivell de comunicació per poder-se sincronitzar. Tot i així, una comunicació detallada és força costosa i sovint n'hi ha prou a assegurar només que els processos no accedeixin a la vegada a un mateix recurs o executin una mateixa operació. En aquest darrer cas direm que els processos competeixen entre si perquè no hi ha manera de saber l'ordre exacte amb què finalment s'executaran els processos.

En la programació concurrent el procés de **sincronització** permet que els processos que s'executen de forma simultània es coordinin, aturant l'execució d'aquells que vagin més avançats fins que es compleixin les condicions òptimes per estar segurs que els resultats finals seran correctes.

Per sincronitzar processos podem fer servir diferents mètodes:

- **Sincronisme condicional**: o condició de sincronització. Un procés o fil es troba en estat d'execució i passa a estat de bloqueig tot esperant que una certa condició es compleixi per continuar la seva execució. Un altre procés fa que aquesta condició es compleixi i així el primer procés passa de nou a l'estat d'execució.

- **Exclusió mútua**: es produeix quan dos o més processos o fils volen accedir a un recurs compartit. S'han d'establir protocols d'execució perquè no accedeixin de forma concurrent al recurs.

- **Comunicació per missatges**: en la qual els processos s'intercanvien missatges per comunicar-se i sincronitzar-se. És la comunicació típica en sistemes distribuïts i és utiltzada també en sistemes no distribuïts.

1.4.2 Sincronització i comunicació

La col·laboració entre processos dóna lloc a un seguit de problemes clàssics de comunicació i sincronització. Sincronitzar i comunicar processos és, doncs, bàsic per tal de solucionar-los.

Seccions crítiques

Les seccions crítiques són un dels problemes que amb més freqüència es donen en programació concurrent. Tenim diversos processos que s'executen de forma concurrent i cadascun d'ells té una part de codi que s'ha d'executar de forma exclusiva ja que accedeix a recursos compartits com ara fitxers, variables comuns, registres de bases de dades, etc.

La solució passarà per obligar a accedir als recursos a través de l'execució d'un codi que anomenarem secció crítica i que ens permetrà protegir aquells recursos amb mecanismes que impedeixin l'execució simultània de dos o més processos dins els límits de la secció crítica.

Aquests algoritmes de sincronització que eviten l'accés a una regió crítica per més d'un fil o procés i que garanteixen que únicament un procés estarà fent ús d'aquest recurs i la resta que volen utilitzar-lo estaran a l'espera que sigui alliberat, s'anomena **algoritme d'exclusió mútua**.

Exclusió mútua (MUTEX, *mutual exclusion* en anglès) és el tipus de sincronització que impedeix que dos processos executin simultàniament una mateixa secció crítica.

Un mecanisme de sincronització en forma de codi que protegeixi la secció crítica haurà de tenir una forma com el següent:

```
Entrada_Secció_Critica /* Sol·licitud per executar Secció Crítica
     */

/* codi Secció Crítica */

Sortida_Secció_Critica /* altre procés pot executar la secció Crí
    tica */
```

L'`Entrada_Secció_Crítica` representa la part del codi en la qual els processos demanen permís per entrar a la secció crítica. La `Sortida_Secció_Critica` en canvi, representa la part que executen els processos quan surten de la secció crítica alliberant la secció i permet a d'altres processos entrar-hi.

Per validar qualsevol mecanisme de sincronització d'una secció crítica s'han de complir els criteris següents:

- Exclusió mútua: no pot haver-hi més d'un procés simultàniament en la secció crítica.
- No inanició: un procés no pot esperar un temps indefinit per entrar a executar la secció crítica.
- No interbloqueig: cap procés de fora de la secció crítica pot impedir que un altre procés entri a la secció crítica.
- Independència del maquinari: inicialment no s'han de fer suposicions respecte al número de processadors o la velocitat dels processos.

Un error de consistència típic quan no hi ha control sobre una secció crítica es pot il·lustrar amb l'exemple de dos processos que volen modificar una variable comuna x. El procés A vol incrementar-la: x++. El procés B disminuir-la: x–;. Si els dos processos accedeixen a llegir el contingut de la variable al mateix temps, els dos obtindran el mateix valor, si fan la seva operació i guarden el resultat, aquest serà inesperat. Dependrà de qui salvi el valor d'x en darrer lloc.

La taula 1.1 mostra un exemple similar. Un codi és accessible per dos fils o processos, veiem que si no hi ha cap tipus de control sobre l'accés, el primer fil accedeix a les instruccions i abans d'arribar a la instrucció d'increment de la variable a++, el segon procés entra a executar el mateix codi. El resultat és que el segon procés pren el valor 4, per tant erroni, ja que el primer procés no hauria augmentat la variable.

TAULA 1.1. Secció crítica

Procés 1	Temps	Procés 2
pout.print(a);	1	
	2	**pout.print(a);**
a=a+4;	3	
	4	**a=a+4;**
pout.print("+4=");	5	
pout.println(a);	6	
	7	**pout.print("+4=");**
	8	**pout.println(a);**

En aquest exemple suposarem què pot representar la sortida a un fitxer, què arriba per paràmetre a la zona crítica i que cada procés treballa amb un fitxer diferent. Suposarem també que la variable *a* té un valor inicial de *4* i que es tracta d'una variable compartida per ambdós processos. Podeu imaginar que les sortides seran força sorprenents, ja que en ambdós fitxers indicaria: *4+4=12*.

Per evitar aquest problema, únicament un fil hauria d'executar aquesta part de codi de forma simultània. Aquesta part de codi, que és susceptible a aquest tipus d'error, s'ha de declarar com a secció crítica per evitar aquest tipus d'error.

Les instruccions que formen part d'una secció crítica s'han d'executar con si fossin una única instrucció. S'han de sincronitzar els processos per tal que un únic procés o fil pugui excloure de forma temporal a la resta de processos d'un recurs compartit (memòria, dispositius, etc.) de tal manera que la integritat del sistema quedi garantida.

Productor-consumidor

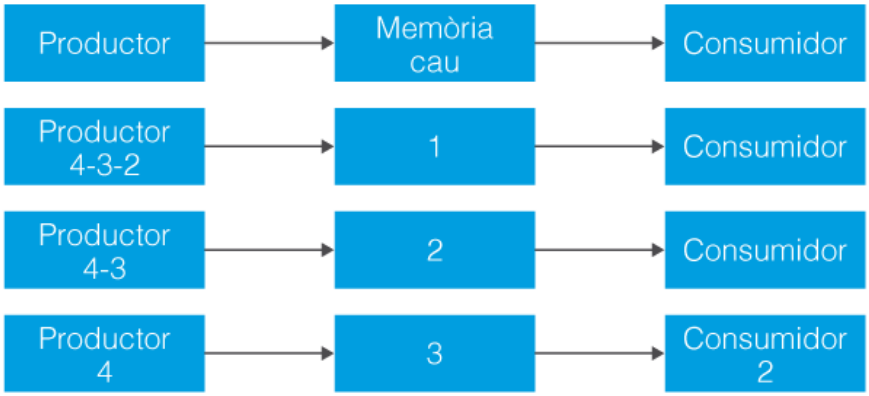
El problema productor-consumidor és un exemple clàssic on cal donar un tractament independent a un conjunt de dades que es van generant de forma més o menys aleatòria o si més no d'una forma en què no és possible predir en quin moment es generarà una dada. Per evitar un ús excessiu dels recursos de l'ordinador esperant l'arribada de dades, els sistema preveu dos tipus de processos: els productors, encarregats d'obtenir les dades a tractar i els consumidors, especialitzats en fer el tractament de les dades obtingudes pels productors.

En aquesta situació el productor genera un seguit de dades que el consumidor recull. Imaginem que és el valor d'una variable que el productor modifica i el consumidor l'agafa per utilitzar-la. El problema ve quan el productor produeix dades a un ritme diferent del que el consumidor les agafa. El productor crea una dada i canvia la variable al seu valor. Si el consumidor va més lent, el productor té temps a generar una nova dada i torna a canviar la variable. Per tant el procés del consumidor ha perdut el valor de la primera dada. En el cas que sigui el consumidor el que va més ràpid, pot passar que agafi dues vegades una mateixa dada, ja que el productor no ha tingut temps de substituir-la, o que no trobi res

per consumir. La situació pot complicar-se encara més si disposem de diversos processos productors i consumidors.

A la figura 1.13 podem veure com dos processos comparteixen un recurs comú (la memòria) i la il·lustració del problema quan no està sincronitzat el procés productor i el consumidor, i el productor és més ràpid que el consumidor.

Figura 1.13. Productor-consumidor



Imaginem que en la figura 1.13 la memòria comú és com una caixa que és capaç de guardar una única dada, un enter. Existeix un procés productor que genera els números enters i els deixa a la caixa. Mentrestant hi ha un procés consumidor que agafa el número enter de la caixa.

Com és el cas, el productor deixa a la memòria el número 1 i abans que el procés consumidor agafi la dada, genera un altre número, el 2, que substitueix l'anterior. El consumidor ara sí que agafa el número 2 però, tal com podem veure, el número 1 s'ha perdut, produint segurament uns resultats erronis.

Una manera de solucionar el problema consisteix a ampliar la ubicació on el productor escriu les dades de manera que sigui possible mantenir diverses dades a l'espera que siguin consumides mentre els processos consumidors estiguin ocupats. És a dir, els productors emmagatzemaran les dades en un llista (*array*), que tradicionalment es coneix com a *buffer* i els consumidors les aniran extraient.

Un ***buffer*** és un espai de memòria per emmagatzemar dades. Es poden implementar en forma de cues.

La figura 1.14 és l'esquema d'un procés productor que deixa informació a un *buffer* i que és agafada per un procés consumidor.

Figura 1.14. Buffer

Malauradament aquest mecanisme no soluciona tots els problemes, ja que pot ser que un consumidor intenti accedir a les dades malgrat el productor no n'hagi encara escrit cap, pot passar que l'espai destinat a emmagatzemar la col·lecció de dades s'ompli a causa que la producció de dades sigui sempre molt més ràpida que els processos consumidors, o bé podria donar-se el cas que dos processos productors coincidissin a l'hora de deixar una dada o que diversos processos consumidors intentessin accedir a l'hora.

Per tant, ha d'existir un mecanisme que aturi l'accés a la dada dels productors i dels consumidors en cas necessari. Una secció crítica. Malauradament no n'hi ha prou a restringir l'accés a les dades, perquè podria donar-se el cas que un procés consumidor esperant l'arribada d'una dada impedeixi l'accés dels processos productors de forma que la dada no arribés mai. Una situació com la descrita es coneix amb el nom de problemes d'interbloqueig (*deadlock* en anglès).

Anomenem ***interbloqueig*** a la situació extrema que ens trobem quan dos o més processos estan esperant l'execució de l'altre per poder continuar de manera que mai aconseguiran desbloquejar-se.

També anomenem ***inanició*** a la situació que es produeix quan un procés no pot continuar la seva execució per falta de recursos. Per exemple si féssim créixer el *buffer* il·limitadament.

Per solucionar el problema cal sincronitzar l'accés al *buffer*. S'ha d'accedir en exclusió mútua per tal que els productors no alimenti el *buffer* si aquest ja està ple i els consumidors no hi puguin accedir si està buit. Però a més caldrà independitzar les seccions crítiques d'accés dels productors de les seccions crítiques d'accés dels consumidors evitant així que es pugui produir l'interbloqueig.

Això però obligarà a crear un mecanisme de comunicació entre seccions crítiques de manera que cada cop que un productor deixi una dada disponible, avisi els consumidors que puguin restar esperant que almenys un d'ells pugui iniciar el processament de la dada.

Lectors-escriptors

Un altre tipus de problema que apareix en la programació concurrent, és el produït quan tenim un recurs compartit entre diversos processos concurrents com poden ser un fitxer, una base de dades, etc. que va actualitzant-se periòdicament. En aquest cas els processos lectors no consumeixen la dada sinó que només la utilitzen i per tant es permet la seva consulta de forma simultània, malgrat que no la seva modificació.

Així, els processos que accedeixin al recurs compartit per llegir el seu contingut s'anomenaran **lectors**. En canvi, els que hi accedeixin per modificar-lo rebran el nom de **escriptors**.

Si la tasca de lectors i escriptors no es realitza de forma coordinada podria passar que un lector llegís diverses vegades la mateixa dada o que l'escriptor modifiques el contingut abans que haguessin llegit tots els lectors, o que la dada actualitzada

per un escriptor malbaratés l'actualització d'un altre, etc. A més, la manca de coordinació obliga als lectors a comprovar periòdicament si els escriptors han fet modificacions, cosa que farà augmentar l'ús del processador i per tant podria minvar l'eficiència.

Aquest problema de sincronització de processos s'anomena **problema de lectors-escriptors**. Per evitar-lo cal assegurar que els processos escriptors accedeixen de forma exclusiva al recurs compartit i que a cada modificació s'avisa als processos lectors interessats en el canvi.

Així, els processos lectors poden restar en espera fins que són avisats que hi ha noves dades i poden començar a llegir; d'aquesta manera s'evita que els lectors estiguin constantment accedint al recurs sense que l'escriptor hagi posat cap nova dada, optimitzant, d'aquesta manera, els recursos.

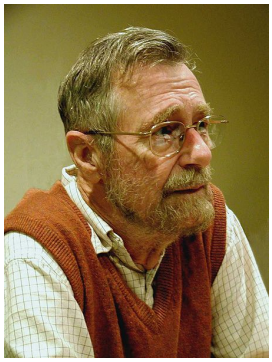
1.4.3 Solucions a problemes de sincronització i comunicació

Al llarg de la història s'han proposat solucions específiques pels problemes anteriors que val la pena tenir en compte, malgrat que resulta difícil generalitzar-ne una solució ja que depenen de la complexitat, la quantitat de seccions crítiques, del nombre de processos que requereixin exclusió mútua i de la interdependència existent entre processos. Aquí veurem la sincronització per mitjà de semàfors, de monitors i de pas de missatges.

Semàfors

Imaginem una carretera d'un sol carril que ha de passar per un túnel. Hi ha un semàfor a cada extrem del túnel que ens indica quan podem passar i quan no. Si el semàfor està en verd el cotxe passarà de forma immediata i el semàfor passarà a vermell fins que surti. Aquest símil ens introdueix a la definició real d'un semàfor.

Els semàfors són una tècnica de sincronització de memòria compartida que impedeix l'entrada del procés a la secció crítica tot bloquejant-lo. El concepte va ser introduït per l'informàtic holandès Dijkstra per resoldre el problema l'exclusió mútua i permetre resoldre gran part dels problemes de sincronització entre processos.



Edsger Wybe Dijkstra, informàtic holandès. La seva idea d'exclusió mútua ideada durant la dècada dels 60 és utilitzada per molts processadors i programadors moderns

Els semàfors, no només controlen l'accés a la secció crítica sinó que a més disposen d'informació complementària per poder decidir si cal o no bloquejar l'accés d'aquells processos que ho sol·licitin. Així per exemple, serviria per solucionar problemes senzills (amb poca interdependència) del tipus escriptors-lectors o productors-consumidors.

La solució per exemple en el cas dels escriptors-lectors passaria per fer que els lectors abans de consultar la secció crítica demanessin permís d'accés al semàfor, el qual en funció de si es troba bloquejat (vermell) o alliberat (verd) aturarà l'execució del procés sol·licitant o bé el deixarà continuar.

Els escriptors per altra banda, abans d'entrar a la secció critica, manipularan el semàfor posant-lo en vermell i no el tornaran a posar en verd fins que hagin acabat d'escriure i abandonin la secció crítica.

De forma genèrica distingirem 3 tipus d'operacions en un semàfor:

El semàfor admet 3 operacions:

- Inicialitza (*initial*): es tracta de l'operació que permet posar en marxa el semàfor. La operació pot rebre un valor per paràmetre que indicarà si aquest començarà boquejat (vermell) o alliberat (verd).
- Allibera(s) (*sendSignal*): canvia el valor intern del semàfor posant-lo en verd (l'allibera). Si existeixen processos en espera, els activa per tal que finalitzin la seva execució.
- Bloqueja(s) (*sendWait*): serveix per indicar que el procés actual vol executar la secció crítica. En cas que el semàfor es trobi bloquejat, s'aturà l'execució del procés. També permet indicar que cal posar el semàfor en vermell.

En cas que es necessiti exclusió mútua, el semàfor disposarà també d'un sistema d'espera (per mitjà de cues de processos) que garanteixi l'accés a la secció crítica d'un únic procés a la vegada.

En realitat la implementació d'un semàfor dependrà molt del problema a resoldre, malgrat que la dinàmica del funcionament sigui sempre molt similar. Així per exemple els semàfor que donin suport al problema de tipus productor-consumidor, necessiten implementar-se fent servir exclusió mútua tant per alliberar com per bloquejar. A més, l'execució d'alliberament incrementarà en una unitat un comptador intern, mentre que l'execució de bloqueig a banda d'aturar el procés quan el semàfor es trobi bloquejat, disminuirà en una unitat el comptador intern.

Tenint en compte que el processos productors executaran sempre l'operació d'alliberament (incrementant el comptador intern) i els processos consumidors demanaran accés executant la operació de bloqueig (que disminuirà el comptador intern), és fàcil veure que el valor del comptador serà sempre igual a la quantitat de dades que els productors ha generat sense que els consumidors hagin encara de consumit. Així podem deduir que si en algun moment el valor arriba al valor zero, significarà que no hi han dades a consumir i per tant farem que el semàfor es trobi sempre bloquejat en aquest cas, però es desbloquegi així que el comptador incrementi el seu valor.

A més de resoldre problemes de tipus productor-consumidor o lector-escritor, podem fer servir semàfors per gestionar problemes de sincronització on un procés hagi d'activar l'execució d'un altre o d'exclusió mútua assegurant que només un procés aconseguirà accedir a la secció crítica perquè el semàfor restarà bloquejat fina a la sortida.

Així si volem sincronitzar dos processos fent que un d'ells (p1) executi una acció sempre abans que l'altra (p2), usant un semàfor, l'inicialitzarem a 0 per assegurar que es troba bloquejat. La codificació del procés p2 assegura que abans de

qualsevol crida a l'acció que volem controlar, sol·licitarà accés al semàfor amb un `sendWait`. Per contra, a la codificació del procés p1 caldrà situar sempre una crida a `sendSignal` just després d'executar l'acció a controlar (veure taula 1.2).

D'aquesta forma ens assegurarem que el procés p1 executarà l'acció sempre abans que p2. Com a exemple, imaginem dos processos. Un ha d'escriure *Hola*, (procés p1) i el procés p2 ha d'escriure *món*. L'ordre correcte d'execució és primer p1 i després p2, per aconseguir escriure *Hola món*. En cas que el procés P1 s'executi abans que p2, cap problema: escriu *Hola* i fa un `signal` al semàfor (semàfor=1). Quan el procés p2 s'executa trobarà semàfor=1, per tant no quedarà bloquejat, podrà fer un `sendWait` al semàfor (semàfor=0) i escriurà *món*.

TAULA 1.2.

Procés 1 (p1)	Procés 2 (p2)
initial(0)	
...	
System.out.print("Hola")	sendWait()
sendSignal()	System.out.print("món")

Però què passa si s'executa primer el procés p2? Com que trobarà semàfor a 0, es quedarà bloquejat en fer la petició cridant *sendWait* fins que el procés p1 escrigui *Hola* i faci el *sendSignal*, desbloquejant el semàfor. Llavors p2, que estava bloquejat, s'activarà, posarà el semàfor de nou a vermell (semàfor=0) i escriurà *món* a la pantalla.

Un altre exemple que pot il·lustrar l'ús de semàfors, seria el d'una oficina bancària que gestiona el nostre compte corrent al qual es pot accedir des de diferents oficines per ingressar diners o treure diners. Aquestes dues operacions modifiquen el nostre saldo. Serien dues funcions com les següents:

```
1  public void ingressar(float diners) {
2          float aux;
3          aux=llegirSaldo();
4          aux=aux+diners;
5          saldo=aux;
6          guardarSaldo(saldo);
7      }
8
9     public void treure(float diners) {
10         float aux;
11         aux=llegirSaldo();
12         aux=aux–diners;
13         saldo=aux;
14         guardarSaldo(saldo);
15     }
```

El problema ve quan de forma simultània es vol fer un ingrés i es vol treure diners. Si per una banda estem traient diners del compte corrent i per una altra banda algú està fent un ingrés, es podria crear una situació anòmala. Hi haurà dos processos concurrents, un traurà diners i l'altre n'ingressarà. Si accedeixen al mateix temps a `llegirSaldo()` els dos agafen el mateix valor, imaginem 100€. El procés que vol ingressar diners, ho vol fer amb la quantitat de 300€. I el que vol treure'n, en vol 30€.

