
RA1. Aplicacions multiprocés i principis de programació paral·lela

Índex

1. Executables, processos i serveis	1
Eines de monitoratge de processos	1
2. Estats d'un procés i planificació	2
3. Gestió de processos en Linux	3
3.1. La funció <code>system()</code>	3
3.2. La família <code>exec()</code>	4
3.3. Creació de processos amb <code>fork()</code>	4
4. Comunicació entre processos	5
4.1. Pipes sense nom	6
4.2. Pipes amb nom (FIFO)	7
5. Sincronització entre processos	7
6. Creació de processos amb Java	9
6.1. Runtime i Process	9
6.2. Llegir la sortida i els errors del procés	9
6.3. ProcessBuilder	10
7. Fils d'execució	11
7.1. Creació de fils amb Java	11
8. Programació concurrent	12
8.1. Condicions de Bernstein	13
8.2. Problemes inherents a la concurrència	13
9. Programació paral·lela i distribuïda	14
9.1. Programació paral·lela	14
9.2. Programació distribuïda	14
9.3. Introducció a PVM (Parallel Virtual Machine)	15
9.4. Programació amb l'API de PVM	16
Índex de figures	19

Cicle: Desenvolupament d'aplicacions multiplataforma (DAM)

Mòdul: 0490. Programació de serveis i processos

1. Executables, processos i serveis

Un **programa** és un conjunt d'instruccions emmagatzemades en un fitxer executable. És una entitat *passiva*: no fa res fins que algú l'engega.

Un **procés** és un programa en execució. Quan el sistema operatiu carrega un executable a memòria, li assigna:

- Un **identificador únic (PID)**.
- Una còpia del codi, les dades i la pila (*stack*) del programa.
- Un joc de registres de CPU, incloent-hi el **comptador de programa**.
- Recursos: descriptors de fitxers oberts, dispositius assignats, memòria...

Tota aquesta informació es guarda en una estructura de dades anomenada **BCP (Bloc de Control de Procés)**, que inclou com a mínim:

Camp	Descripció
Identificació	PID únic del procés
Estat	En execució, bloquejat, llest...
Comptador de programa	Adreça de la següent instrucció
Registres de CPU	Context d'execució
Planificació	Prioritat i altres paràmetres
Gestió de memòria	Taules de pàgines, límits, etc.
Comptabilitat	Temps de CPU consumit
Estat d'E/S	Fitxers oberts, dispositius assignats

Un **servei** (o *dimoni*, *daemon*) és un procés especial que s'executa en segon pla, normalment sense terminal associat, oferint una funcionalitat contínua (un servidor web, un servidor de bases de dades, un dimoni de xarxa...). Un mateix programa pot generar diversos processos: per exemple, un navegador web sol tenir un procés que gestiona la interfície i un altre (o diversos) que fan les peticions al servidor.

Eines de monitoratge de processos

A **Linux**, l'ordre `ps` permet consultar els processos actius:

```
$ ps
  PID TTY          TIME CMD
 1906 pts/0    00:00:00 bash
 2362 pts/0    00:00:00 ps
```

- **PID**: identificador del procés.
- **TTY**: terminal associat (? si no en té).
- **TIME**: temps de CPU consumit.
- **CMD**: nom del procés.

Amb `ps -f` s'obté informació ampliada (usuari, PID del pare, hora d'inici) i amb `ps -AF` es mostren tots els processos actius amb el màxim de detall (mida virtual SZ, memòria resident RSS, processador assignat PSR, etc.). Gràficament, l'eina **Monitor del sistema** (gnome-system-monitor) ofereix la mateixa informació de forma visual.

A **Windows**, l'equivalent en línia d'ordres és `tasklist`:

```
D:\>tasklist
```

Nombre de imatge	PID	...	Uso de memoria
=====	===	===	=====
svchost.exe	1956		5.536 KB

I amb `tasklist /svc /fi "imagename eq svchost.exe"` es poden veure els serveis que corren sota cada procés `svchost.exe`. La interfície gràfica corresponent és l'**Administrador de tasques de Windows** ([Ctrl]+[Alt]+[Supressió]).

2. Estats d'un procés i planificació

En un sistema **multitasca** (o de multiprogramació) es poden tenir diversos processos "executant-se" alhora encara que només hi hagi una CPU: el sistema operatiu va alternant ràpidament l'assignació del processador entre ells mitjançant la planificació per temps compartit (time-sharing), creant la il·lusió de concurrència. Si el maquinari disposa de diversos nuclis o CPUs físiques, parlem pròpiament de multiprocés (paral·lelisme real).