Si continuem l'execució dels dos processos, depenent de quin sigui l'ordre d'execució de les instruccions, ens podem trobar amb un saldo diferent. Si després de llegir el saldo, el procés d'ingrés acaba l'execució i guarda el saldo, guardaria 400 € (100 € + 300 €). Però posteriorment acabaria el procés de treure diners i guardaria a saldo el valor de 70 € (100 € - 30 €). Hem perdut l'ingrés. Hi ha dos processos que s'estan executant a una secció crítica que hauríem de protegir en exclusió mútua. Únicament un procés ha de poder accedir a aquesta secció crítica i poder modificar la variable compartida *saldo*.

Per evitar el problema podem utilitzar un semàfor. L'iniciarem a 1, indicant el número de processos que podran entrar a la secció crítica. I tant en el procés de treure dines com en el d'ingressar-ne afegirem un `sendWait()` a l'inici de les seccions crítiques i un `sendSignal()` al final.

```
1  public void ingressar(float diners) {
2            sendWait();
3            float aux;
4            aux=llegirSaldo();
5            aux=aux+diners;
6            saldo=aux;
7            guardarSaldo(saldo);
8            sendSignal();
9    }
10
11     public void treure(float diners) {
12           sendWait();
13           float aux;
14           aux=llegirSaldo();
15           aux=aux-diners;
16           saldo=aux;
17           guardarSaldo(saldo);
18           sendSignal();
19   }
```

D'aquesta forma quan un procés entra a la secció crítica d'un mètode, agafa el semàfor. Si és 1, podrà fer el `sendWait`, per tant el semàfor es posarà a 0, tancat. I cap altre procés no podrà entrar a cap dels dos mètodes. Si un procés intenta entrar, trobarà el semàfor a 0 i quedarà bloquejat fins que el procés que té el semàfor faci un `sendSignal`, posi el semàfor a 1 i alliberi el semàfor.

Utilitzar semàfors és una forma eficient de sincronitzar processos concurrents. Resol de forma simple l'exclusió mútua. Però des del punt de vista de la programació, els algoritmes són complicats de dissenyar i d'entendre, ja que les operacions de sincronització poden estar disperses pel codi. Per tant es poden cometre errors amb facilitat.

Monitors

Una altra forma de resoldre la sincronització de processos és l'ús de monitors. Els monitors són un conjunt de procediments encapsulats que ens proporcionen l'accés a recursos compartits a través de diversos processos en exclusió mútua.

Les operacions del monitor estan encapsulades dins d'un mòdul per protegir-les del programador. Únicament un procés pot estar en execució dins d'aquest mòdul.

El grau de seguretat és alt ja que els processos no saben com estan implementats aquests mòduls. El programador no sap com i en quin moment es criden les operacions del mòdul, així és més robust. Un monitor, un cop implementat, si funciona correctament sempre funcionarà bé.

Un monitor es pot veure com una habitació, tancada amb una porta, que té a dins els recursos. Els processos que vulguin utilitzar aquests recursos han d'entrar a l'habitació, però amb les condicions que marca el monitor i únicament un procés a la vegada. La resta que vulgui fer ús dels recursos haurà d'espera que surti el que és dins.

Per la seva encapsulació, l'única acció que ha de prendre el programador del procés que vulgui accedir al recurs protegit és informar al monitor. L'exclusió mútua hi és implícita. Els semàfors, en canvi, s'han d'implementar amb una seqüència correcta de senyal i esperar per tal de no bloquejar el sistema.

Un **monitor** és un algoritme que fa una abstracció de dades que ens permet representar de forma abstracta un recurs compartit mitjançant un conjunt de variables que defineixen el seu estat. L'accés a aquestes variables únicament és possible des d'uns mètodes del monitor.

Els monitor ha de poder incorporar un mecanisme de sincronització. Per tant, s'han d'implementar. Es pot fer ús de senyals. Aquests senyals s'utilitzen per impedir els bloquejos. Si el procés que és en el monitor ha d'esperar un senyal, es posa en estat d'espera o bloquejat fora del monitor, permetent que un altre procés faci ús del monitor. Els processos que són fora del monitor, estan a l'espera d'una condició o senyal per tornar a entrar.

Aquestes variables que s'utilitzen pels senyals i són utilitzades pel monitor per la sincronització s'anomenen **variables de condició**. Aquestes poden ser manipulades amb operacions de `sendSignal` i `sendWait` (com els semàfors).

- *sendWait*: un procés que està esperant a un esdeveniment indicat per una variable de condició abandona de forma temporal el monitor i es posa a la cua que correspon a la seva variable de condició.
- *sendSignal*: desbloqueja un procés de la cua de processos bloquejats amb la variable de condició indicada i es posa en estat preparat per entrar al monitor. El procés que entra no ha de ser el que més temps porta esperant, però s'ha de garantir que el temps d'espera d'un procés sigui limitat. Si no existeix cap procés a la cua, l'operació *sendSignal* no té efecte, i el primer procés que demani l'ús del monitor entrarà.

Un monitor consta de 4 elements:

- Variables o mètodes permanents o privats: són les variables i mètodes interns al monitor que només són accessibles des de dins del monitor. No es modifiquen entre dues crides consecutives al monitor.

- Codi d'inicialització: inicialitza les variables permanents, s'executa quan el monitor és creat.
- Mètodes externs o exportats: són mètodes que són accessibles des de fora del monitor pels processos que volen entrar a fer-ne ús.
- Cua de processos: és la cua de processos bloquejats a l'espera del senyal que els alliberi per tornar a entrar al monitor.

wait, *notify* i *notifyAll*, són operacions que permeten sincronitzar el monitor.

En el camp de la programació un monitor és un objecte en el qual tots els seus mètodes estan implementats sota exclusió mútua. En el llenguatge Java són objectes d'una classe en els quals tots els seus mètodes públics són *synchronized*.

Un semàfor és un objecte que permet sincronitzar l'accés a un recurs compartit i un monitor és una interfície d'accés al recurs compartit. Són l'encapsulament d'un objecte, per allò que fa un objecte més segur, robust i escalable.

Sincronitzar a Java

A Java l'execució d'objectes no està protegida. Si volem aplicar exclusió mútua a Java hem d'utilitzar la paraula reservada *synchronized* a la declaració d'atributs, de mètodes o en un bloc de codi dins d'un mètode.

Tots els mètodes amb el modificador *synchronized* s'executaran en exclusió mútua. El modificador assegura que si s'està executant un mètode sincronitzat cap altre mètode sincronitzat es pugui executar. Però els mètodes no sincronitzats poden estar executant-se i amb més d'un procés a la vegada. A més, únicament el mètode sincronitzat pot estar executat per un procés, la resta de processos que volen executar el mètode s'hauran d'esperar que acabi el procés que l'està executant.

Sincronitzar mètodes a Java

```
1   public synchronized  void metodeSincronitzat {
2       //part de codi sincronitzat
3   }
```

Si no ens interessa sincronitzar tot el mètode, sinó únicament una part del codi, utilitzarem la paraula reservada *synchronized* sobre el mateix objecte que ha cridat el mètode en el qual es troba la part del codi que s'ha de sincronitzar.

Sincronitzar blocs de codi a Java

```
1  public  void metodeNoSincronitzat {
2      //part de codi sense sincronitzar
3      synchronized(this){
4        //part de codi sincronitzat
5      }
6
7      //part de codi sense sincronitzar
8  }
```

`this` és l'objecte que ha cridat al mètode, però també podem utilitzar un altre objecte `synchroniced(objecte)`, si el que volem és realitzar més d'una operació atòmica sobre un objecte.

Java, per tant, ens proporciona amb *synchronized* els recursos que ens donaria la implementació d'un semàfor o d'un monitor. Les seves llibreries ens proporcionen un accés en exclusió mútua i controlen els errors de bloqueig o interbloqueig que es pugui ocasionar entre fils en execució.

Atomicitat d'una operació i classes atòmiques de Java

L'únic àrbitre que tenim en la gestió de processos és el planificador. S'encarrega de decidir quin procés fa ús del processador. Després les instruccions o operacions que formen el procés es van executant al processador seguint un ordre concret.

Una operació o un grup d'operacions s'han de poder executar com si fossin una única instrucció. A més, no es poden executar de forma concurrent amb qualsevol altra operació en la qual hi hagi involucrada una dada o recurs que utilitza la primera operació.

L'atomicitat d'una operació és poder garantir que no es poden executar dues operacions de forma concurrent si fan ús d'un recurs compartit fins que una de les dues deixa lliure aquest recurs. Una operació atòmica únicament ha d'observar dos estats: l'inicial i el resultat. Una operació atòmica, o s'executa completament o no ho fa en absolut.

L'**atomicitat real** són les instruccions que executen a codi màquina. En canvi, l'**atomicitat model** és un grup d'instrucció, a codi màquina, que s'executa de forma atòmica.

Classes d'operacions atòmiques

La instrucció `x++;` o `x= x + 1;` té la següent seqüència d'instruccions atòmiques reals:

1. Llegir valor x
2. Afegir a x1
3. Guardar valor x

L'atomicitat de model entén la instrucció completa, `x = x + 1` com a atòmica.

Dos processos executen en paral·lel el següent exemple:

```
1  package operacionsatomiques;
2  public class OperacionsAtomiques {
3    public static void main(String[] args) {
4      int x=0;
5      x=x+1;
6      System.out.println(x);
7    }
8  }
```

Si tenim en compte que els dos processos executen en paral·lel la instrucció `x=x+1;` en l'atomicitat real una ordenació d'execució seria la següent (taula 1.3):

Taula 1.3.

1		int x=0;
	Procés 1 (x=x+1)	**Procés 2(x=x+1)**

2	Llegir valor x	
3		Llegir valor x
4	Afegir a x 1	
5		Afegir a x 1
6	Guardar valor x	
7		Guardar valor x
8	System.out.printl(x);	

Depenent de l'ordre d'execució de les operacions atòmiques d'un procés i l'altre el resultat o el valor d'x pot variar.

Classes atòmiques de Java

Les classes atòmiques de Java són una primera solució als problemes de processos concurrents. Permeten executar algunes operacions com si fossin atòmiques reals.

Una operació atòmica té únicament dos estats: l'inicial i el resultat, és a dir, es completa sense interrupcions des de l'inici fins al final.

Volatile és un modificador, utilitzat en la programació concurrent, que es pot aplicar a alguns tipus primitius de variables. Indica que el valor de la variable quedarà guardat a la memòria i no a un registre del processador. Així, és accessible per un altre procés o fil per modificar-la. Per exemple:
`volatile int c;`

A Java, l'accés a les variables de tipus primitius (excepte *double* i *long*) és atòmic. Com que també és atòmic l'accés a totes les variables de tipus primitiu en les quals apliquem el modificador *volatile*.

Java assegura l'atomicitat de l'accés a les variables **volàtils** o primitives, però no a la seves operacions. Això vol dir que l'operació `x++;` en la qual `x` és un `int`, no és una operació atòmica i per tant no és un fil segur.

Java incorpora, des de la versió 1.5, al paquet java.util.concurrent.atomic, les classes que ens permeten convertir en atòmiques (fils segurs) algunes operacions, com ara augmentar, reduir, actualitzar i afegir un valor. Totes les classes tenen mètodes *get* i *set* que també són atòmiques.

Les classes que incorpora són: `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray` i actualitzadors que són bàsicament derivats d'una variable tipus *volatile* i són `AtomicIntegerFieldUpdater`, `AtomicLongFieldUpdater` i `AtomicReferenceFieldUpdater`.

Per tant, podem convertir l'operació `x++` o `x=x+1` en una operació atòmica .

Classe Comptador sense operacions atòmiques

```
1  public class Comptador {
2      private int x = 0;
3      public void augmenta() {
4          x++;
5      }
6      public void disminueix() {
7          x--;
8      }
9      public int getValorComptador() {
```

```
10          return x;
11      }
12    }
```

Per convertir a operacions atòmiques l'exemple anterior, utilitzarem els mètodes de la classe `AtomicInteger` del paquet java.util.concurrent.atomic.AtomicInteger.

Classe contador utilitzant classes atòmiques

```
1
2  import java.util.concurrent.atomic.AtomicInteger;
3      public class ContadorAtomic {
4        private AtomicInteger x =new AtomicInteger(0);
5        public void augmenta() {
6          x.incrementAndGet();
7        }
8        public void disminueix() {
9          x.decrementAndGet();
10       }
11       public int getValorContadorAtomic() {
12         return x.get();
13       }
14    }
```

A la documentació oficial podeu trobar més informació sobre aquest paquet. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html

Col·leccions concurrents

Les col·leccions són objectes que contenen múltiples dades d'un mateix tipus. Les col·leccions poden estar involucrades també en l'execució de processos concurrents i ser utilitzades per diferents processos de manera que s'acabin compartint també les dades contingudes a la col·lecció.

A Java per treballar amb col·leccions hem d'utilitzar el Framework Collections en el qual es troben els paquets java.util i java.util.concurrent, que contenen les classes i interfícies que ens permeten crear col·leccions.

A Java les interfície List<E>, Set<E> i Queue<E> serien les col·leccions més importants. Representen llistes, conjunts i cues. Map<E> és també una col·lecció de *key*/valor. Cadascuna amb les seves característiques d'ordenació, els mètodes d'inserció, d'extracció i consulta, si permeten elements repetits o no, etc.

El problema amb aquestes col·leccions és que en la programació concurrent en la qual siguin compartides per diferents processos o fils de forma simultània, podrien donar resultats inesperats.

Alguns llenguatges de programació van solucionar aquest problema sincronitzant les classes. A Java s'utilitza, per exemple, Collections.synchronizedList(List<T> list) sobre la interfície List<E> per poder sincronitzar la classe que la implementa, la classe `ArrayList`. Altres classes que implementen la interfície List és `Vector`. Java ja sincronitza aquesta classe.

La classe `CopyOnWriteArrayList` cada cop que es modifica l'*array* es crea en la memòria un *array* amb el contingut sense processos de sincronització. Això fa que les consultes no siguin tan costoses.

La cua FIFO (en anglès, *First In Firts Out*) és una estructura de dades implementada de forma que els primers elements en entrar a l'estructura són els primers en sortir-ne.

En la versió 1.5 va aparèixer java.util.concurrent que incorpora classes que ens permeten solucionar problemes de concurrència d'una forma simple.

`ArrayList` és una col·lecció de dades i si volem que sigui utilitzada per diferents fils d'execució el que hem de fer és sincronitzar totes les operacions de lectura i escriptura per preservar la integritat de les dades. Si aquest `ArrayList` és molt consultat i poc actualitzat, la sincronització penalitza molt l'accés. Java crea la classe `CopyOnWriteArrayList`, que és l'execució de fil segur d'una `ArrayList`.

La interfície **BlockingQueue** implementa una cua FIFO, que bloqueja el fil d'execució si intenta treure de la cua un element i la cua està buida o si intenta inserir un element i està plena. També conté mètodes que fan que el bloqueig duri un temps determinat abans de donar un error d'inserció o lectura i així evitar una execució infinita.

La classe `SynchronousQueue` és una cua de bloqueig, implementa en BlockingQueue. El que fa és que per cada operació d'inserció ha d'esperar una d'eliminació i per cada inserció ha d'espera una eliminació.

`ConcurrentMap` és una altra classe que defineix una taula *hash* amb operacions atòmiques.

Podeu trobar més informació sobre el paquet a la plana oficial de Java. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

Comunicació amb missatges

Una solució als problemes de concurrència de processos, molt adequada sobretot pels sistemes distribuïts, són els missatges. Recordem que un sistema distribuït no comparteix memòria, per tant no és possible la comunicació entre processos amb variables compartides.

Aquest sistema de sincronització i comunicació està inclòs en la majoria de sistemes operatius i es fa servir tant per la comunicació entre diferents ordinadors, com per comunicar entre processos d'una mateix ordinador. La comunicació a través de missatges sempre necessita un procés emissor i un altre de receptor per poder intercanviar informació. Per tant, les operacions bàsiques seran enviar(missatge) i rebre(missatge).

Donat que en aquesta comunicació poden estar implicats diversos processos, cal permetre a les operacions bàsiques identificar qui és el destinatari i fins i tot convindria en ocasions identificar l'emissor. Caldrà estendre les operacions bàsiques incorporant a qui va dirigit el missatge o com el receptor podria indicar de qui espera el missatge.

Segons la forma de referenciar el destinatari i l'emissor, parlarem d'enviament de missatges en **comunicació directa** o **comunicació indirecta**.

En la **comunicació directa**, l'emissor identifica el receptor explícitament quan envia un missatge i viceversa, el receptor identifica explícitament l'emissor que li envia el missatge. Es crea un enllaç de comunicació entre el procés emissor i

procés receptor. Les ordres d'enviament i recepció serien:

enviar(A,missatge)->Enviar un missatge al procés A

rebre(B,misstage)->Rebre un missatge del procés B

Aquest sistema de comunicació ofereix una gran seguretat ja que identifica els processos que actuen en la comunicació i no introdueix cap retard a l'hora d'identificar els processos. Per altra banda això també és un desavantatge, ja que qualsevol canvi en la identificació de processos implica un canvi en la codificació de les ordres de comunicació.

La comunicació directa només és possible d'implementar si podem identificar els processos. És a dir caldrà que el sistema operatiu i el llenguatge de programació suportin la gestió de processos. També cal que els processos s'executin en un mateix ordinador, ja que sinó és impossible accedir-hi de forma directa.

Quan els processos s'executin en diferents ordinadors, serà necessari interposar entre ells un sistema de missatgeria que enviï i reculli els missatges amb independència de quin sigui el procés emissor i receptor. Només caldrà que ambdós processos tinguin accés al mateix sistema d'intercanvi de missatges i que aquest permeti identificar d'alguna manera l'emissor i el receptor. Cal adonar-se que es tracta identificació interna del sistema de missatgeria. En cap cas són referencies directes als processos.

En la **comunicació indirecta** es desconeixen les referències als processos emissor i receptor del missatge. És necessària l'existència d'un sistema de missatgeria capaç de distribuir missatges identificats amb algun valor específic i únic, i permetre als processos la consulta dels missatges a partir d'aquest identificador per destriar els vagin dirigits a ells. En certa forma és com vehicular la comunicació a través d'una bústia comuna (l'identificador). L'emissor envia el missatge a una bústia específica i el receptor recull el missatge de la mateixa bústia.

Les ordres serien de la següent forma:

enviar(BústiaA,missatge)→ El procés emissor deixa el missatge a la bústia A

rebre(BústiaA,missatge)→ El procés receptor recull el missatge de la bústia A

Aquest sistema és més flexible que la comunicació directa ja que ens permet tenir més d'un enllaç de comunicacions entre els dos processos i podem crear enllaços de tipus *un a molts*, *molts a un* i *molts a molts*. En els enllaços de comunicació *un a molts* com ara els sistemes client/servidor les bústies s'anomenen **ports**. Aquests múltiples enllaços no es podrien fer amb el sistema de comunicació directa. El sistema operatiu és l'encarregat de realitzar l'associació de les bústies als processos i ho pot fer de forma dinàmica o estàtica.

D'altra banda, si tenim en compte la immediatesa en la recepció dels missatges entre processos parlarem de: comunicació **síncrona** o **asíncrona**.

En la **comunicació síncrona** el procés que envia un missatge a un altre procés es queda a l'espera de rebre resposta. Per tant, l'emissor queda bloquejat fins que rep aquesta resposta. I viceversa. En canvi en la **comunicació asíncrona** el procés que

envia el missatge continua la seva execució sense preocupar-se de si el missatge ha estat rebut o no. Per tant aquí haurà d'existir una memòria intermèdia o que es vagin acumulant els missatges. El problema vindrà quan la memòria intermèdia es trobi plena. El procés que envia el missatge queda bloquejat fins que troba l'espai dins la memòria intermèdia.

Un símil a aquests dos tipus de sincronització podria ser l'enviament de missatges a través de correu electrònic o de xat. El primer, que va deixant els missatges a una bústia sense preocupar-se si els receptor el rep o no, seria una comunicació asíncrona. En canvi en un xat la comunicació és totalment síncrona ja que l'emissor i el receptor han de coincidir en el temps.

Implementació usant Java

La comunicació i sincronització entre processos d'ordinadors diferents connectats en xarxa (sistemes distribuïts) requereixen d'una mecanisme de transferència que assumeixi la comunicació indirecta, els sòcols o *sockets*. Es tracta d'unes biblioteques que disposen de tots els mecanismes necessaris per enviar i rebre missatges a través de la xarxa.

Dins del mateix ordinador, Java disposa de mètodes com `wait()`, `notify()` i `notifyAll()`, que modifiquen l'estat d'un fil aturant o activant l'execució dels processos referenciats. Aquests mètodes han de ser invocats sempre dins d'un bloc sincronitzat (amb el modificador `synchronize`).

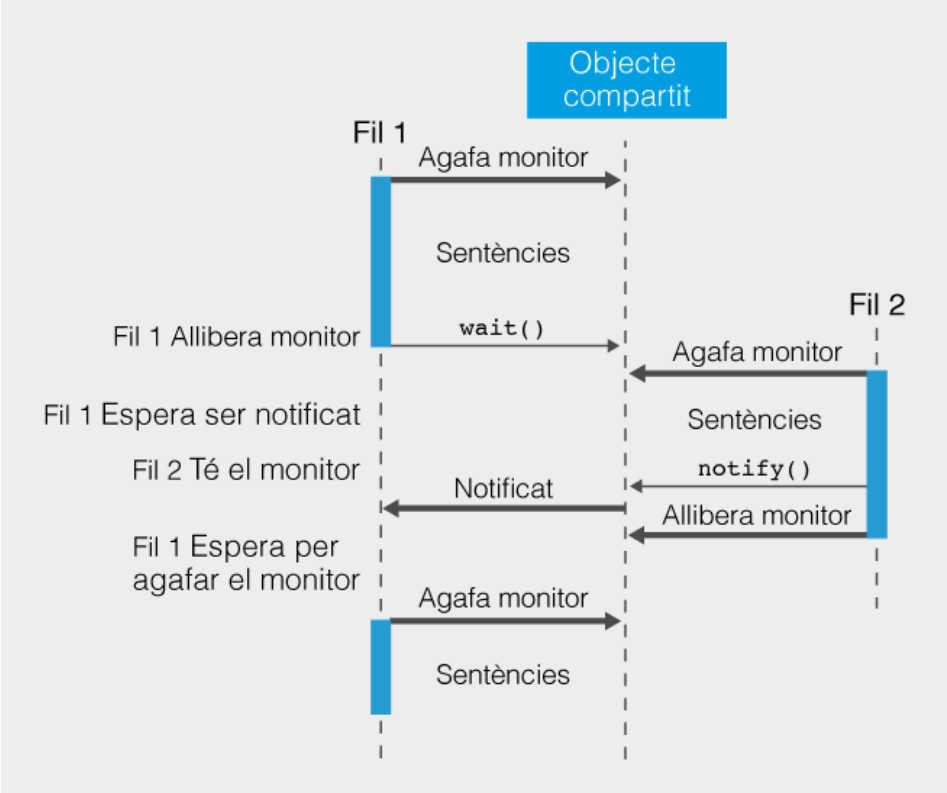
`wait()`: posa el fil que s'està executant en estat d'espera. Aquest fil abandona la zona d'exclusió mútua i espera, en una cua d'espera, a ser tornat a a activar per un mètode `notify()` o `notifyAll()`.

`notify()`: un fil de la cua d'espera passa a l'estat de preparat.

`notifyAll()`: tots els fils de la cua d'espera passen a l'estat de preparat.

Quan un procés entra en la secció crítica, és a dir que controla el monitor, cap altre fil pot entrar a la secció crítica del mateix objecte que està controlat pel mateix monitor. Per poder coordinar-se i comunicar-se es faran ús del mètodes anteriors. Quan no es donen les condicions per continuar amb l'execució d'un fil que es troba a la secció crítica, aquest pot abandonar la secció i permetre que un altre fil que es troba en espera pugui agafar el monitor i entrar a la secció crítica. En la següent figura (figura 1.15) es mostra com dos fils volen accedir a un recurs comú i es comuniquen per sincronitzar-se. Imaginem que el recurs compartit és una targeta SIM d'un telèfon mòbil. El fil 1 vol accedir a la targeta per fer una trucada, agafa el monitor del recurs compartit per tenir accés exclusiu a la targeta, però la SIM no està activada encara. La condició per poder executar-se no es compleix. Per tant, executa el mètode `wait()` que el manté en espera fins que la condició es compleixi. Qualsevol altre procés que vulgui fer la trucada i agafi el monitor farà el mateix, executarà el mètode `wait()` i es quedarà a la cua d'espera.

Figura 1.15. wait() i notify()



El fil 2 és un procés que s'encarrega d'activar la targeta SIM. Quan agafa el monitor té accés amb exclusió mútua a la targeta. Fa les sentències per activar la SIM i executa `notify()`. Aquesta sentència notifica al primer fil, que està a la cua de processos en espera, que ja es compleixen les condicions per poder continuar l'execució dels processos de trucades i allibera el monitor. La diferència entre `notify()` i `notifyAll()` és que la notificació es fa a un fil en espera o a tots. Per últim, el fil 1 agafa el monitor i executa la seves sentències de trucada.

A Java, fent ús de ***synchronized*** es creen semàfors. Si hi afegim els mètodes `wait()`, `notify()` o `notifyAll()` crearem monitors fàcilment i de forma eficient.

2. Programació multifil

La programació multifil permet dur a terme diferents fils d'execució a la vegada, és a dir, permet realitzar diferents tasques en una aplicació de forma concurrent. La majoria de llenguatges de programació permeten treballar amb fils i realitzar programació multifil. També són multifil la majoria de les aplicacions que fem servir als nostres ordinadors (editors de text, navegadors, editors gràfics...), fet que ens dóna la sensació de més agilitat d'execució.

En un editor de text, per exemple, un fil pot estar controlant l'ortografia del text, un altre pot estar atent a les pulsacions sobre les icones de la interfície gràfica i l'altre guardant el document.

2.1 Els fils

La programació tradicional està dissenyada per treballar de forma seqüencial, de manera que quan acaba un procés se'n posa en marxa un altre. Els fils són una forma d'executar paral·lelament diferents parts d'un mateix programa. A Java els fils són coneguts com a *threads*.

2.1.1 Definició de fil

> Un fil o *thread* és la unitat de processament més petita que pot ser planificada per un sistema operatiu.

El fils ens permeten executar concurrentment diferents tasques amb una despesa mínima de recursos ja que tots els fils d'un mateix procés comparteixen l'espai de memòria i la seva creació consumeix poc temps de processador. Són **entitats lleugeres**.

De la mateixa manera que un sistema operatiu permet executar diferents processos a la vegada per concurrència o paral·lelisme, dins d'un procés hi haurà un o diversos fils en execució.

Els fils representen exclusivament les execucions de les instruccions d'un programa que es duen a terme simultàniament en el context d'un mateix procés. És a dir, compartint l'accés a la mateixa zona de memòria assignada al procés al que pertanyen. Cada procés conté com a mínim un fil, malgrat que en pot arribar a contenir molts.

2.1.2 Relació procés i fil

Recordeu que els processos no comparteixen memòria entre ells, són independents, porten informació sobre el seu estat i interactuen amb altres processos a través de mecanismes de comunicació donats pel sistema. Aquesta comunicació o el canvi d'estat d'un procés són mecanismes costosos pel microprocessador. És per això que s'anomenen entitats pesades. Els fils, en canvi, comparteixen recursos. El canvi d'estat i d'execució es realitza molt més ràpidament i poden comunicar-se entre ells fent servir les dades de la memòria que comparteixen. El temps d'ús del microprocessador dedicat a la gestió de fils és força menyspreable.

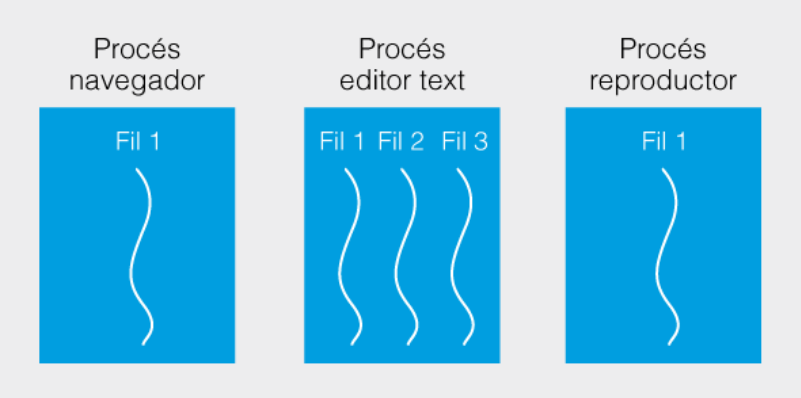
El fet que els fils d'execució d'un mateix procés comparteixin els recursos fa que qualsevol fil pugui modificar-los. Si un fil modifica una dada en la memòria, la resta dels fils tenen accés a la nova dada de forma immediata. Quan hi ha un procés executant-se, com a mínim sempre hi ha un fil en execució. Parlem de processos multifil quan s'executen diversos fils concurrentment, realitzant diferents tasques i col·laborant entre ells.

Quan un procés finalitza, tots els seus fils també ho fan. De forma equivalent, quan finalitzen tots els fils d'un procés, el procés també acaba i tots els seus recursos són alliberats.

Els sistemes operatius actuals admeten la concurrència, és a dir, poden executar diferents processos a la vegada. Podem estar navegant per Internet amb un navegador, escoltant música amb un reproductor i, amb un editor de text, redactant un document. Cada aplicació que s'executa és un procés i cada un d'aquests processos té com a mínim un fil d'execució, però en podrien tenir més d'un. L'editor de text podria tenir un fil que s'encarregués de veure quina tecla estem picant, un altre que vagi comprovant l'ortografia i un altre que de tant en tant s'activi per guardar el document. Cal veure que un fil pertany a un procés i que els diferents fils poden fer tasques diferents dins del seu procés.

A la figura 2.1 veiem un símil dels tres processos anteriors amb els seus fils d'execució.

FIGURA 2.1. Diferents processos amb fils

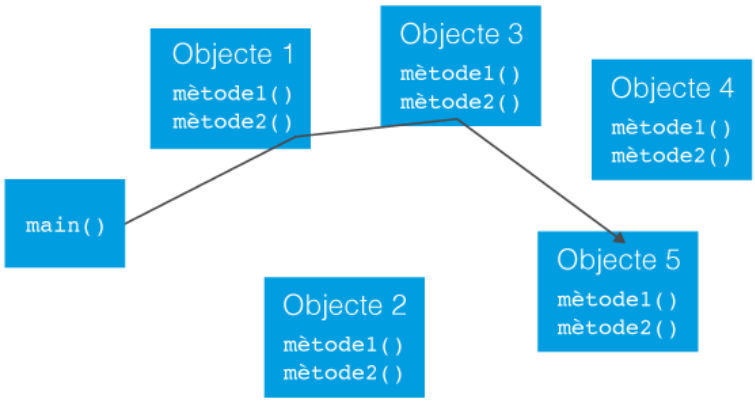


2.1.3 El fils a Java

Quan comencem un programa en Java hi ha un fil en execució, el **fil principal**, que és creat pel mètode `main()`. Aquest fil és important ja que és l'encarregat, si cal, de crear la resta de fils del programa. S'ha de programar l'aplicació perquè el fil principal sigui l'últim en acabar la seva execució. Això s'aconsegueix fent esperar els fils creats pel fil principal que aquest últim finalitzi.

Si no es creen més fils que el principal, només n'existeix un i serà l'encarregat de fer les crides als mètodes i crear el objectes que indiqui el programa. A la figura 2.2 podem veure com el fil principal va executant seqüencialment els mètodes dels diferents objectes que el programa li va indicant.

FIGURA 2.2. Un únic fil d'execució

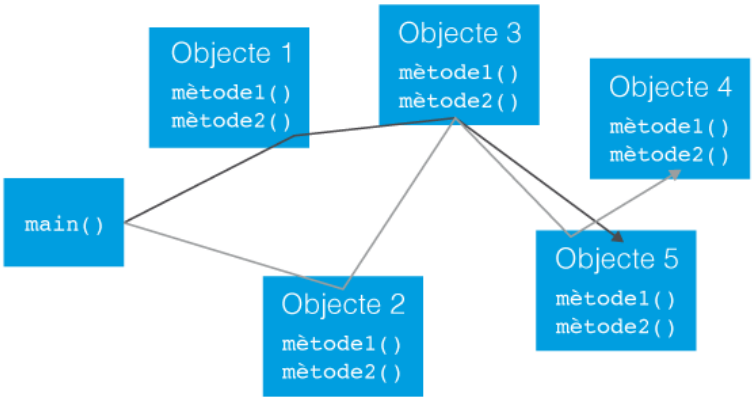



```
1  package filexecuciounic;
2  public class FilExecucioUnic {
3
4      public static void main(String[] args) {
5         for (int x=0;x<10;x++)
6              System.out.println("x=" + x);
7      }
8  }
```

En l'exemple anterior, quan s'executa el `main()`, s'imprimeixen els diferents valors d' `x`. Això passa en un únic fil d'execució.

Però des del fil principal es poden crear altres fils que aniran executant altres mètodes dels objectes instanciats o mètodes nous que calgui crear, de forma simultània. A la figura 2.3 veiem com, des del fil principal, es crea un altre fil que va seguint un ordre d'execució diferent de l'execució del principal i com, en un moment donat, executen de forma simultània el mateix mètode del mateix objecte.

FIGURA 2.3. Dos fils d'execució



Un fil creat pel `main()` pot crear altres fils, que a la vegada poden crear nous fils. Els fils creats depenen directament dels seus creadors, d'aquesta manera es creen jerarquies de pares i fills de fils d'execució.

Java defineix la funcionalitat principal de la gestió dels fils a la classe `Thread`. Gràcies a la classe `Thread` podem crear i executar nous fils, aturar-los, reprendre'n l'execució, esperar la finalització dels fils creats, etc. La funcionalitat de `Thread` es complementa amb la funcionalitat específica de la classe `object` destinada a resoldre la sincronització entre fils. La sincronització s'encapsula dins la classe `Object` per tal que qualsevol execució disposi sempre dels mètodes i mecanismes necessaris per dur-la a terme.

L'herència ens permetrà, de forma fàcil, incorporar la funcionalitat de la classe `Thread` a les nostres implementacions. Tot i així, a causa que Java no suporta l'herència múltiple, el llenguatge Java ens ofereix una forma alternativa d'incorporar la funcionalitat de `Thread` a les nostres implementacions fent servir la interfície `Runnable`. Veiem l'especificació dels principals constructors i mètodes d'aquesta classe.

La classe Thread

Constructors:

- `Thread()`: reserva memòria per crear un objecte `Thread`.
- `Thread(Runnable target)`: reserva memòria per la creació d'un nou objecte de la classe `Thread` el qual gestionarà l'execució de l'objecte Runnable passat per paràmetre.
- `Thread(Runnable target, String name)`: reserva memòria per la cre-

ació d'un nou objecte de la classe `Thread` el qual gestionarà l'execució de l'objecte Runnable passat per paràmetre. L'objecte creat es podrà identificar amb el nom passat en el segon paràmetre.

- `Thread(String name)`: reserva memòria per crear un objecte `Thread` identificat amb el nom passat per paràmetre.
- `Thread(ThreadGroup group, Runnable target)`: reserva memòria per un nou objecte `Thread` que gestionarà l'execució de l'objecte Runnable (*target*) i que pertany al grup de fils (*group*).
- `Thread(ThreadGroup group, Runnable target, String name)`: reserva memòria per un nou objecte `Thread` identificat amb el nom *name*, que pertany al grup de fils (*group*) i que gestionarà l'execució de l'objecte `Runnable` *target*.
- `Thread(ThreadGroup group, String name)`: crea un nou objecte `Thread` identificat per *name* i que pertany al grup de fils concret (*group*) .

Mètodes més importants:

- `static Thread currentThread()`: ens retorna el fil que s'està executant.
- `string getName()`: retorna el nom del fil.
- `void setName(String nom)`: assigna un nom al fil.
- `int getPriority()`: ens diu la prioritat d'execució del fil.
- `void setPriority(int Prioritat)`: assigna la prioritat d'execució del fil.
- `ThreadGroup getThreadGroup()`: retorna el grup de fils al qual pertany el fil.
- `boolean isAlive()`: mira si el fil està actiu. És a dir, si hi ha hagut una crida inicial al seu mètode `start()` i encara no ha finalitzat l'execució.
- `boolean isDaemon()`: ens indica si el fil és un dimoni o no.
- `void setDaemon(boolean on)`: indica si el fil serà un dimoni o un fil d'usuari. Els dimonis estan lligats al seu creador, si el seu creador acaba la seva execució, els *fills* dimonis també finalitzen.
- `void join()`: Atura l'execució del fil que fa la crida i espera la finalització del fil invocat.
- `void join(long Millis)`: espera la finalització del fil invocat, però un cop passat el temps del paràmetre (expressat en mil·lisegons), es continuarà l'execució.
- `void run()`: Representa l'inici d'execució del fil. Aquesta execució comença immediatament després d'haver cridat el mètode `star()`.

- `static void sleep(long Millis)`: atura el fil invocat, durant el nombre de mil·lisegons indicats al paràmetre.
- `void start()`: posa un fil en estat de preparat per executar-se.
- `static void yield()`: fa que el fil que s'està executant passi a estat de preparat. Surt del processador.
- `static void sleep(long Millis)`: fa que un fil que s'està executant s'adormi durant un temps, que s'estableix en mil·lisegons. El fil no perd cap propietat de bloqueig.
- `String toString()`: retorna una cadena que representa el fil, nom prioritat i el seu grup.

Podeu trobar més informació sobre mètodes, propietats i constructors de la classe `Thread` a:

http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.html

La interfície Runnable

En el llenguatge Java, haver d'heretar de la classe Thread per crear classes amb aquesta funcionalitat comporta un problema no pas menor, ja que Java no suporta l'herència múltiple i per tant ens resultarà impossible fer que la nova classe n'estengui cap altre que no sigui Thread. És a dir, ens quedarà molt limitada la implementació de classes fil, usant herència. Per evitar aquesta limitació podem fer servir la Interfície Runnable. La interfície Runnable força l'existència del mètode *run* però no limita l'herència. De fet els objectes `Runnable` no tenen la capacitat d'executar-se com un fil independent. Per això cal fer servir un objecte `Thread` per gestionar la seva execució.

La classe Object

La classe `Object` és molt extensa. Aquí únicament ens centrarem en els mètodes que ens proporciona, relacionats amb els fils. Són mètodes que ens permeten la comunicació i manipulació de fils. Aquesta classe, jeràrquicament, es troba a la part superior de totes les classes de Java, per tant cada classe de Java hereta la seva funcionalitat. Cada nova classe, doncs, inclou els mètodes següents:

- `wait()`: aquest mètode fa que un fil d'execució passi a un estat d'espera fins que se li notifiqui que pot continuar l'execució.
- `notify()`: és el mètode encarregat de notificar a un fil que es troba en espera que pot continuar l'execució.
- `notifyAll()`: funciona de forma similar a `notify()`, però notifica que poden continuar l'execució tots els fils en espera.

Aquests mètodes són utilitzats quan executem la programació concurrent de diferents fils (programació multifil), i serveixen per fer esperar un fil que un altre acabi de fer alguna tasca. També que, un cop acabada aquesta tasca, notifiqui al fil en espera que pot continuar la seva execució. Aquests mètodes únicament poden ser cridats dins de mètodes sincronitzats:

```
synchronized public void metodeSincronitzat(){
   //...
}
```

O dins de blocs de codi sincronitzats:

```
public void metode(){
   synchronized (this) {
      //....
     }
//...
}
```

2.2 Gestió de fils

Quan un fil és llançat hauríem de poder controlar el seu comportament per comprovar que la nostra aplicació fa el que hauria de fer. És cert que no tenim el control total sobre la forma en què s'executen el fils, però sí que podem controlar moltes coses. Hem de conèixer com es comporta un fil, comprendre el seu cicle de vida, per, a continuació, veure quines són les possibilitats de control que tenim sobre ells.

2.2.1 Creació i execució d'un fil

Existeixen dues formes de creació de fils a Java:

- Heretant la classe `Thread`.
- Implementant la interfície `Runnable`.

Si volem crear una classe en la qual les seves instàncies siguin fils, haurem d'heretar la classe `Thread`. Aquesta té un mètode anomenat `run()` (*public void run()*) que haurem de sobreescriure per codificar les instruccions a processar durant l'execució del fil. Podríem dir que el mètode `run()` és per un fil, el que és el mètode `main()` per un programa.

L'execució del mètode `run` no garantir una execució en un fil independent. Si volem invocar el processament d'un fil caldrà cridar el mètode `start()`. La seva invocació iniciarà l'execució d'un nou fil just a la primera instrucció del mètode `run()`. És a dir la crida d'*start* implica la invocació posterior del mètode `run`.

Al següent exemple crearem una classe que hereta de la classe `Thread`, per tant, les seves instàncies seran fils. El constructor de la classe rep un `String` que serà la paraula que volem imprimir. Al mètode `run()` implementem el codi que permetrà imprimir 5 vegades la paraula.

```
package heretafil;

public class HeretaFil extends Thread {

    String strImprimir;
    public HeretaFil(String strP) {
        strImprimir=strP;
    }

    public void run(){
        for(int x=0;x<5;x++){
            System.out.println(strImprimir+ " " + x);
        }

    }

    public static void main(String[] args) {

        Thread primer = new HeretaFil("Fil 1");
        Thread segon = new HeretaFil("Fil 2");
        // Hem creat dos fils primer i segon, però no s'han executat.
        // Per poder-lo executar s'ha de cridar al mètode start()
        primer.start();
        segon.start();

        System.out.println("Final Fil Principal");

    }
}
```

Les línies següents creen dos objectes tipus `HeretaFil` que són fils:

```
Thread primer = new HeretaFil("Fil 1");
        Thread segon = new HeretaFil("Fil 2");
```

Un cop creats els fils s'han de posar en execució, per això es crida al mètode de la classe `thread`, `start()`.