Un procés pot trobar-se en tres estats principals:

- **En execució**: fa servir la CPU en aquest instant.
- **Bloquejat**: espera que succeeixi un esdeveniment extern (per exemple, que acabi una operació d'entrada/sortida) i no pot avançar fins que això passi.
- **Llest**: està preparat per executar-se, però de moment el processador l'ha "aparcats" per atendre'n un altre.

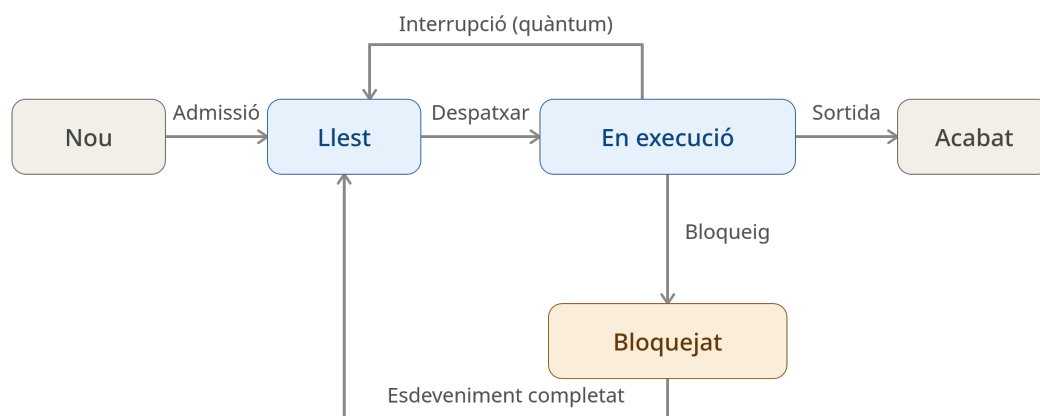


Figura 1: Estats d'un procés

Les transicions possibles són:

Transició	Motiu
En execució → Bloquejat	El procés espera un esdeveniment extern (p. ex. E/S)
Bloquejat → Llest	L'esdeveniment esperat ja ha ocorregut
Llest → En execució	El planificador li assigna temps de CPU
En execució → Llest	S'esgota el <i>quantum</i> de temps assignat

El component del sistema operatiu que decideix quin procés passa a “En execució” en cada moment és el **planificador de CPU**. Aquest fa servir criteris com la prioritat del procés, el temps que espera o el tipus de tasca (interactiva o de càlcul intensiu) per repartir el temps de processador de manera eficient i evitar que cap procés “mori de fam”.

3. Gestió de processos en Linux

Linux ofereix diverses funcions de la biblioteca estàndard per llançar i controlar processos des d'un programa en C.

3.1. La funció `system()`

Declarada a `<stdlib.h>`, executa una cadena de text com si s'hagués escrit a l'interpret d'ordres:

```
int system(const char *cadena);
```

Retorna -1 si la crida pròpia del sistema falla. Si la comanda s'ha executat, retorna un enter amb l'estat codificat de terminació (wait status), del qual podem extreure el codi de sortida real amb la macro `WEXITSTATUS(status)`.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    system("ls > llistat.txt");
    printf("Llistat generat.\n");
    return 0;
}
```

Precaució: no s'hauria de fer servir `system()` en programes amb privilegis d'administrador, perquè hereta les variables d'entorn de l'usuari i això podria comprometre la integritat del sistema. En aquests casos és preferible `exec()`.

3.2. La família exec ()

exec() (declarada a <unistd.h>) **substitueix** el codi del procés actual pel del programa indicat; si té èxit, mai retorna a la instrucció següent:

```
int execl(const char *pathname, const char *arg0, ... /*, (char *)
↪ NULL */);
```

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Els fitxers del directori són:\n");
    execl("/bin/ls", "ls", "-l", (char *) NULL);
    perror("Error en executar execl"); // Executat si execl falla!
    return 1;
}
```

3.3. Creació de processos amb fork ()

Per **crear** un procés nou (no substituir l'actual) es fa servir fork (), sense paràmetres:

```
#include <unistd.h>
pid_t fork(void);
```

fork () crea un **procés fill**, còpia exacta del codi i les dades del procés que l'ha cridat (el **procés pare**), amb la mateixa posició del comptador de programa. A partir d'aquest instant, pare i fill tenen memòries independents: modificar una variable en un no afecta l'altre.