El mètode `start()` fa una inicialització del fil i el deixa a l'organitzador de fils. Quan l'organitzador de fils ho decideix, es crida al mètode `run()`, que és qui executa el codi del fil.

```
primer.start();
        segon.start();
```

Aquest exemple ha creat dos fils des del fil principal que han executat el mateix codi. Una possible sortida del programa podria ser:

```
run:
Fil 2 0
Fil 1 0
Fil 2 1
Fil 1 1
Final Fil Principal
Fil 2 2
Fil 1 2
Fil 2 3
```

```
Fil 1 3
Fil 2 4
Fil 1 4
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tal com es pot observar, l'ordre es va intercalant. Tenim tres fils d'execució: el principal i els dos creats. Aquesta pot ser una possible sortida, ja que l'execució dels tres fils és independent i pot ser normal que acabi el fil principal abans que els dos creats per ell.

Ja que Java no suporta l'herència múltiple, si volem crear fils en els quals la nostre classe hereti d'alguna altra classe, s'haurà d'utilitzar la interfície `Runnable`. La classe `Thread` conté un constructor que ens permet passar-li com a paràmetre qualsevol objecte que implementi la interfície `Runnable`: *Thread (Runnable unFil)*.

En l'exemple següent es reprodueix el codi anterior però implementant la interfície `Runnable`.

En aquest cas els objectes de la classe `RunnableFil` no són fils ja que no hereten de la classe `Thread`. Si volem que els objectes de la classe `RunnableFil` s'executin com a fils independents haurem de crear un objecte de la classe `Thread` i passar-los com a paràmetres.

```
package runnablefil;

public class RunnableFil implements Runnable {

String strImprimir;
    public RunnableFil(String strP) {
        strImprimir=strP;
    }

    public void run(){
        for(int x=0;x<5;x++){
            System.out.println(strImprimir+ " " + x);
        }

    }

    public static void main(String[] args) {

        //Creem dos objecte de la classe RunnableFil
        RunnableFil objRunnable1 = new RunnableFil("Fil 1");
        RunnableFil objRunnable2 = new RunnableFil("Fil 2");
        //Creem dos Fils i li passem per paràmetres els objecte de la classe
            RunnableFil
        Thread primer = new Thread(objRunnable1);
        Thread segon = new Thread(objRunnable2);
        // Hem creat dos fils primer i segon, però no s'han executat.
        // Per poder-lo executar s'ha de cridar al mètode start()
        primer.start();
        segon.start();

        System.out.println("Final Fil Principal");

    }
}
```

Podem utilitzar l'extensió de la classe `Thread` o bé una implementació de la interfície `Runnable` indistintament per crear fils. Heretar de la classe `Thread`

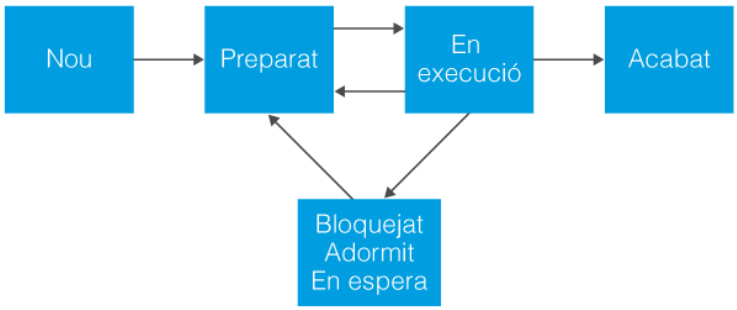
limita les possibilitats de la nostra classe a no poder heretar d'una altra classe en cas que ho necessitem. En canvi, implementar la interfície `Runnable` pot ser una mica més complex, però ens permet heretar d'una altra classe i copiar el seu comportament. És l'opció més utilitzada pels programadors per crear fils.

És possible, però, que el programa principal (el fil principal) finalitzi l'execució abans que els fils creats per ell. Per evitar això, podem utilitzar un mètode de la classe `thread`, `sleep()` que deixa el fil en estat de suspens durant un cert temps. O també utilitzar el mètode `join()` que permet fer que el fil pare esperi que el fils fills acabin.

2.2.2 Estats i canvis d'estat d'un fil

El cicle de vida d'un fil contempla diferents estats pels que pot anar passant durant la seva execució (figura 2.4).

FIGURA 2.4. Estats d'un fil



Nou és l'estat en què es troba un fil quan ha estat creat amb el mètode `new()`. El mètode `start()` posa el fil en estat de **preparat**. Encara no està en execució, si no que està preparat per utilitzar el processador si l'organitzador de fils ho troba oportú. Si l'organitzador li indica que passa a fer ús del processador, entra en l'estat d'**en execució**. De l'estat d'execució pot passar a un estat de **bloquejat**, **adormit** o **en espera** per diferents motius. En aquests estats un fil no s'executa ni es troba a la cua dels fils en espera d'execució. Quan algun esdeveniment el desbloquegi o arribi la hora de despertar-se, el fil transitarà a l'estat de preparat per tornar a ser executat. Direm que un fil està viu (*alive*), mentre transiti entre els estats *preparat*, *en execució*, *bloquejat*, *adormit* o *en espera*.

Quan un fil finalitzi la seva execució arribant al final dl mètode `run`, passarà a l'estat d'acabat i direm que està mort (*dead*), perquè des d'aquest estat no podrà tornar-se a executar més. Així doncs, la invocació del mètode `start` d'un fil acabat, provocarà una excepció en temps d'execució. La finalització d'un fil no implica pas que l'objecte `Thread` desaparegui i alliberi la memòria. Les instàncies de Thread segueixen les mateixes regles que qualsevol altre objecte, mentre es trobin assignades a alguna variable, es mantindran en memòria. Podem forçar

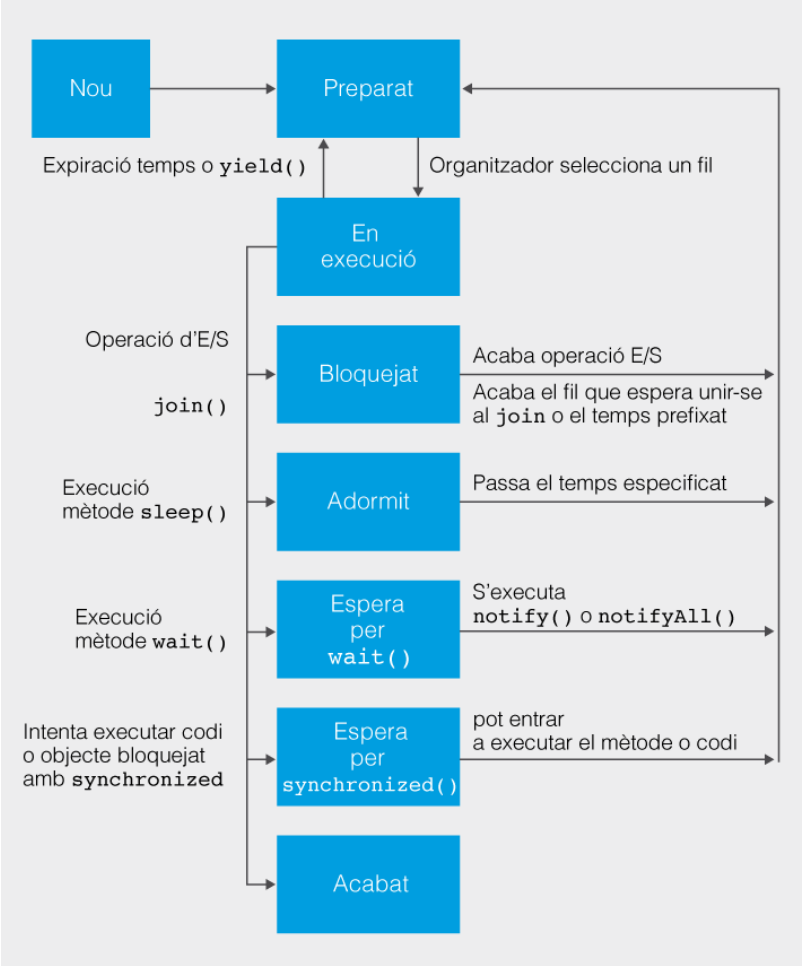
l'alliberament de memòria assignant un valor `null` a la variable que mantenia l'objecte `Thread`, un cop aquest hagi finalitzat.

Mentre el fil està en execució, podrà sortir-ne per diversos motius:

- Ha expirat el temps d'ús del processar per part del fil o s'executa el mètode `yield()`.
- Inicia una operació d'entrada/sortida i passa a un estat de **bloquejat** fins que la operació acabi.
- Crida al mètode `join()` d'un altre fil per esperar la finalització del mateix, El fil invocador passa a l'estat de **bloquejat** fins que el fil invocat passi a l'estat *acabat*.
- Intenta executar un codi que està sincronitzat (*synchronized*) i no pot perquè hi ha un altre fil executant-lo, passa a un estat de **bloquejat** fins que el bloc sincronitzat quedi alliberat.
- Es crida el mètode `wait()` passant-lo a l'estat *d'espera*. Quan un fil entre en estat d'espera, només en podrà sortir si s'invoca algun dels seus mètodes `notify()` o `notifyAll()`.
- S'invoca el mètode `sleep`. El fil passa a estat adormit fins que hagi transcorregut el temps indicat com a paràmetre. L'estat adormit, és molt útil per mantenir un fil aturat durant un temps determinat minimitzant al màxim els recursos utilitzats. Suposem que un programa de tractament de textos, disposa d'un fil que guarda els documents automàticament cada 10 minuts.

La figura 2.5 resumeix tots els canvis i transaccions que són possibles en els estats d'un fil.

FIGURA 2.5. Canvi d'estat d'un fil



La classe `Thread` conté el mètode `isAlive()` que ens indica si un fil ha estat inicialitzat (amb el mètode `start()`). Si `isAlive()` retorna *false* , sabem que el fil està a l'estat de **nou** o **finalitzat**, no sent possible diferenciar aquests dos estats. En canvi si retorna *true* ens diu que el fil ha estat inicialitzat amb `start()` i es trava en qualsevol dels altres estats.

2.2.3 Mètode thread.join()

La classe `thread` conté el mètode no estàtic `join()` que permet que un fil d'execució esperi a un altre. Dit d'una altra manera, la classe `thread` permet, a través del mètode `join()`, que un fil s'uneixi al final de la seva execució a un altre fil.

Si tenim un fil A que no pot completar la seva feina fins que un altre fil B hagi acabat, el que s'ha de fer és que el fil A s'uneixi a al fil B. El fil A estarà en estat de **bloquejat** fins que el fil B hagi acabat la seva execució i estigui en estat d'**acabat** (*dead*). El que farà `fil.join()`, doncs, és esperar en estat de bloquejat, fins que el fil que l'ha invocat acabi. En cas que el fil que l'ha invocat ja hagi acabat ignora l'ordre.

També és utilitzat per unir el fils d'execució d'una aplicació per tal d'evitar que el fil principal acabi abans que els fils creats en el codi de l'aplicació.

En la figura 2.6 es veu com des d'un fil (el fil principal) es creen 3 fils que executen les seves funcions (tirar un dau) i s'esperen que el fil que l'ha invocat acabi (el fil principal).

La figura correspon al següent codi, que crea tres fils i cada un llença un dau. El programa principal ha de mostrar el resultat de la suma de les tres tirades:

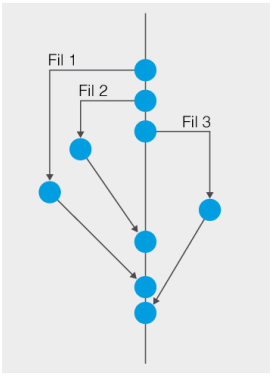
```
1  package joinfils;
2
3  class TiradaDaus {
4      private int tiradaDau;
5
6      public TiradaDaus (int e) {
7          tiradaDau=e;
8      }
9
10     public synchronized int getSumaTirada() {
11         return tiradaDau;
12     }
13
14     public synchronized void setSumaTirada(int e) {
15         tiradaDau += e;
16     }
17    }
18
19
20 public class JoinFils implements Runnable {
21
22     private TiradaDaus xobj;
23
24     public JoinFils(TiradaDaus m) {
25         xobj=m;
26     }
27     public void run(){
28         try{
29             Thread.sleep(1000);
30             int resultatDau=(int) (Math.random()*6) + 1;
31             xobj.setSumaTirada(resultatDau);
32             System.out.println("Tirada fil "+Thread.currentThread().getName() +
                   ": " +resultatDau);
33         }catch (InterruptedException e){
34         }
35     }
36
37     public static void main(String[] args) throws InterruptedException {
38         TiradaDaus ans=new TiradaDaus(0);
39
40         JoinFils obj1 = new JoinFils(ans);
41         JoinFils obj2 = new JoinFils(ans);
42         JoinFils obj3 = new JoinFils(ans);
43         Thread fil_1 = new Thread(obj1);
44         fil_1.setName("Dau 1");
45         Thread fil_2 = new Thread(obj2);
46         fil_2.setName("Dau 2");
47         Thread fil_3 = new Thread(obj3);
48         fil_3.setName("Dau 3");
49         fil_1.start();
50         fil_2.start();
51         fil_3.start();
52         fil_1.join(); //Espera el fil_1 que el fil principal, el que l'ha
               invocat acabi
53         fil_2.join();
54         fil_3.join();
55
```

```
56          System.out.println("Total tirada: "+ ans.getSumaTirada());
57          System.out.println("Final Fil Principal");
58
59      }
60
61  }
```

Resultat:

```
1  run:
2  Tirada fil Dau 1: 2
3  Tirada fil Dau 3: 1
4  Tirada fil Dau 2: 5
5  Total tirada: 8
6  Final Fil Principal
7  BUILD SUCCESSFUL (total time: 1 second)
```

FIGURA 2.6. Esquema de com actua el mètode "join" aturant l'execució del fil principal a l'espera que acabin els tres fils que ha creat



Evidentment per treure la suma de les tres tirades, en la línia:

```
1  System.out.println(ans.getSumaTirada());
```

hem d'esperar que els tres fils acabin l'execució. I per fer-ho hem de fer que el fil principal (en aquest cas ja que els fils s'han executat des del *main*) esperi per acabar la seva execució que els tres fils acabin. Això ho fem amb el mètode `join`.

```
1  fil_1.join(); //Espera el fil_1 que el fil principal, el que l'ha invocat,
       acabi
2  fil_2.join();
3  fil_3.join();
```

Si no esperem la finalització dels fils, el fil principal dóna de resultat la suma de les tres tirades i pot ser un resultat erroni, ja que és possible que algun dels tres fils no hagi acabat. Sense el `join` aquest seria un possible resultat:

```
1  run:
2  Tirada fil Dau 1: 4
3  Tirada fil Dau 2: 1
4  Total tirada: 5
5  Final Fil Principal
6  Tirada fil Dau 3: 6
7  BUILD SUCCESSFUL (total time: 1 seconds)
```

Tal com podem veure, el programa principal mostra el resultat únicament de dos tirades, els fils que s'han executat abans de la finalització del fil principal. L'última tirada no l'ha tinguda en compte.

2.3 Sincronització de fils

En entorns multifils que s'executen concurrenment és habitual que entre els diferents fils d'execució hi hagi la necessitat de compartir dades. Aquesta compartició de dades ens pot servir per comunicar fils. Utilitzarem variables compartides per indicar que un fil ha acabat una certa funció o no i, per tant, un altre fil podrà executar la part de codi que estava bloquejat per aquesta variable. També podem compartir dades d'un objecte comú. Diversos fils han de modificar una propietat d'un objecte. En ambdós casos, convé sincronitzar l'accés als recursos compartits per evitar incosistència de dades o interferències. Qualsevol variable accessible pot ser manipulada per un fil o el fil principal. Si aquest accés no es controla pot provocar situacions no esperades o errònies.

Treballant amb fils aquesta compartició de variables és molt fàcil i gens costosa, ja que els fils d'un mateix procés comparteixen el mateix espai de memòria.

Java proporciona mecanismes per sincronitzar els fils que tracten aquests problemes.

Quan més d'un fil ha d'accedir al mateix recurs és quan apareix la possibilitat d'operacions que s'executen concurrentment utilitzant el mateix objecte o recurs i, per tant, es poden donar casos de dades corruptes o resultats no esperats. Aquestes zones de codi que accedeixen a un recurs compartit i que no poden ser accessibles al mateix temps per més d'un fil, s'anomenen seccions crítiques. Un efecte equivalent és quan diversos fils volen executar el codi d'una secció crítica. Els fils cooperen utilitzant el protocol que, per realitzar algunes operacions sobre un objecte, han d'agafar el bloqueig de l'objecte. Agafar el bloqueig d'un objecte significa que no deixa que altres fils agafin aquest bloqueig fins que el fil que el té l'alliberi. Això s'anomena exclusió mútua. Si aquesta acció es realitza correctament, el fils d'execució no realitzaran operacions que interfereixin entre si.

Allò que s'intenta és que les operacions sobre les seccions crítiques siguin atòmiques. Que s'executin totalment o no s'executin.

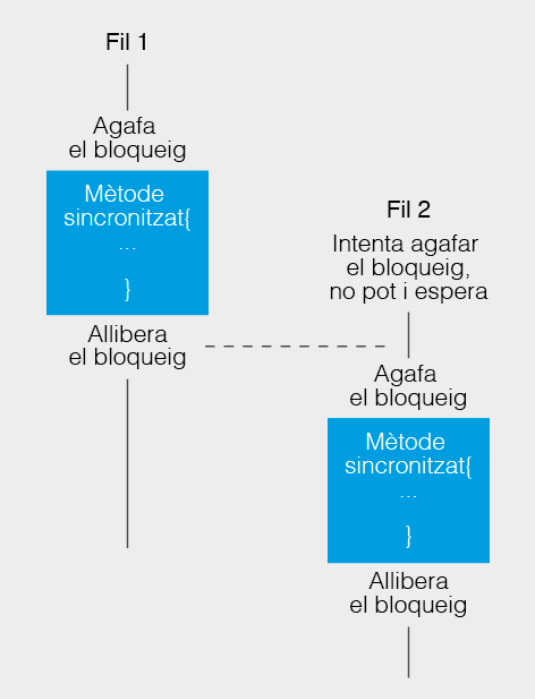
Per defecte, a Java un objecte no està protegit. Això vol dir que qualsevol nombre de fils pot executar codi dins de l'objecte. L'exclusió mútua s'aconsegueix a Java amb la paraula reservada `synchronized`. Aquesta paraula pot ser aplicada a blocs de codi dins d'un mètode i també a mètodes sencers.

2.3.1 Mètodes synchronized

Si volem protegir els objectes d'una classe de possibles interferències a un entorn multifil, és a dir, que els mètodes d'aquesta classe s'executin en exclusió mútua, haurem de posar el modificador ***synchronized*** als mètodes que formin part de la secció crítica. El modificador *synchronized* ens garanteix que cap altre mètode sincronitzat de l'objecte podrà executar-se. Quan un fil executa un mètode sincronitzat agafa el bloqueig de l'objecte fins que acaba la seva execució i s'allibera del bloqueig. Si altre fil crida el mètode sincronitzat del mateix objecte, queda bloquejat fins que quedi el fil que l'està executant l'alliberi del bloqueig.

En la figura 2.7 es mostra com dos fils intenten executar un mètode sincronitzat. Un agafa el bloqueig i l'altre es queda a l'espera que l'alliberi per poder executar el mètode.

Figura 2.7. Bloqueig-Alliberar mètode sincronitzat



Els mètodes sincronitzats proporcionen exclusió mútua per accedir al recurs compartit. Els mètode no sincronitzats del mateix objecte poden ser executats per diversos fils a la vegada. La possessió del bloqueig d'un mètode és única per fil. És a dir, un fil no pot bloquejar dos cops el mateix mètode, però sí que pot fer crides recursives a un mètode bloquejat per ell. La forma natural d'alliberar un bloqueig és que el fil arribi al final del codi del mètode o trobi una sentència `return`. També pot alliberar-lo per altres motius, com ara que es llanci una excepció.

Aquest sistema que ens proporciona Java d'exclusió mútua fa que el programador no hagi de recordar de posar *bloqueig* i *alliberar bloqueig* a cada mètode. Java ja ho fa per nosaltres.

En el següent codi es pot veure com sincronitzem alguns mètodes d'una classe i d'altres no. Aquells que volem que s'executin en exclusió mútua els sincronitzem amb la paraula reservada `synchronized`.

```
1  public class sincronitzaMetodes {
2
3   public void metodeNoSincronitzat_1(){
4   //.........
5   }
6
7    public synchronized void metodeSincronitzat_1(){
8   //.........
9   }
10   public synchronized void metodeSincronitzat_2(){
11  //.........
12  }
13     public void metodeNoSincronitzat_2(){
14  //.........
15  }
16 }
```

En una situació en la qual dos fils estan accedint a la mateixa instància d'una classe i volen executar el mateix mètode accedint i modificant simultàniament el mateix objecte i mateixes variables, si aquest accés no està controlat i les dades que modifiquen els fils serveixen per indicar com continua el programa funcionant, el resultat serà amb tota seguretat erroni i inesperat.

Imaginem una agència de viatges que venen passatges d'avió. Hi ha dues persones interessades en viatjar amb la seva família a Roma al vol SP233 a les 20.00h. El primer que hauria de fer el sistema és mirar si hi ha places suficients al vol i si n'hi ha, descomptar les places de l'avió i assignar-les al client.

Però què passaria si les dues persones volen fer la reserva al mateix temps?

Al `main` de la classe `AgenciaViatges` es creen dos fils sobre la mateixa instància de l' `object`. Quan s'executen al mètode `run()`, el dos clients intenten fer la reserva de 3 seients per la seva família de forma simultània. Inicialment només hi ha 5 seients lliures.

En el mètode `run` es crida al mètode `gestioSeientsLliures(3)`. Aquest mètode el que farà és: si hi ha tres seients lliures els assigna al client (és a dir, els resta dels seients lliures).

Si hi ha seients lliures s'executa un `sleep()` d'1 segon, que farà adormir el fil per simular un sistema més real amb més volum de dades.

> El mètode `sleep(milisegons)` deixa el fil en estat de **bloquejat** durant un temps marcat en milisegons i un cop passat aquest temps torna a l'estat de **preparat**. Durant aquest temps si el fil tenia el bloqueig d'un objecte, el continuarà tenint. No perd el bloqueig durant aquest temps.

Després de despertar el fil, crida el mètode `reservaSeients()` de la classe `SeientsAvio` que restarà els seients reservats dels seients lliures.

```
package agenciaviatges;


//Implementem runnable per poder crear fils sobre les seves instàncies
public class AgenciaViatges implements Runnable {

    private SeientsAvio sa = new SeientsAvio();

    public void run(){
                gestioSeientsAvio(3);
    if(sa.getSeientsLliures()<=0)
                   System.out.println("No hi ha seients lliures");
         }

    public void gestioSeientsAvio(int numSeientsReserva){

//Comprovem si hi ha seients suficients
        if(sa.getSeientsLliures()>=numSeientsReserva)
        {
            System.out.println(Thread.currentThread().getName()+" farà una
                 reserva de "+ numSeientsReserva +" places");
            try {
                    Thread.sleep(1000); //adormim el fil 1 segon
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
             }
  //Fem la reserva dels seients
            sa.reservaSeients(numSeientsReserva);

            System.out.println(Thread.currentThread().getName() + " Reserva
                 realitzada amb èxit."+" Les places lliures són "+sa.
                 getSeientsLliures());
         }else{
            System.out.println("No hi ha places suficients pel client." +
                 Thread.currentThread().getName()+" Les places lliures són "+sa
                 .getSeientsLliures());
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }


    }


    public static void main(String[] args) {

        AgenciaViatges objAg = new AgenciaViatges();

        //creem els dos fils sobre la mateixa instància
        Thread Fil_1 = new Thread(objAg);
        Thread Fil_2 = new Thread(objAg);
        Fil_1.setName("Client 1");
        Fil_2.setName("Client 2");

        Fil_1.start();
        Fil_2.start();

    }
}


class SeientsAvio {
   //comencem amb 5 seients lliures a l'avió
    private int seientsLliures = 5;
```

```

    public int getSeientsLliures(){
        return seientsLliures;
    }

    public boolean getSeientsLliures(int numPlaces){
        if (numPlaces<=seientsLliures)
            return true;
        else
            return false;
    }

    public void reservaSeients(int numSeientsReserva){
        seientsLliures = seientsLliures - numSeientsReserva;
    }

}
/
```

Un possible resultat és:

```
run:
Client 1 farà una reserva de 3 places
Client 2 farà una reserva de 3 places
Client 1 Reserva realitzada amb èxit. Les places lliures són 2.
Client 2 Reserva realitzada amb èxit. Les places lliures són -1
BUILD SUCCESSFUL (total time: 2 seconds)
```

S'ha produït un error inesperat, hem reservat places que no existeixen. Hem creat un *overbooking* a l'avió.

Aquest error s'ha produït perquè el Client 1 ha consultat les places lliures de l'avió i ha vist que n'hi havia 5. S'ha adormit durant 1 segon i mentre estava adormit i abans de fer la reserva (és a dir, restar els 3 seients), el Client 2 ha consultat el número de seients i ha vist que també en quedaven 5 i que, per tant també podia fer la seva reserva, ignorant que el Client 1 està pendent de reservar les seves. Finalment tots dos completen el procés i creen un número negatiu de seients lliures.

La solució a aquest problema és utilitzar semàfors sobre les seccions crítiques. Les seccions crítiques en aquest exemple són la consulta de seients lliures i la resta de seients reservats. És a dir, les operacions de `gestioAvio()`.

Un semàfor és com una cadenat amb una única clau. Aquesta clau únicament la tindrà el client que l'agafa primer i cap altre client pot agafar-la fins que ell la deixi.

El que farem serà protegir els mètodes de la secció crítica amb aquest cadenat, de manera que si el Client 1 consulta els seients lliures agafa la clau i tanca el semàfor, quan el Client 2 vol consultar el seients lliures, necessita la clau i com que no la pot agafar es queda en espera fins que el Client 1 acaba la seva transacció (restar els seients reservat dels lliures) i allibera la clau.

Java pot crear semàfors amb la paraula clau *synchronized*. Posarem el cadenat al mètode encarregat de fer tota la gestió de consulta i reserva de seients, és a dir, al mètode `gestioAvio()`. D'aquesta manera podrem estar segurs que quan un client agafa la clau del cadenat cap altre client pot fer aquesta gestió.

Per fer-ho, hem d'afegir dues línies més a l'inici i al final del mètode que mostren qui i quan agafa la clau i després l'allibera.

```
public synchronized void gestioSeientsAvio(int numSeientsReserva){

    System.out.println(Thread.currentThread().getName() + " té la clau del
        cadenat");
//Comprovem si hi ha seients suficients
        if(sa.getSeientsLliures()>=numSeientsReserva)
        {
            System.out.println(Thread.currentThread().getName()+" farà una
                reserva de "+ numSeientsReserva +" places");
            try {
                    Thread.sleep(1000); //adormim el fil 1 segon
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
             }
  //Fem la reserva dels seients
            sa.reservaSeients(numSeientsReserva);

            System.out.println(Thread.currentThread().getName() + " Reserva
                realitzada amb èxit"+" Les places lliures són "+sa.
                getSeientsLliures());
         }else{
            System.out.println("No hi ha places suficients pel client: " +
                Thread.currentThread().getName()+" .Les places lliures són "+
                sa.getSeientsLliures());
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
     System.out.println(Thread.currentThread().getName() + " deixa la clau del
         cadenat.");

    }
```

El resultat serà el següent

```
run:
Client 1 té la clau del cadenat.
Client 1 fa una reserva de 3 places.
Client 1 reserva realitzada amb èxit. Les places lliures són 2.
Client 1 deixa la clau del cadenat.
Client 2 té la clau del cadenat.
No hi ha places suficients pel Client 2. Les places lliures són 2.
Client 2 deixa la clau del cadenat.
BUILD SUCCESSFUL (total time: 2 seconds)
```

Tal com es pot veure, el Client 1 agafa la clau del cadenat i fa tota la gestió: consulta el número de seient lliures i fa la reserva dels seients. Quan acaba tot el procés deixa la clau.

Posteriorment agafa la clau el Client 2, que quan consulta el número de seients lliures s'adona que no n'hi ha suficients per fer la seva reserva. Acaba dient que no hi ha places suficients i allibera la clau.

2.3.2 Objectes synchronized

Quan sincronitzem mètodes estem bloquejant aquests mètodes per tal que únicament un fil pugui executar-los a la vegada. Però si sincronitzem molt, tots els avantatges que ens proporciona la concurrència es perden. Per tant, hem de sincronitzar únicament aquell codi que ens interessa. Amb Java podem sincronitzar només una part de codi dins d'un mètode, aquella part que volem que s'executi en exclusió mútua. Per sincronitzar únicament una part del codi necessitem fer referència a un objecte. El següent codi mostra com es sincronitza una part del codi sobre l'objecte que ha cridat al mètode `metodeNoSincronitzat()`.

La part del codi sincronitzada farà que s'executi en exclusió mútua amb qualsevol altre bloc sincronitzat del mateix objecte. També s'executaran en exclusió mútua els mètodes amb el modificador *synchronized*.

```
class Sincronitzat {

    public void metodeNoSincronitzat(){
        //Part de codi no sincronitzat

        synchronized (this){
            //Codi sincronitzat
        }

        //Part de codi no sincronitzat

    }
}
```

En l'exemple anterior podríem modificar el codi per sincronitzar un bloc de codi de la següent manera:

```
public class AgenciaViatges implements Runnable {

    private SeientsAvio sa = new SeientsAvio();
    public void run(){
        //Codi no sincronitzat
       System.out.println("Seients Lliures: " + sa.getSeientsLliures());
       synchronized(sa){
                gestioSeientsAvio(3);
                if(sa.getSeientsLliures()<=0)
                   System.out.println("No hi ha seients lliures");
         }
//Codi no sincronitzat

}

    public  void gestioSeientsAvio(int numSeientsReserva){...
//.
}
```

En l'exemple no es guanya gaire ja que no hi ha codi per sobre del bloc sincronitzat. També hem de veure que no cal sincronitzar el mètode `gestioSeientsAvio(int)` , ja que al cridar-lo des d'un un bloc sincronitzat, l'objecte el manté bloquejat. Per sincronitzar el bloc podríem haver posar `synchronized(this)`, ja que l'objecte que bloquegem és el mateix que invoca el mètode.

També podem sincronitzar part del codi sobre un objecte diferent a aquell que ha cridat el mètode que s'està executant. Abans d'executar el codi que està sincronitzat, hem d'agafar el bloqueig de l'objecte `altreObjecte`. Cap altre fil haurà de tenir el bloqueig d'aquest objecte.

```
class SincronitzatAltreObjecte {

    public void metodeNoSincronitzat(){
        //Part de codi no sincronitzat

        synchronized (altreObjecte){
            //Codi sincronitzat per l'objecte altreObjecte
        }

        //Part de codi no sincronitzat

    }
}
```

És un mètode que podem fer servir si volem realitzar més d'una operació atòmica sobre un objecte, com per exemple consultar les places lliures d'un avió i després realitzar una reserva.

```
synchronized (seientsAvio){
      if (seientsAvio.getSeientsLliures(numPlaces))
              seientsAvio.reservaSeients.(numPlaces);
      }
```

2.3.3 Sincronitzar mètodes estàtics

Els mètodes estàtics també es poden sincronitzar. Ja que únicament es protegeix una còpia de les dades estàtiques que es manipulen, únicament es necessita un bloqueig per classe.

Totes les instàncies tenen un objecte associat a java.lang.Class. Aquesta instància és la que s'utilitza per protegir (sincronitzar) els mètodes estàtics.

Si volem que un mètode protegeixi una classe sencera, tots el objectes creats d'una classe, haurem de sincronitzar un mètode estàtic.

La sincronització de mètodes estàtics és similar als mètodes no estàtics.

```
public static synchronized void metodeSincronitzatEstatic() {...
 //.
}
```

També podem sincronitzar blocs de codi de mètodes estàtics. Per fer-ho hem d'indicar quina classe volem sincronitzar. En l'exemple, `LaMevaClasse.class` és una classe literal. El que fa la màquina virtual de Java és anar a buscar la instància de `Class` que està representant la classe que crida `LaMevaClasse`.

```
public static void metodeNoSincronitzat() {
//Codi no sincronitzat
synchronized(LaMevaClasse.class) {
```

```
//Codi sincronitzat
}
//Codi no sincronitzat
}
```

El següent codi és equivalent a l'anterior:

```
public static void metodeNoSincronitzat() {
//Codi no sincronitzat
Class lmc = Class.forName("LaMevaClasse");
synchronized(lmc) {
//Codi sincronitzat
}
//Codi no sincronitzat
}
```

2.3.4 Variables volatile

Les variables primitives a Java són atòmiques en lectura i actualització però no en operacions. Per tant, podem ometre'n la sincronització si l'únic que volem fer és modificar un valor d'una variable primitiva. D'aquesta manera augmentem el rendiment de l'aplicació. Hem de tenir en compte, però, un aspecte important. La màquina virtual de Java optimitza codi quan els compila si es troba un codi com el següent:

```
class mètodeExemple {
private bolean valor = false;
public void cambiarValor () {
valor = true;
}
public class classeExemple {

    private boolean flag = false;

    public void cambiarFlag(){
        flag=true;
    }

    public synchronized void metodeExemple () {
        flag = false;
        // un altre fil prodria cridar al mètode cambiarFlag()
        if (flag) {
            // ...
        }
     }
 }
```

La màquina virtual de Java pot optimitzar el codi pensant que el que es troba dins d' `if` (*flag*) no s'executarà mai, ja que *flag* no pot ser mai *true* perquè la línia anterior el posa a *false*. Si creu que és codi mort no s'executarà mai. Però un altre fil pot modificar el valor de *flag* accedint al mètode `cambiarFlag()` quan un primer fil està a punt d'entrar a `if` i, per tant, sí que es podria entrar al bloc d' `if`.

La forma d'evitar-ho és declarar la variable *flag* com a ***volatile***. D'aquesta manera li podem dir a la màquina virtual de Java que un altre fil pot modificar aquest

valor. La paraula reservada ***volatile*** aplicada sobre variables indica que se'n farà l'actualització de forma atòmica.

2.3.5 Interbloqueig

Els avantatges de sincronitzar mètodes a Java són evidents. No podríem realitzar concurrència sense aquest tipus de seguretat i bloquejos. Però hem d'anar amb cura quan apliquem aquests mètodes. En primer lloc, només hem de sincronitzar aquells mètodes o blocs de codi necessaris, ja que abusar de la sincronització penalitza el rendiment de l'aplicació. En segon lloc perquè si no sincronitzem correctament es podem produir errors. Evidentment podem realitzar bloquejos sense fer servir les primitives que ens proporciona Java, però la font d'errors és major, ja que hem de recordar d'agafar el bloqueig i alliberar-lo, cosa que amb *synchronized*, tal com hem vist, no és necessària.

L'error més greu que es pot produir quan es treballa en programació multifil relativament complexa és l'anomenat **interbloqueig** (en anglès *deadlock*). L'interbloqueig es produeix quan dos fils s'estan executant concurrentment i cada un té un bloqueig exclusiu que l'altre necessita per continuar. El fenomen pot passar amb més de dos fils i per norma general passa quan els fils agafen els mateixos bloquejos en diferent ordre. La realitat és que és difícil que passi un interbloqueig en aplicacions simples amb un bloqueig o dos, però quan la programació és més complexa, amb molts bloquejos i alliberaments a tot el codi, el problema és molt real.

Imaginem una aplicació que gestiona els recursos d'una entitat bancària (comptes bancaris, documents, etc.) que permetin que les dades puguin ser transferides d'un recurs a un altre. Cada recurs té un bloqueig associat per a ell. Si es vol fer una transferència, el fil ha d'agafar el bloqueig de tots els recursos que intervenen en la transferència. Si dos fils inicien la transferència que afecta a dos recursos i els fils agafen el bloqueig dels recursos en diferent ordre ja tenim un interbloqueig.

```
1  public class FerTransferencia {
2      private static class CompteCorrent {
3          //Codi de la classe
4      }
5      private CompteCorrent transferenciaA = new CompteCorrent();
6      private CompteCorrent transferenciaB = new CompteCorrent();
7
8      public int metodeTransferenciaArxius() {
9          synchronized(transferenciaA) { // Possible interbloqueig
10             synchronized(transferenciaB) {
11                 //Codi del mètode
12             }
13         }
14     }
15     public void metodeTransferenciaCalers() {
16         synchronized(transferenciaB) { // Possible interbloqueig
17             synchronized(transferenciaA) {
18                 //Codi del mètode
19             }
20         }
21     }
22 }
```

La classe és estàtica, cosa molt usual quan volem protegir amb sincronització els objectes d'una aplicació. Si mirem el codi anterior i imaginem un fil que comença a executar el mètode `metodeTransferenciaArxius` i agafa el bloqueig sobre l'objecte `transferenciaA`. Un segon fil entra i executa el mètode `metodeTransferenciaCalers` i agafa el bloqueig sobre el mètode `transferenciaB`.

En aquest moment s'ha produït un interbloqueig. El primer fil espera que s'alliberi la `transferenciaB` per poder agafar el bloqueig i continuar la seva execució. En canvi, el segon fil està esperant que s'alliberi `transferenciaA` per poder agafar el bloqueig i continuar la seva execució. Evidentment cap de les dues coses passarà. Hi haurà una espera infinita, un interbloqueig. És possible que aquest mateix codi s'executi sempre bé, sense que es produeixi un interbloqueig mai, però la probabilitat hi és.

En aquest cas, la solució és realment senzilla. Només cal assegurar que la sincronització d'ambdós objectes (transferenciaA i transferenciaB) es produeixi sempre en el mateix ordre. En el codi que ens ocupa, simplement caldrà canviar les ordres de sincronització en un dels mètodes, per exemple en `metodeTransferenciaCalers` perquè per tal que l'ordre de sincronització coincideixi amb el del mètode `metodeTransferenciaArxius`.

No sempre és, però, tan trivial de detectar un possible codi amb interbloqueig. En el següent exemple, hi ha un mètode que s'encarrega de fer una transferència entre dos comptes, de `compteOrigen` a `compteDesti`. El codi bloqueja els dos comptes perquè la transferència sigui segura i atòmica.

```
1  public void transferirDiners(CompteCorrent compteOrigen, CompteCorrent
       compteDesti,double quantitat) {
2      synchronized (compteOrigen) {
3        synchronized (compteDesti) {
4          if (compteOrigen.hihaSaldoSuficient(quantitat) {
5            compteOrigen.reintregre(quantitat);
6            comptedesti.ingres(quantitat);
7          }
8        }
9      }
10   }
```

Ara bé, que un fil invoca la següent sentència:

```
1  transferirDiners(compte2100, compte8888, quantitatIngressar1);
```

El fil intentarà bloquejar el primer el compte `compte2100` i posteriorment el compte `compte8888` abans d'executar el codi sincronitzat.

Però que passarà, si mentre es realitza el bloqueig de `compte2100`, un altre fil invoca la següent instrucció?

```
1  transferirDiners(compte8888, compte2100, quantitatIngressar2);
```

El que succeirà és que el segon fil agafarà el bloqueig del compte `compte8888` i en intentar realitzar el bloqueig de l'altre compte, es queda a l'espera perquè el primer fil el mantindrà bloquejat. El primer fil restarà també boquejat esperant

que el compte `compte8888` quedi lliure, però això no es produirà mai, doncs ens trobem davant d'una situació d'interbloqueig.

Per solucionar-ho, haurem d'aconseguir de nou, que els bloquejos es facin sempre en el mateix ordre, independentment de quin sigui el compte d'origen i quin el compte de destí.

En el següent codi veurem una possible solució que passa per crear dos objectes de tipus `compteCorrent` i assignar-li el compte d'origen o destí en funció del seu ordre numèric. Hem creat també un mètode que ens retorna el número de compte. Ara, sempre agafarà el bloqueig en el mateix ordre independentment de la crida que es faci a `transferirDiners()`.