El valor retornat permet distingir qui som:

Valor retornat	Situació
-1	Error en crear el fill
0	Estem al procés fill
> 0 (el PID del fill)	Estem al procés pare

Per obtenir identificadors de processos es fan servir:

```
pid_t getpid(void); // PID del procés actual
pid_t getppid(void); // PID del procés pare
```

Exemple: un pare crea un fill

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
```

```

int main(void) {
    pid_t pid, pid_fill;

    pid = fork();

    if (pid == -1) {
        // error
        printf("No s'ha pogut crear el procés fill...\n");
        exit(EXIT_FAILURE); // o exit(1);
    }
    if (pid == 0) {
        // Soc el fill
        printf("Sóc el procés fill\n\tEl meu PID és %d, el del meu pare és\n\t↪ %d\n",
            getpid(), getppid());
    } else {
        // Soc el pare
        pid_fill = wait(NULL); // espera que acabi el fill
        printf("Sóc el procés pare\n\tEl meu PID és %d\n\tEl meu fill %d\n\t↪ ha acabat\n",
            getpid(), pid_fill);
    }
    return 0;
}

```

La funció `wait(NULL)` bloqueja el procés pare fins que finalitzi el fill; és la manera més senzilla d'evitar processos *zombi* i de sincronitzar-se amb la finalització d'un fill.

Estructures avi-pare-fill: com que `fork()` es pot cridar dins d'un fill, és possible construir jerarquies de processos (avi → pare → fill), fent una segona crida a `fork()` dins de la branca `if (pid == 0)`.

4. Comunicació entre processos

Els mecanismes d'**IPC** (*Inter-Process Communication*) més habituals a Linux són els *pipes*, les cues de missatges, els semàfors i la memòria compartida. En aquest document ens centrem en els més senzills: **pipes** i **FIFO**.

4.1. Pipes sense nom

Un *pipe* és una mena de fitxer fals que connecta dos processos: el que escriu ho fa com si fos un fitxer de sortida, i el que llegeix ho fa com si fos un d'entrada. Si el *pipe* és buit, la lectura es bloqueja; si és ple, l'escriptura es bloqueja. El nucli s'encarrega de tota la sincronització.

```
#include <unistd.h>
int pipe(int fd[2]);
```

- `fd[0]` → descriptor de **lectura**.
- `fd[1]` → descriptor d'**escriptura**.

Com que `fork()` duplica els descriptors de fitxer del pare al fill, ambdós processos comparteixen inicialment els dos extrems del *pipe*. Cal **tancar l'extrem que no es fa servir** en cada procés, segons el sentit de la comunicació:

Sentit del flux	El pare tanca	El fill tanca
Pare → Fill	<code>fd[0]</code> (lectura)	<code>fd[1]</code> (escriptura)
Fill → Pare	<code>fd[1]</code> (escriptura)	<code>fd[0]</code> (lectura)

Exemple: el fill envia un missatge al pare

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int fd[2];
    char buffer[30];
    pid_t pid;

    pipe(fd);           // es crea el pipe
    pid = fork();       // es crea el procés fill

    if (pid == 0) {     // FILL
        close(fd[0]);   // no llegirem
        write(fd[1], "Hola pare", 10);
    } else {           // PARE
        close(fd[1]);   // 1. Tancar l'extrem no utilitzat
        ↪ IMMEDIATAMENT
        ssize_t bytes = read(fd[0], buffer, sizeof(buffer) - 1);
        if (bytes >= 0) buffer[bytes] = '\0';
        wait(NULL);    // 2. Esperar el fill després de
        ↪ consumir el pipe
        close(fd[0]);
        printf("Missatge rebut: %s\n", buffer);
    }
    return 0;
}
```

Quan cal comunicació **en tots dos sentits** (per exemple, un fill que rep una petició i respon), es fan servir **dos pipes**, un per a cada direcció.

4.2. Pipes amb nom (FIFO)

Els *pipes* anteriors només permeten comunicar processos **emparentats** (pare-fill). Els **FIFO** (*First In, First Out*) permeten comunicar **qualsevol parell de processos**, encara que no tinguin relació de parentiu, ja que existeixen com a fitxers amb nom dins del sistema de fitxers.