```
1  public void transferirDiners(CompteCorrent compteOrigen, CompteCorrent
       compteDesti,double quantitat) {
2     CompteCorrent primerBloqueig, segonBloqueig;
3      if (compteOrigen.numeroCompte() == compteDesti.numeroCompte())
4        throw new Exception("No es pot fer una transferència sobre el mateix
             compte");
5      else if (compteOrigen.numeroCompte() < compteDesti.numeroCompte()) {
6        primerBloqueig = compteOrigen;
7        segonBloqueig = compteDesti;
8      }
9      else {
10       primerBloqueig = compteDesti;
11       segonBloqueig = compteOrigen;
12     }
13     synchronized (primerBloqueig) {
14       synchronized (segonBloqueig) {
15
16     synchronized (compteOrigen) {
17       synchronized (compteDesti) {
18         if (compteOrigen.hihaSaldoSuficient(quantitat) {
19           compteOrigen.reintregre(quantitat);
20           comptedesti.ingres(quantitat);
21         }
22       }
23     }
24   }
```

La part més important per evitar interbloquejos és la documentació. Si creem una bona documentació explicant on s'agafen els bloquejos, quins objectes hi intervenen i com han de ser les crides als mètodes, ens estalviarem molts problemes quan algun altre programador hagi de fer un manteniment de l'aplicatiu. Els programadors sabran que cridar un mètode en concret pot implicar un interbloqueig.

2.4 Compartir informació entre fils

La paraula clau *synchronized* ens dóna seguretat i integritat en mètodes o parts de codi. Però a més de sincronització i protecció, els fils també han de poder comunicar-se entre ells. Java ens proporciona mètodes que fan que els fils esperin i continuïn la seva execució. Són els mètodes: `wait()`, `notify()` i `notifyAll()`.

2.4.1 Mètodes wait(), notify() i notifyAll()

Sincronitzar mètodes ens proporciona una exclusió mútua per accedir a una secció crítica. Però si existeixen fils interdependents o un fil ha d'esperar una condició per poder continuar la seva execució i aquesta condició l'ha de donar un altre fil, el sincronisme no ho soluciona.

Imaginem dos fils que controlen l'enviament i el processament de correus electrònics. El primer fil està buscant correus electrònics per posar a la cua d'enviats i el que processa els correus els construeix per deixar-los a la cua. Per què el primer fil ha d'estar malgastant temps i recursos cada dos segons buscant a la cua de correus per enviar si encara no n'hi ha?. És millor si el posem en suspensió i quan arribi un correu a la cua ja l'avisarem i verificarem el correu per enviar-lo.

Els mètodes que proporciona Java per esperes i notificacions són:

- `wait()`: treu el fil en curs i manté el bloqueig de la zona d'exclusió mútua. L'envia a una **cua d'espera** o **conjunt d'espera** (*wait set* en anglès) i el passa a l'estat d'**en espera**.

Haurà d'esperar a la cua fins que un altre fil el tregui d'allà i el canviï a l'estat de **preparat** mitjançant els mètodes `notify()` o `notifyAll()`.

- `notify()`: un fil de la cua d'espera és seleccionat per passar a l'estat de **preparat**.
- `notifyAll()`: tots els fils del la cua d'espera passen a l'estat de **preparat**.

Tant `notify()` com `notifyAll()` posen a *preparat* els fils de la cua d'espera, si n'hi ha. Si no hi ha cap fil esperant, la instrucció és ignorada.

Aquests mètodes són de la classe `object`, per tant són heretats per totes les classes de forma implícita.

> Els tres mètodes, `wait()`, `notify()` i `notifyAll()`, han d'executar-se sobre el fil que té el bloqueig. Per tant, totes les crides a aquests mètodes s'han de fer des de mètodes o blocs de codi sincronitzats.

La utilització de sincronisme i comunicació de fils a Java és la forma de crear monitors. Recordem que un monitor és una estructura d'alt nivell que gestiona la concurrència de forma que només permet a un únic fil agafar el monitor (un únic fil hi té accés a la vegada) i conté una llista de fils en espera per entrar a gestionar el seu accés.

Existeix una forma més o menys estàndard d'implementar els fils que han d'esperar que succeeixi un esdeveniment per poder continuar l'execució.

El codi següent mostra de quina manera, si una condició no és certa, el fil passarà a l'estat *en espera* i deixarà lliure el bloqueig sobre l'objecte. Quan un altre fil li notifiqui que pot continuar la seva execució i s'executi, tornarà a avaluar la condició i, si és certa, executarà el codi. L'encarregat de notificar que pot continuar i que ha de sortir de la cua d'espera és un mètode que fa que la condició es faci certa.

```
1  synchronized void metodeWait () {
2  while (!condicio)
3  try {
4  wait();
5  }
6  catch (InterruptedException e) {
7  // codi d'excepció
8  }
9  // codi que s'executa quan la condició és certa.
10 }
11
12 synchronized void metodeCanviCondicio(boolean valor) {
13 condicio = valor;
14 notify(); // o notifyAll()
15 }
```

Tot allò que hem de tenir en compte quan programem mètodes d'espera i notificació és:

- El mètode `metodeWait()` ha d'executar-se sincronitzat per tal d'assegurar que en canviar la condició del `while` i produir-se la notificació, només un fil aconseguirà executar el codi situat a la sortida del bucle.
- El mètode `metodeCanviCondicio()` modifica el valor de la condició usada en el `while` del mètode anterior i notifica el canvi per tal que si hi ha algun fil esperant dins del bucle, s'alliberi. El fil desbloquejat tornarà a bloquejar-se en cas que la condició avaluada prengués el valor cert, però aconseguiria sortir del bucle i continuar l'execució en cas que la condició fos falsa. Aquestes dues operacions s'han d'executar sempre de forma atòmica, doncs en cas contrari perdria la seva efectivitat, perquè podria passar que es cridés el mètode `notify()` entre l'alliberació del bloqueig i el canvi d'estat a **en espera**. En aquest cas el mètode `notify()` no afectaria al fil.
- Un fil que és alliberat a través d'un `notify()`passarà a l'estat de **preparat**. És a dir, no vol dir que s'executi immediatament sinó que tornarà a lluitar per agafar el bloqueig amb altres fils en el mateix estat.
- Quan un fil passa pel `wait()` es queda en espera immediatament, i no fa res per veure si allò que estava esperant ja ha passat. Això només passa si el fil que executa el mètode `metodeCanviCondicio()`, invoca `notify()`.

En el següent codi d'exemple veurem com aplicar l'espera i la notificació en un programa petit.

La idea és crear dos fils: un ha d'escriure *hola* i l'altre *adéu*. Però s'ha de fer en ordre, primer imprimim *hola* i després *adéu*. Si no implementem una espera i una notificació i llancem els dos fils, l'ordre de impressió podria ser erroni.

```java
package holaadeum;

public class HolaAdeuM {
    public static void main(String args[]) {
        escriuHolaAdeu eha = new escriuHolaAdeu();
  new E_Adeu(eha);
        new E_Hola(eha);
    }
}


class E_Hola implements Runnable {
      escriuHolaAdeu eh;
      E_Hola (escriuHolaAdeu eh) {
    this.eh =  eh;
    new Thread(this, "Hola").start();
      }
      public void run() {
            try{
                for(int x=0;x<5;x++) {
                    Thread.sleep(1000);
                    eh.eHola();
                }
            }catch (InterruptedException e){
            }
      }
}

class E_Adeu implements Runnable {
      escriuHolaAdeu eh;
      E_Adeu (escriuHolaAdeu eh) {
    this.eh =  eh;
    new Thread(this, "Adéu").start();
      }
      public void run() {
    for(int x=0;x<5;x++) {
          eh.eAdeu();
    }
      }
}

class escriuHolaAdeu {

    boolean escritHola = false;

    public synchronized void eAdeu() {
        while (escritHola == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        escritHola = false;
        System.out.println(" Adéu ");
    }

    public synchronized void eHola() {
        System.out.println(" Hola ");
        escritHola = true;
        notify();
    }

}
```

La classe `escriuHolaAdeu()` és la que conté els mètodes d'escriptura i la lògica de l'espera i notificació.

El mètode `eAdeu()` és l'encarregat d'escriure *Adéu*. Com que aquest mètode ha d'esperar que *hola* estigui escrit, va comprovant la variable booleana `escritHola` i, mentre sigui falsa, envia el fil a la cua d'espera. Únicament quan el mètode `eHola()` és executat per un fil que imprimeix el missatge, col·loca la variable a `true` i posteriorment llança un `notify()` que allibera el fill que es troba en espera.

Quan el fil és seleccionat per executar-se troba la variable a `true`, salta el `while` i escriu el missatge. Després torna a posar la variable booleana a `false` per les execucions següents.

El resultat seria semblant a:

```
run:
 Hola  Adéu  Hola  Adéu  Hola  Adéu  Hola  Adéu  Hola  Adéu
BUILD SUCCESSFUL (total time: 5 seconds)
```

Una altra manera de solucionar el mateix problema és utilitzant un **monitor**. Un monitor no és més que una classe, una col·lecció de mètodes i dades, que s'utilitza com a eina de sincronització amb la característica que si un fil agafa el control del monitor, cap altre fil pot agafar-lo fins que el primer no el deixi.

Els fils poden cridar els mètodes visibles del monitor, però no poden accedir directament a les estructures de dades internes del monitor.

Bàsicament, un monitor ha de tenir: un mètode que enviï els fils a la cua d'espera (`wait()`) depenent d'una condició; un altre mètode que rescati els fils de la cua d'espera (`notify()`) i canviï el valor de la condició; i un constructor que inicialitzi la variable de la condició.

La classe `escriuHolaAdeu()` de l'exemple anterior podria ser el monitor de l'aplicació.

2.4.2 Productor-consumidor

Un exemple clàssic de programació concurrent i multifils és el model de productor-consumidor. Es tracta d'un fil (productor)que genera una sortida que és agafada per un altre fil (consumidor).

Imaginem un sistema en el qual un fil (productor) va generant números aleatoris i un altre fil (consumidor) els va agafant per mostrar-los.

Ens hi trobem diferents classes. La classe `monitor` serà el nostre recipient de números. Hi hauran dos mètodes: `agafar()` que ens retornarà el número generat pel productor i `deixar(int)` que deixarà el número a disposició del consumidor. Per tant, la classe monitor realitzarà el control de transacció i comunicació entre els dos fils: productor i consumidor.

```
class Monitor {

    private int numero = 0;
    void Monitor(){
```

```
        numero=0;
    }

    public synchronized int afagar() {
        return( numero );
    }

    public synchronized void deixar( int num ) {
        numero = num;
    }
 }
```

La classe `productor` s'encarregarà d'anar produint els números i de deixar-los a disposició del consumidor del monitor.

```
class Productor implements Runnable {
    private Monitor safata;

    public Productor( Monitor s ) {
        safata = s;
     }
    private int numero =0;

    public void run() {

        // Posa 10 lletres a la canonada
        for( int i=0; i < 5; i++ ) {
                numero= (int)(10*Math.random());
                safata.deixar(numero);
                System.out.println( "Produït el número "+numero );
                try {
                    Thread.sleep( (int)(Math.random() * 1000 ) );
                } catch( InterruptedException e ) {;}
            }
        }
    }
```

La classe `consumidor` s'encarregarà d'anar consumint els números del monitor.

```
class Consumidor implements Runnable {

    private Monitor safata;

    public Consumidor( Monitor s ) {
        safata = s;
     }


    public void run() {
        int num;

        for( int i=0; i < 5; i++ ){
                num = safata.afagar();
                System.out.println( "Número agafat: "+num );
                try {
                    Thread.sleep( (int)(Math.random() * 2000 ) );
                } catch( InterruptedException e ) {;}
            }
        }
    }
```

La classe principal crea dos fils: un productor i un consumidor i els executa.

```
package productor.consumidor;

public class ProductorConsumidor {
```

```
4
5   public static void main(String[] args) throws InterruptedException {
6        Monitor saf = new Monitor();
7
8       Productor p = new Productor(saf);
9       Consumidor c = new Consumidor(saf);
10      Thread productor = new Thread(p);
11      Thread consumidor = new Thread(c);
12
13      productor.start();
14      consumidor.start();
15
16
17  }
18
19 }
```

El resultat de l'execució del codi anterior és:

```
1  run:
2  Produït el número 4
3  Número agafat: 4
4  Produït el número 3
5  Número agafat: 3
6  Número agafat: 3  // El consumidor ha anat més ràpid que el productor i ha
       tornat a agafar el número 3
7  Produït el número 2
8  Produït el número 5
9  Número agafat: 5
10 Produït el número 2
11 Número agafat: 2
12 BUILD SUCCESSFUL (total time: 5 seconds)
```

Hem obviat part de codi per poder veure els possibles errors. Tal com es veu en el resultat, és un petit desastre. El model planteja alguns problemes. Ara no hi ha comunicació entre fils, per tant la producció i el consum de números és arbitrari, depèn de la velocitat del productor i del consumidors (que es troba simulada per `sleep()`). Si el consumidor és més ràpid que el productor, pot passar que agafi el mateix número diversos cops i si el productor és més ràpid pot sobreescriure el número i el consumidor mai el podrà agafar. Una solució al problema d'un productor més ràpid que un consumidor és col·locar un *buffer*, una cua que permeti guardar més d'un número.

Una cua LIFO és una estructura de dades en forma de cua en la qual l'element que entra a l'estructura en últim lloc és el primer en sortir-ne (*Last In Firts Out* en anglès). Aquest tipus de cua també s'anomena *pila*.

Crearem una cua tipus LIFO per anar guardant els números produïts pel `productor` i solucionarem la comunicació entre fils. Per fer això hem de modificar la classe `monitor`.

```
1  class Monitor {
2
3      private int cuaNumeros[] = new int[6];
4      private int segNum = 0;
5
6      private boolean cuaPlena = false;
7      private boolean cuaBuida = true;
8
9
10
11     public synchronized int afagar() {
12
13         // Si la cua es troba buida envia el fil a la cua d'espera
14          while( cuaBuida  ) //surt quan cuaBuida és false
15             {
16             try {
```

```
                wait();
            } catch( InterruptedException e ) {

                }
            }
        // Disminueix segNum perquè agafa un número i hi haurà un número menys
            a la cua
        segNum--;
        // Si no queden números
        if( segNum == 0 )
            cuaBuida = true; // La cua es troba buida
        cuaPlena = false;// La cua no està plena perquè hem consumit un número

        notify();

        // Retorna el número al fil consumidor
        return( cuaNumeros[segNum] );

    }

    public synchronized void deixar( int num ) {
       // Si la cua està plena, no hi entren més números, enviem el fil a la
           cua d'espera
        while( cuaPlena ) //Si cuaPlena és false, podrà continuar produint
            {
            try {
                wait();
            } catch( InterruptedException e ) {

                }
            }
        // afegim el nou número a la cua al primer lloc disponible
        cuaNumeros[segNum] = num;
        // augmentem els números a la cua
        segNum++;
        // Mirem si la cua està plena
        if( segNum == 6 )
            cuaPlena = true;
        cuaBuida = false; //és false perquè acaben de produir un número
        notify();
        }
    }
```

Ara, al monitor, controlem quan el consumidor vol agafar un número si hi ha números disponibles i si no n'hi ha enviem el fil a l'estat d'**en espera**. En cas contrari consumim el número, l'eliminem de la cua i despertem el fils productors que puguin estar en espera.

És possible que hi hagi productors en espera perquè la cua de números està plena.

Quan produïm també controlem si la cua de números és plena. Si es troba plena, no entren més números i, per tant, enviem a l'estat d'espera al fil productor fins que un fil consumidor el desperti i el posi en estat de preparat perquè ja ha consumit un número i queda espai a la cua. Finalment desperta els fils consumidors que puguin estar en espera ja que la cua es troba buida.

Un resultat possible és el següent:

```
run:
Produït el número 3
Número agafat: 3
Produït el número 1
Produït el número 5 //produeix dos números seguits
Número agafat: 5 // que són consumits en ordre LIFO aquí
```

```
Número agafat: 1
Produït el número 6
Número agafat: 6
Produït el número 3
Produït el número 0
Produït el número 9
Número agafat: 9
Número agafat: 0
Produït el número 7
Produït el número 7
Produït el número 3
Número agafat: 3
Número agafat: 7
Número agafat: 7
Número agafat: 3
BUILD SUCCESSFUL (total time: 9 seconds)
```

Si ens fixem en l'ordre, s'han produït diversos números seguits sense que s'hagin consumit. Quan el consumidor actua, agafa el número que més tard ha entrat a la cua de números i els va extraient en aquest ordre.

2.5 Agrupació de fils

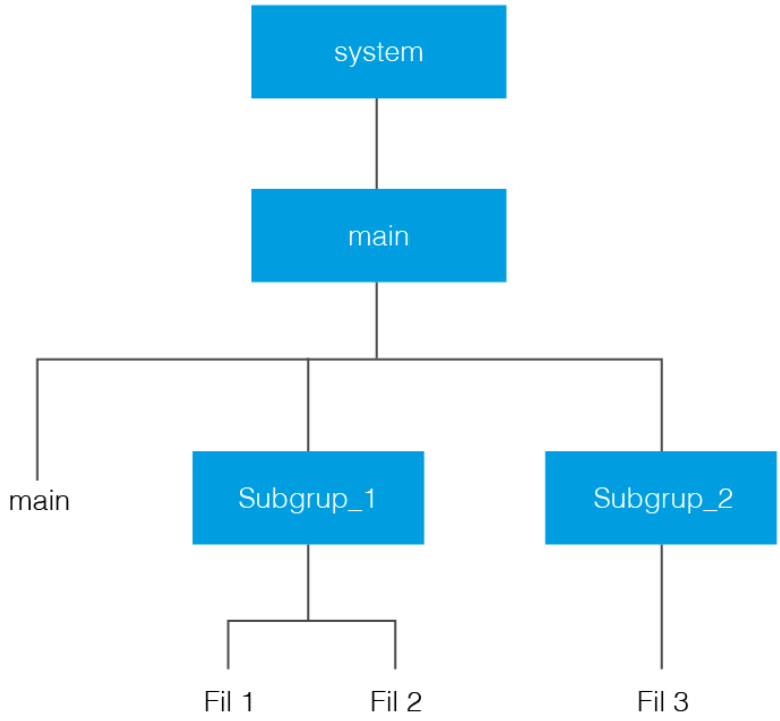
En un entorn de xarxa, un servidor rep moltes peticions a la vegada: de programes en execució, de clients, càlculs complexes, transaccions de base de dades, etc. Un fil acostuma a crear fils nous per atendre aquestes peticions. Si el número de peticions és gran, el número de fils també ho és i l'administració dels fils és complicada. Per simplificar aquesta administració es creen grups de fils. La classe `ThreadGroup` és la que defineix i implementa tot allò relacionat amb grups de fils.

2.5.1 Agrupació per defecte i creació de grups

Tots els fils d'execució a Java pertanyen a un grup. Els grups de fils són un mecanisme que ens permet agrupar fils a un únic objecte i ens permet la manipulació de tots el fils que pertanyen al grup com si fos un i no de forma individual. Podríem arrancar o adormir tots els fils d'un grup, aplicant el mètode corresponen a l'objecte grup.

En la figura 2.8 podem veure com existeix com a mínim un fil d'execució principal creat pel mètode `main` i després els diferents subgrups creats per l'aplicació. Cada grup té un número de fils. El `subgrup_1` té dos fils (`Fil1` i `Fil2`) i el `subgrup_2` un fil (`Fil3`).

Figura 2.8. Grups de fils



La classe `ThreadGroup` ens proporciona dos constructors:

- `ThreadGroup(String name)`
- `ThreadGroup(ThreadGroup parent, String name)`

Tots els fils tenen un grup i tots els grups creats a l'aplicació tenen un grup pare. Els dos constructors creen grups i els donen un nom. La diferència és l'elecció de a quin subgrup pertanyen.

El primer constructor pertany al mateix grup que el fil que fa li fa la crida. El segon constructor indica a quin pare pertany. Per exemple, si el fil principal fa una crida a `ThreadGroup(String name)` , el nou grup de fils té com a grup pare el `main`.

El exemple següent mostra com s'utilitzen els dos constructors. És crea un subgrup com a grup pare el `main` amb el nom `subGrup_1` i posteriorment es crea un altre grup que té com a pare el grup que acaben de crear:

```
1 public static void main (String [] args) {
2 ThreadGroup sGrupFils1 = new ThreadGroup ("subGrup_1");
3 ThreadGroup sGrupFils1_2 = new ThreadGroup (sGrupFils1, "subGrup_1_2");
4 }
```

La creació de grups de fils no té gaire utilitat si no li assignem fils al grup.

En el següent codi es mostra com es crea un grup anomenat subgrup 2 i posteriorment com es crea un fil (Fil 1) i l'assignem al grup.

```
1 ThreadGroup sgr = new ThreadGroup ("subgrup 2");
2 Thread fil = new Thread (sgr, "Fil 1");
```

2.5.2 Classe ThreadGroup

`ThreadGroup` conté tota la funcionalitat per la manipulació de grups de fils. Un grup pot tenir un número de fils il·limitat. Habitualment tots el fils d'un grup tenen alguna cosa en comú: qui els va crear, quina funció tenen o quan han d'executar-se o finalitzar. En el següent codi creem un grup (`subGrup 1`) que té com a pare `main`. Després creem dos fils i li diem que pertanyen al grup creat. Creem un altre grup (`subgrup 2`) i creem un fil (`Fil 3`) que pertany al nou grup. És l'esquema de la figura 2.8.

El mètode `activeGroupCount()` ens diu quants grups hi ha actius i `currentThread().getThreadGroup()`, com es diu el grup del fil actiu. `list()` és un mètode que dóna els detalls del grup actiu i els seus subgrups (nom, prioritat...)

```
1  public class GrupsFils {
2
3      public static void main(String[] args) {
4   ThreadGroup sg = new ThreadGroup ("subgrup 1");
5       Thread fil1 = new Thread (sg, "Fil 1");
6       Thread fil2 = new Thread (sg, "Fil 2");
7       sg = new ThreadGroup ("subgrup 2");
8       Thread fil3 = new Thread (sg, "Fil 1_1");
9       sg = Thread.currentThread ().getThreadGroup ();
10      int agc = sg.activeGroupCount ();
11      System.out.println ("Grup de fil actiu " + sg.getName () +
12                          ". Grups actius: " + agc);
13      sg.list ();
14
15  }
16 }
```

El codi anterior té el següent resultat:

```
1  run:
2  Grup de fil actiu main. Grups actius: 2
3  java.lang.ThreadGroup[name=main,maxpri=10]
4      Thread[main,5,main]
5      java.lang.ThreadGroup[name=subgrup 1,maxpri=10]
6      java.lang.ThreadGroup[name=subgrup 2,maxpri=10]
7  BUILD SUCCESSFUL (total time: 1 second)
```

La prioritat d'un grup és la prioritat més alta dels fils que pertanyen al grup.

Els mètodes següents de la classe `ThreadGroup` únicament tenen efecte sobre el grup, però no sobre els fils del grup:

- `getMaxPriority`: retorna la prioritat del grup.
- `setMaxPriority`: modifica la prioritat del grup, no la dels fils. Per tant, si disminuïm la prioritat del grup, pot ser que hi hagi un fil amb una prioritat més alta que el grup.
- `getDaemon` i `SetDaemon`: retorna si és un grup dimoni i assigna a un grup de fils si es comportarà com a dimoni o no.

- `getName`: retorna el nom del grup.
- `getParent`: retorna el pare d'un grup.
- `parentOf`: ens diu si el grup de fil pertany al grup que li passem per paràmetres.
- `toString`: converteix en *string* els detalls del grup (nom, prioritat, etc).

2.6 Gestió de fils per part del sistema operatiu

La màquina virtual de Java (JVM, *Java Virtual Machine* en anglès) és un sistema multifil, i és l'encarregada de gestionar els fils, d'assignar el temps d'execució, les prioritats, etc. És l'encarregada de decidir quin fil entra a executar-se. La gestió que fa dels fils és similar a la que té el sistema operatiu quan gestiona més d'un procés. La màquina virtual de Java és un procés dins del sistema operatiu, és a dir, ja que el fils s'executen a la màquina virtual, comparteixen recursos.

2.6.1 Planificació de fils

La manera que té el sistema operatiu de comportar-se amb l'execució de fils és un tema important en la programació multifil. La planificació fa referència a quina política s'ha de seguir per decidir quin fil agafa el control del processador i en quin moment. També s'ha de planificar quan ha de deixar d'executar-se. Java està dissenyat per tal que sigui un sistema portable. El sistema de fils ha de ser suportat per qualsevol plataforma i, ja que cada plataforma funciona de forma diferent, el tractament dels fils ha de ser molt general. Com a característiques més importants de la planificació de fils tenim:

- Tots els fils tenen assignada una prioritat i l'encarregat de decidir quin fil s'executa ha de garantir que els fils amb prioritat més alta tinguin preferència. Però això no implica que en un moment donat un fil amb prioritat més baixa estigui executant-se.
- S'ha de garantir que tots els fils s'executin en algun moment.
- El temps assignat a cada fil per fer ús del processador pot aplicar-se o no depenent del sistema operatiu en el qual s'executa la màquina virtual.

Amb aquestes premisses el programador ha de tenir en compte a l'hora de programar que els fils poden intercalar la seva execució en qualsevol moment. Si volem una ordenació en l'execució de fils, hem d'afegir codi a la nostra programació per tal que passi.

2.6.2 Prioritat de fils

Les prioritats dels fils a Java són inicialment la mateixa que els fils creadors. Per defecte tenen una prioritat de 5. El rang de prioritats va des d'1, definida amb la propietat de `Thread MIN_PRIORITY` a 10 definida per `MAX_PRIORITY`. La prioritat per defecte amb valor 5 és `NORM_PRIORITY`.

Els fils amb una prioritat més alta seran posats a execució per part del planificador en primer lloc i els de prioritat més baixa quan els de prioritat superior estiguin bloquejats o hagin acabat l'execució. La classe `Thread` té dos mètodes que permeten veure i modificar les prioritats dels fils:

- `setPriority()` permet modificar la prioritat d'un fil:

```
1 fil.setPriority(8);
```

- `getPriority()` torna la prioritat d'un fil:

```
1 System.out.println"(Prioritat fil " +  fil.getPriority());
```

Swing és una API de Java que proporciona la interfície gràfica d'usuari a Java. És l'API que substitueix a l'altre API gràfica (AWT) que únicament permet una aparença gràfica igual al sistema nadiu de la plataforma en la qual s'executa. En canvi Swing també permet una aparença que no té res a veure amb la plataforma nativa.

Els fils de les classes que generen interfícies d'usuari (Swing i AWT) tenen per defecte una prioritat de 6. D'aquesta manera la interacció amb l'usuari és més fluïda.

Si no hi ha cap esdeveniment d'entrada els fils estan a l'espera. Així els altres fils no són perjudicats per ells. Quan hi ha un esdeveniment d'entrada per ratolí o teclat els fils responen immendiatament.

El mètode `yield()` fa que un fil que s'està executant passi de nou a l'estat i a la cua de preparats. D'aquesta manera permet que altres fils de prioritat superior o de la mateixa prioritat es puguin executar. Però pot passar que un cop passat a preparat torni a ser escollit per executar-se de nou.

En el següent codi d'exemple es mostra com creem dos fils que competiran per qui acaba primer la seva execució. En condicions normals, amb la mateixa prioritat, el guanyador s'aniria alternant, depenent de quin escollís l'organitzador de la cua de fils. Però al modificar les prioritats dels fils, al primer li assignem la prioritat més baixa i al segon la més alta. D'aquesta manera ens assegurem que sempre guanyi el segon. Això farà que el segon acapari el processador fins que acabi la seva execució.

Els resultats també dependran del sistema operatiu sobre el qual s'executa la màquina virtual, ja que les prioritats poden variar d'un sistema a un altre.

Utilitzem el mètode `setName()` de la classe `Thread` per assignar-li un nom a cada fil.

```java
package prioritatsfils;

public class PrioritatsFils implements Runnable {

String strImprimir;
    public PrioritatsFils(String strP) {
        strImprimir=strP;
    }

    public void run(){
        for(int x=0;x<100;x++){
            System.out.println(strImprimir);
        }

    }

    public static void main(String[] args) {
        PrioritatsFils objRunnable1 = new PrioritatsFils("Corredor 1");
        PrioritatsFils objRunnable2 = new PrioritatsFils("Corredor 1");

        //Creem el fil amb l'objecte runnable
        Thread primer = new Thread(objRunnable1);
        //Assignem un nom al fil primer
        primer.setName("Corredor 1");

        Thread segon = new Thread(objRunnable2);
        //Assignem un nom al fil primer
        segon.setName("Corredor 1");
        //Canviem la prioritat de fil primer i li posem les més baixa
        primer.setPriority(Thread.MIN_PRIORITY);

        //Mostrem el nom del fil i la seva prioritat
        System.out.println("Prioritat " + primer.getName()+ ": " + primer.
            getPriority());

        //Canviem la prioritat de fil segon i li posem les més altes
        segon.setPriority(Thread.MAX_PRIORITY);
        System.out.println("Prioritat " + segon.getName()+ ": " + segon.
            getPriority());

        // Cridem al mètode start() per tal que comenci per posar-los a
            preparats
        primer.start();
        segon.start();

        System.out.println("Final Fil Principal");
            }
}
```

2.6.3 Fils dimonis

Quan creem un fil i abans de cridar el mètode `start()` podem indicar que el fil serà executat com un dimoni (*daemon* en anglès). Això permetrà al fil executar-se a un segon pla. Els fils dimonis tenen la prioritat més baixa. Si un fil dimoni crea un altre fil, aquest també serà un dimoni. Aquesta qualitat s'hereta. I no es pot canviar un cop el fil ha estat inicialitzat. Es troba a l'estat de preparat.

Garbage collector és un dimoni creat per la màquina virtual de Java, que s'encarrega d'alliberar memòria ocupada per objectes que no estan referenciats.

Per indicar que un fil és un dimoni s'utilitza el mètode `setDaemon(true)`. En el cas de `setDaemon(false)`, el fil és d'usuari, l'opció per defecte. Els fils dimonis finalitzen quan finalitza l'últim fil d'usuari i l'aplicació acaba.

```
fil.setDaemon(true);
```

Per veure si un fil és un dimoni, s'utilitza el mètode `isDaemon()` que retornarà un booleà indicant si el fil és un dimoni o no.

```
System.out.println(fil.isDaemon());
```

El següent exemple mostra com es crea un dimoni. I després com, des d'aquest dimoni, es crea un *array* de 10 dimonis. Hem de veure que els fils dimonis creats pel dimoni inicial no els indiquem amb `setDaemon(true)`, però hereten aquesta qualitat del fil que l'ha creat.

Si volguéssim crear fils que no fossin dimoni, hauríem d'indicar-ho expressament amb `setDaemon(false)`, abans del mètode `start()` del fil.

```
package filsdimonis;
import java.io.IOException;

public class FilsDimonis extends Thread{
  private static  final int MIDA = 10;
  private Thread[] fils = new Thread[MIDA];
  public FilsDimonis(){
    setDaemon(true);
    start();
  }
  public void run(){
    for(int i = 0; i<MIDA; i++){
      fils[i]= new CrearDimoni(i);
    }
    for(int i=0; i<MIDA; i++){
    System.out.println("fils[" + i + "].isDaemon() = " + fils[i].isDaemon() );
    }
    while(true) yield();
  }

class CrearDimoni extends Thread{
  public CrearDimoni(int i){
    //si un fil no ens interessa que sigui dimoni, ho hem d'indicar aquí
    //setDaemon(false); abans del mètode start()
    //en cas contrari, hereta la qualitat de dimoni del seu pare.

    System.out.println("Dimoni creat " + i );
    start();
  }
  public void run(){
    while (true) yield();
  }
}

public static void main(String[] args) throws IOException{
    Thread fil = new FilsDimonis();
    System.out.println("isDaemon() = " +  fil.isDaemon() );
    System.out.println("Pica una tecla per finalitzar");
    System.in.read();
  }
    }
```

2.7 Documentació d'aplicacions i depuració d'aplicacions multifils

Una documentació eficient soluciona molts problemes a l'hora de mantenir l'aplicació, ja que les persones encarregades del manteniment, millores o ampliacions poden no ser les mateixes que la van construir. O fins i tot poden anar canviant en cada una de les fases de la creació de l'aplicació.

Un programador ha d'intentar comentar el seu codi per tal que sigui el màxim d'entenedor per als altres programadors.

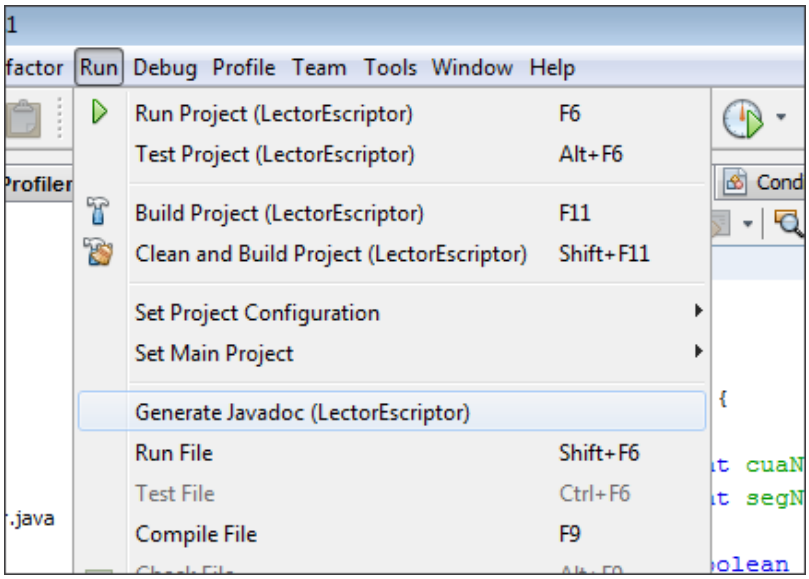
Java ens proporciona una eina per crear documentació tècnica a nivell d'implementació, s'anomena **JavaDoc**. JavaDoc analitza el codi i els comentaris sobre el codi i genera una documentació que pot ser en HTML amb un llistat de les classes i mètodes. Fins i tot indica si hem utilitzat algun mètode obsolet.

És molt important, doncs, que els codi estigui comentat amb /* */.

A NetBeans per generar la documentació amb JavaDoc hem d'anar al menú *Executar* i clicar sobre l'opció *Generar Javadoc* (figura 2.9).

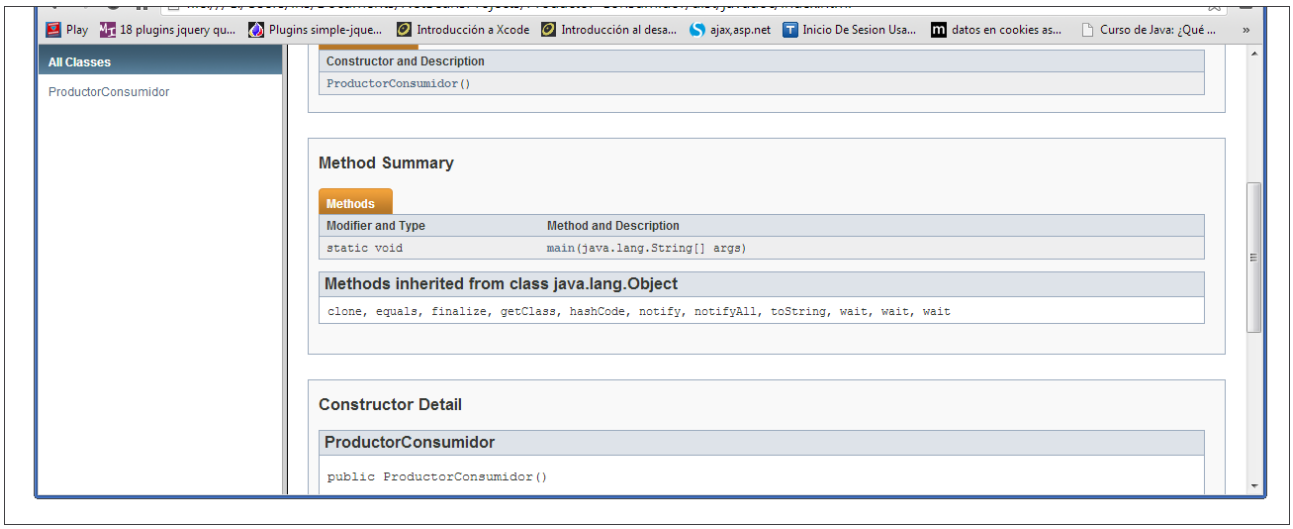
NetBeans és entorn de desenvolupament integrat (IDE, *Integrated Development Environment*, en anglès). Pensat per desenvolupar aplicacions, va ser creat en Java per Sun MicroSystems i és lliure i gratuït.

FIGURA 2.9. Crear javadoc



El resultat és un document HTML (figura 2.10).

FIGURA 2.10. Document creat per javadoc



2.7.1 Depuració d'aplicacions multifils

Depurar aplicacions és una de les tasques que més sovint fem els programadors. Ens permet verificar l'estat de la nostra aplicació en un punt en concret. Posar marques al codi o utilitzar les eines de depuració (*debugging*, en anglès) amb totes les utilitats que ens proporcionen alguns IDE, és una tasca imprescindible per revisar la nostra aplicació. Depurar una aplicació seqüencial en la qual únicament hi ha una línia d'execució és simple amb una eina de depuració. Anem avançant pas a pas al fil d'execució i anem mirant valors de variable, condicions... fins que acabem. Però en la programació multifil hi ha més d'un fil d'execució. Haurem d'anar consultant cada fil per separat per veure el seu funcionament i quins valors va prenent.

Veurem com podem depurar una aplicació multifil i utilitzarem l'eina per detectar possibles errors com l'interbloqueig a una aplicació.

Per fer les proves utilitzarem l'IDE de Netbeans. Amb altres IDEs el funcionament és similar. Per provar com consultar els fils obrirem un projecte i seleccionarem el projecte a la finestra de projectes i triarem l'opció *Debug*. El projecte que analitzarem és el següent:

```
1  package nauespaial;
2
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.Image;
7  import java.util.Random;
8  import javax.swing.ImageIcon;
9  import javax.swing.JFrame;
10 import javax.swing.JPanel;
11
12 /**
```

```
 *
 * @author jaleo
 */
public class NauEspaial  extends javax.swing.JFrame {

    public NauEspaial() {
        initComponents();
    }


    @SuppressWarnings("unchecked")

    private void initComponents() {

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setBackground(new java.awt.Color(255, 255, 255));

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(
            getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
                LEADING)
            .addGap(0, 400, Short.MAX_VALUE)
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
                LEADING)
            .addGap(0, 300, Short.MAX_VALUE)
        );

        pack();
    }


    public static void main(String args[]) {

        try {
            for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.
                UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {
                    javax.swing.UIManager.setLookAndFeel(info.getClassName());

                    break;
                }
            }

        } catch (ClassNotFoundException ex) {
            java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                java.util.logging.Level.SEVERE, null, ex);
        } catch (InstantiationException ex) {
            java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                java.util.logging.Level.SEVERE, null, ex);
        } catch (IllegalAccessException ex) {
            java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                java.util.logging.Level.SEVERE, null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex) {
            java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                java.util.logging.Level.SEVERE, null, ex);
        }

           NauEspaial f = new NauEspaial();
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            f.setTitle("Naus Espaials");
            f.setContentPane(new PanelNau());
            f.setSize(480, 560);
            f.setVisible(true);

        }
    }
```

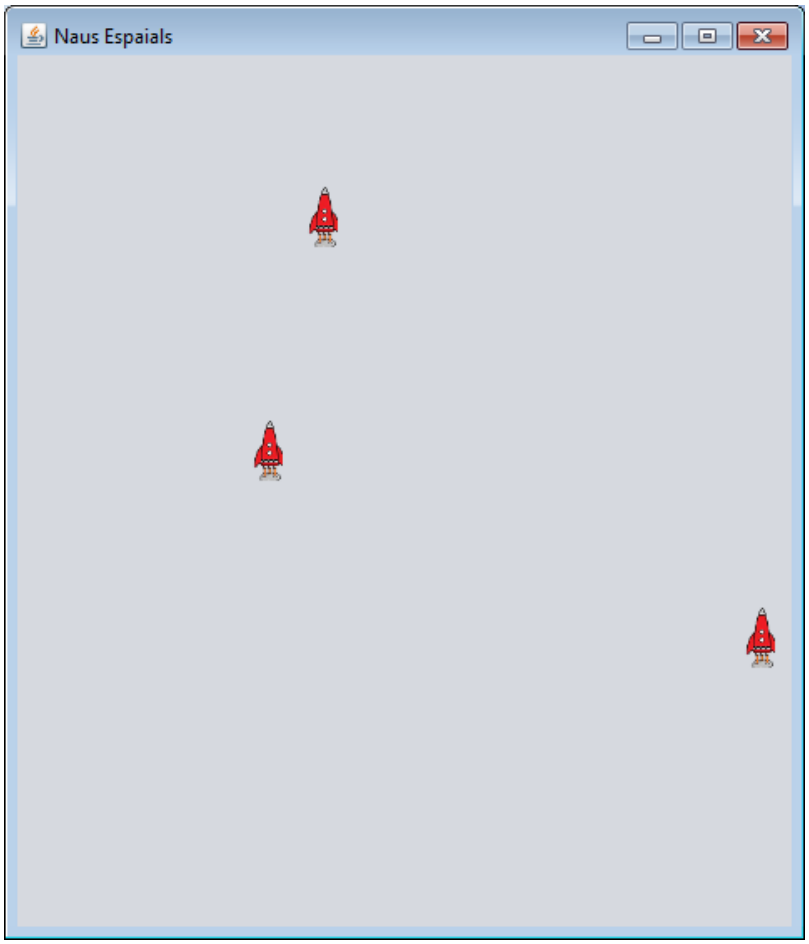
```
75
76
77
78  class PanelNau extends JPanel implements Runnable{
79
80      private int numNaus=3;
81      Thread[] filsNau;
82      Nau[] nau;
83
84
85
86      public PanelNau(){
87
88         filsNau = new Thread[numNaus]; //Creo dos hilos
89         nau = new Nau[numNaus];
90         for (int i=0;i<filsNau.length;i++){
91            filsNau[i]= new Thread(this);
92            filsNau[i].setName("Fil Nau—"+i);
93
94            Random rand = new Random();
95            int velocitat=rand.nextInt(50);
96            int posX=rand.nextInt(100)+30;
97            int posY=rand.nextInt(100)+30;
98            int dX=rand.nextInt(3)+1;
99            int dY=rand.nextInt(3)+1;
100           nau[i]= new Nau(posX,posY,dX,dY,velocitat);
101
102        }
103
104           for (int i=0; i<filsNau.length; ++i)
105              filsNau[i].start();
106 }
107
108
109 public void run(){
110      for (int i=0; i<filsNau.length; ++i){
111       while (filsNau[i].currentThread()== filsNau[i]) {
112             try {
113                 filsNau[i].sleep(nau[i].velocitat()); //
114                 nau[i].moure();
115              }catch (InterruptedException e) { }
116             repaint();
117         }
118
119       }
120 }
121
122
123   public void paintComponent(Graphics g) {
124       super.paintComponent(g);
125       for(int i=0; i<nau.length;++i)
126           nau[i].pinta(g);
127     }
128   }
129
130
131
132
133 class Nau {
134     private int x, y;
135     private int dsx, dsy, v; //desplaçaments, v sleep
136     private int tx = 10; //marge x
137     private int ty = 10; //marge i image
138
139     private String img = "/images/nau.jpg";
140     private Image image;
141
142
143
144       public Nau(int x, int y, int dsx, int dsy, int v ){
```

```
145             this.x=x;
146             this.y=y;
147             this.dsx= dsx;
148             this.dsy=dsy;
149             this.v=v;
150             image = new ImageIcon(Nau.class.getResource("/images/nau.png")).
                    getImage();
151
152         }
153
154         public int velocitat (){ //sleep en milisegons
155             return v;
156         }
157
158         public void moure (){
159           x=x + dsx;
160           y=y + dsy;
161         // arriva als marges
162           if ( x>= 450-tx || x<= tx)
163               dsx = -dsx;
164           if ( y >= 500- ty || y<=ty )
165              dsy = -dsy;
166
167         }
168         public void pinta (Graphics g){
169               Graphics2D g2d = (Graphics2D)g;
170               g2d.drawImage(this.image, x, y, null);
171         }
172
173
174 }
```

Els punts de ruptura (*breakpoints* en anglès) són marques que indiquen a l'IDE on cal aturar l'execució del codi quan es faci en mode depuració.

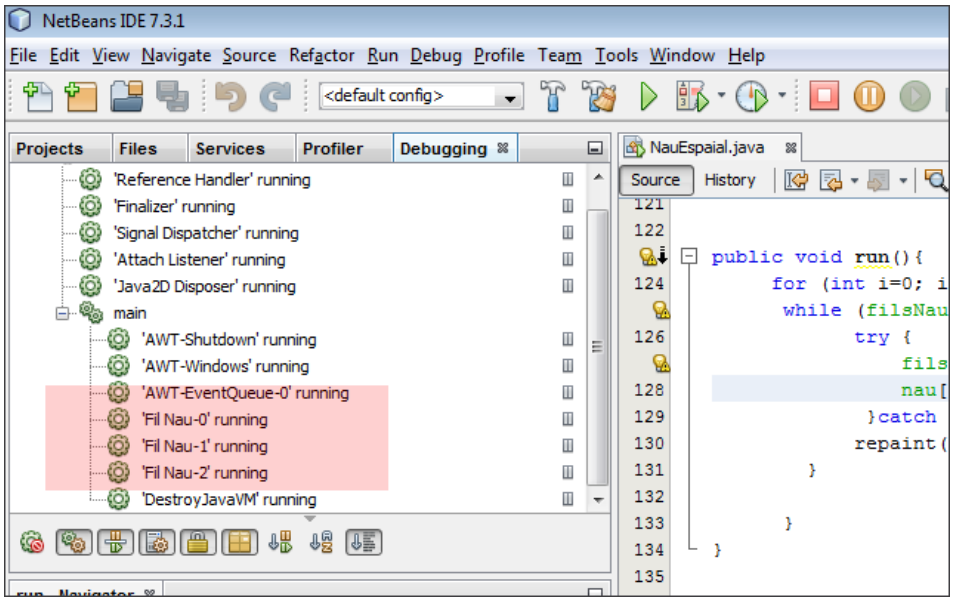
Si no tenim posats punts de ruptura l'aplicació s'executarà fins a acabar. L'aplicació és un panell en el qual es mostren unes imatges que es mouen. Cada imatge (nau espacial) és controlada per un fil. L'aplicació crea 3 fils, per tant hi haurà tres imatges al panell en moviment (figura 2.11).

FIGURA 2.11. Aplicació de naus en funcionament



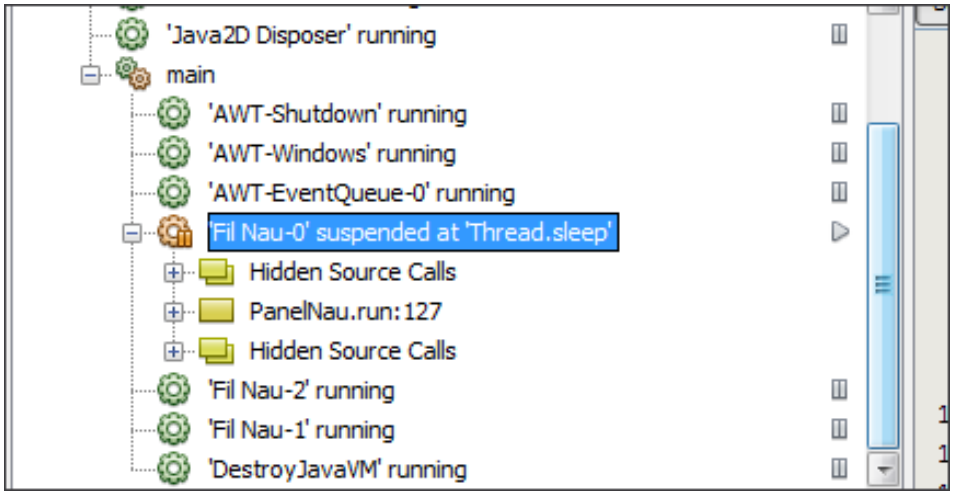
A la finestra de depuració podem veure els tres fils creats (figura 2.12), amb el nom de *Fil Nau 0*, *Fil Nau 1* i *Fil Nau 2*. La resta de fils creats són el fil principal i els del sistema.

FIGURA 2.12. Depuració de fils



A la dreta del nom del fil hi ha un símbol de pausa. Si el clquem el fil s'aturarà. El símbol es convertirà en el triangle de *play*. Quan el fil es troba aturat (figura 2.13) podem veure que surt una carpeta que es pot expandir. Dins hi veiem la llista de trucades del fil. Ja que el fil està aturat, la nau (la imatge que es mou pel panell) s'atura també.

Figura 2.13. Fil aturat

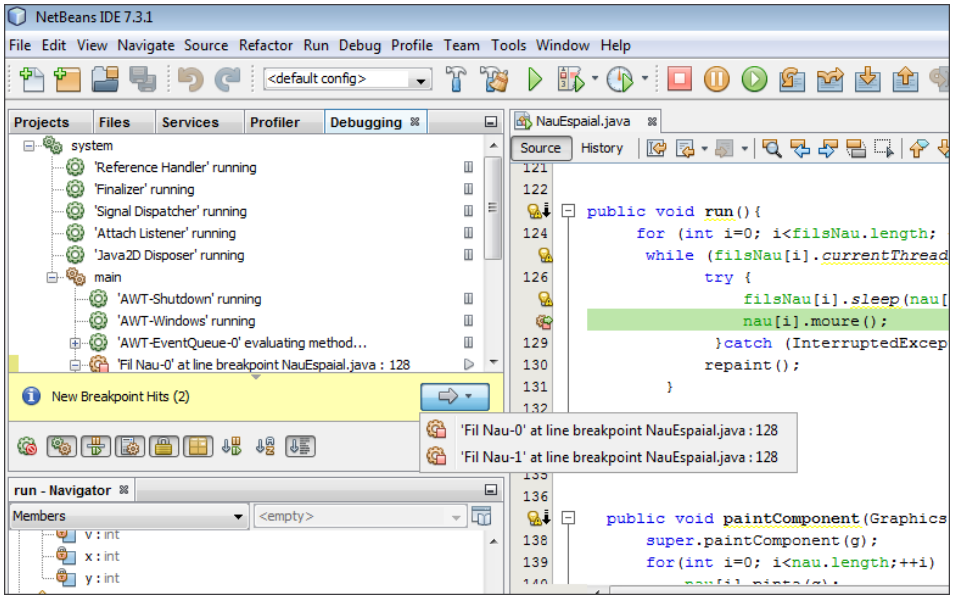


Si hem posat algun punt d'interrupció, l'aplicació es comporta com si no treballés amb fils. L'execució s'atura en aquest punt.

Per posar un punt d'interrupció hem de picar al marge esquerre de l'editor al costat del número de línia en el qual volem posar el punt d'interrupció. Apareixerà un quadrat vermell que marca el punt.

En la figura 2.14 es veu que el punt d'interrupció està posat quan el fil executa el moviment de la imatge.

Figura 2.14. Punt d'interrupció

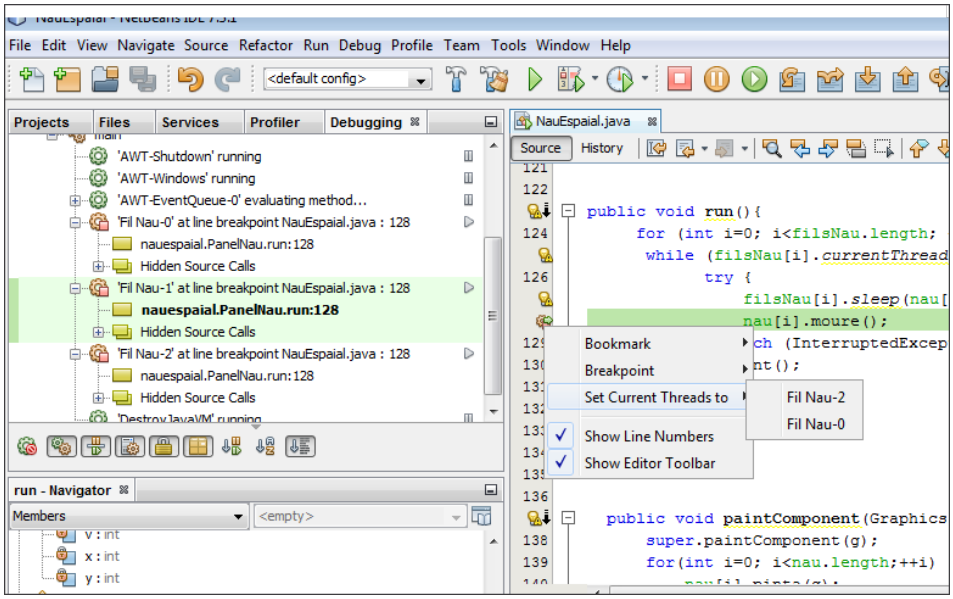


En la finestra de depuració es veu com s'està executant el primer fil que arriba. Clicant les tecles de funció F7 o F8 anirà avançant la línia de programa sobre el fil que està actiu. La resta de fils estan preparats per executar-se. Si volem que els altres fils comencin haurem de cliar sobre la fletxa que està sota la finestra

de depuració i seleccionar el fil que volem executar. Un cop tots els fils s'estan executant, podem veure l'execució d'un o altre fil picant sobre el fil a la finestra de depuració. El fil que estem controlant apareix amb un marc verd.

A l'esquerra del codi apareixen les icones de la roda dentada que ens indiquen on es troben aturats els fils que no estem depurant. Si piquem sobre la roda dentada amb al botó dret podem veure quin és el fil que està aturat en aquest punt (figura 2.15).

FIGURA 2.15. Intercanviant l'execució dels fils



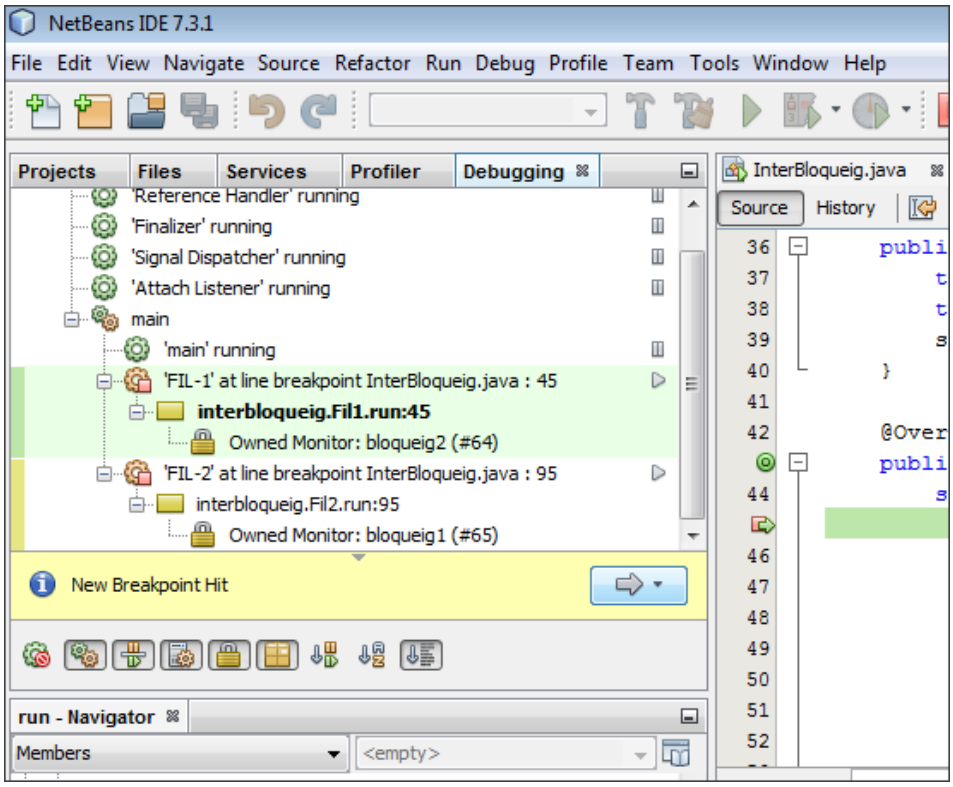
Si volem canviar per continuar depurant un altre fil el seleccionarem de la llista. Únicament podem depurar un fil amb un punt de ruptura (*breakpoint*, en anglès). Quan cliquem sobre F7 o F8 per anar avançant únicament avança un fil, els altres no s'aturen al punt d'interrupció.

Depurar aplicacions és bastant simple i molt útil. Els IDE cada vegada proporcionen eines que ens ajuden més a la correcció d'errors. Netbeans a la seva eina de depuració inclou una funció que detecta interbloquejos automàticament.

Per utilitzar-la hem de crear punts de ruptura. Els punt d'interrupció es posen on nosaltres pensem que hi ha el punt crític de la nostra aplicació i on podria ser la font dels errors. En aquest cas els posarem al mètode `run()` de la classe que hereta `threads` o implementa `runnable`.

Executem l'aplicació en mode depuració i veiem com s'atura als punt d'interrupció (figura 2.16).

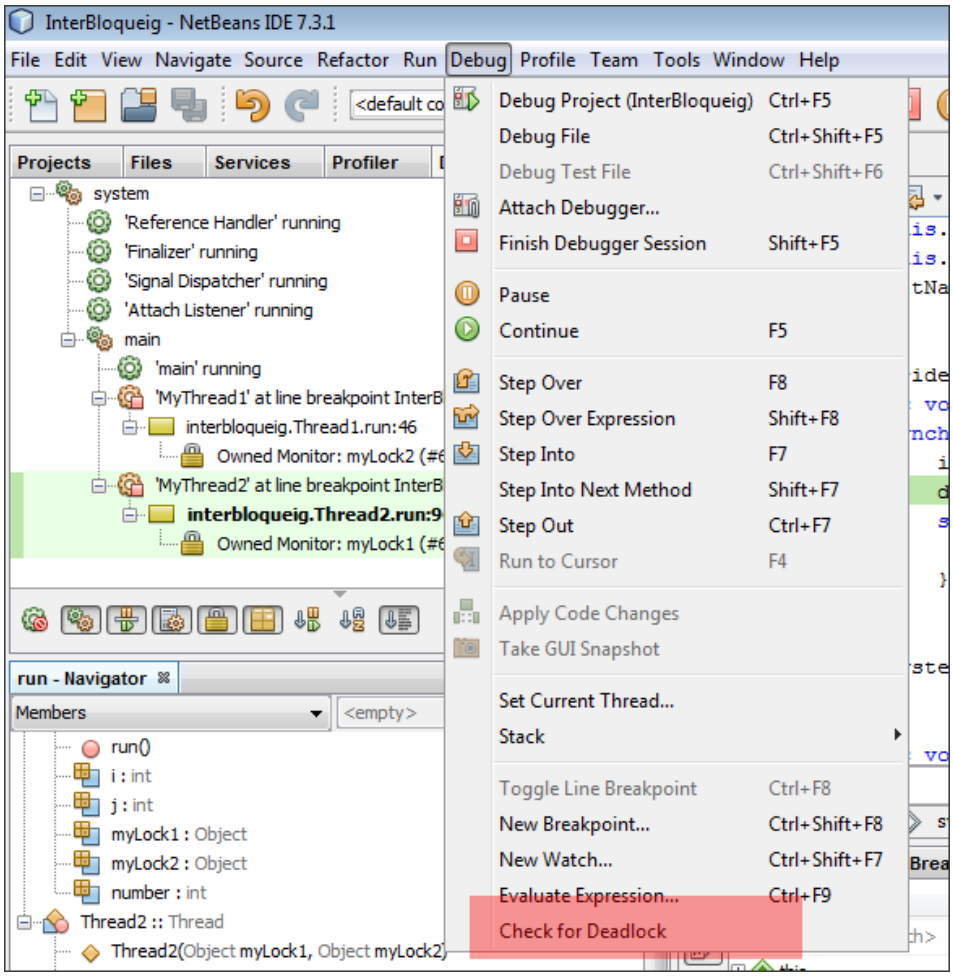
FIGURA 2.16. Punt d'interrupció



En aquest punt podem continuar l'execució (tecles de funció F7 o F8) i veure com avança l'aplicació i van variant els valors de les variables, la línia d'execució, etc.

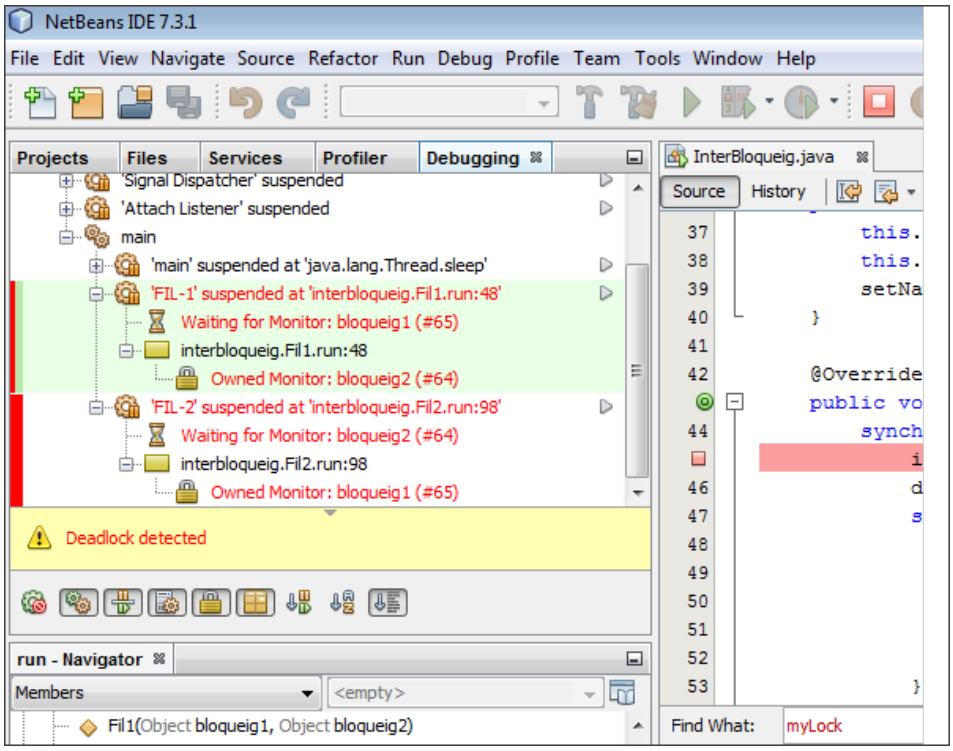
Per detectar l'interbloqueig hem d'acabar l'execució. Podem clicar sobre cada botó dret i seleccionar *resume*, podem clicar F5 o clicar sobre la icona de la barra d'eines del triangle blanc (*play*) sobre fons verd. El resultat que tindrem és l'aplicació executant-se. Per veure si s'ha produït un interbloqueig hem d'anar al menú de depuració i clicar sobre l'opció *Check for DeadLock* (figura 2.17).

FIGURA 2.17. Examinar interbloqueig



Automàticament Netbeans ens dirà si existeixen o no interbloquejos. Ens mostrarà un missatge dient si n'ha trobat o no (figura 2.18).

Figura 2.18. Missatge d'interbloqueig



La finestra de depuració ens diu on es troben els fils bloquejats en situació d'interbloqueig.

Les eines de depuració són molt útils per trobar possibles errades de programació. Quan programem utilitzant molts fils, la depuració de les aplicacions és fonamental.