Es poden crear des de la línia d'ordres:

```
$ mknod FIF01 p
$ ls -l FIF01
prw-r--r-- 1 usuari usuari 0 gen 1 00:00 FIF01
```

o des d'un programa en C, amb la funció `mknod()`:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Per crear un FIFO cal indicar `S_IFIFO` combinat (amb OR bit a bit) amb els permisos desitjats:

```
mknod("FIF02", S_IFIFO | 0666, 0);
```

Un procés que **llegeix** d'un FIFO buit es queda esperant fins que un altre procés hi **escrigui**; la informació es consumeix en ordre d'arribada i no es pot tornar a llegir un cop llegida.

5. Sincronització entre processos

A més de comunicar-se, els processos sovint necessiten **coordinar-se** en el temps: que un no continuï fins que un altre hagi arribat a un cert punt. Un mecanisme senzill per fer-ho són els **senyals**.

Un **senyal** és un avís que un procés envia a un altre. Les funcions bàsiques són:

```
#include <signal.h>
#include <unistd.h> // sleep() i pause() pertanyen a
↳ unistd.h

void (*signal(int senyal, void (*func)(int)))(int); // instal·la un
↳ gestor
int kill(int pid, int senyal); // envia un senyal
int pause(void); // espera un senyal
unsigned int sleep(unsigned int segons); // pausa temporal
```

- `signal(SIGUSR1, gestor)` indica que, quan arribi el senyal SIGUSR1, s'executi la funció `gestor()`.

- `kill(pid, SIGUSR1)` envia el senyal SIGUSR1 al procés amb identificador `pid` (malgrat el nom, no necessàriament el mata: depèn del senyal).
- `pause()` bloqueja el procés fins que rep qualsevol senyal per al qual tingui un gestor instal·lat.

Exemple: el fill avisa el pare tres cops i després l'acomiada

En comptes d'un bucle simètric infinit, aquest exemple fa que sigui el **fill** qui porti la iniciativa: envia tres avisos SIGUSR1 espaiats en el temps (el pare només els compta) i, en acabat, li envia SIGTERM perquè finalitzi. Així es veu que el gestor de senyal es pot fer servir també per acumular informació (un comptador) i no només per imprimir un missatge:

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

volatile sig_atomic_t avisos_rebuts = 0;

void comptar_avis(int s) {
    avisos_rebuts++;
    // Segons POSIX, l'escriptura segura en senyals s'ha de fer amb
    ↪ write()
    // o simplement incrementar la bandera i imprimir al bucle del main:
    const char msg[] = "[PARE] Avis rebut\n";
    write(STDOUT_FILENO, msg, sizeof(msg) - 1);
}

int main(void) {
    pid_t ppid = getpid();
    signal(SIGUSR1, comptar_avis);

    pid_t fid = fork();
    if (fid == 0) {
        for (int n = 1; n <= 3; n++) {
            sleep(1);
            kill(ppid, SIGUSR1);
        }
        printf("[FILL] Tres avisos enviats. Acomiadant el pare amb
        ↪ SIGTERM...\n");
        kill(ppid, SIGTERM); // Ara sí envia SIGTERM com diu el text!
        exit(0);
    } else {
        // El pare espera en bucle; en rebre SIGTERM per defecte
        ↪ finalitzarà de cop
        while (1) {
            pause();
        }
    }
    return 0;
}
```

Amb SIGTERM sense gestor propi, el procés pare finalitza automàticament en rebre'l, així que no cal cap `kill` manual des d'un altre terminal. Si es volgués aturar abans, es pot

premer [Ctrl]+[C] o localitzar el PID amb `ps -fe | grep <nom> i fer kill <pid>.`

6. Creació de processos amb Java

Java, dins del paquet `java.lang`, ofereix les classes **Runtime** i **Process** per llançar processos del sistema operatiu des d'una aplicació Java.

6.1. Runtime i Process

```
static Runtime getRuntime();    // obté l'entorn d'execució de
↳ l'aplicació
Process exec(String ordre);    // executa una ordre en un procés a
↳ part
```

`exec()` pot llançar `SecurityException`, `IOException`, `NullPointerException` o `IllegalArgumentException`.

```
public class ObreBlocDeNotes {
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        try {
            r.exec("NOTEPAD");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Important: `exec()` no és un intèrpret d'ordres (*shell*); només executa un programa. Per llançar ordres internes de l'intèrpret de Windows (com `DIR`) cal invocar `CMD /C <ordre>`. A Linux, l'equivalent seria fer servir `/bin/sh -c "<ordre>"`.

6.2. Llegir la sortida i els errors del procés

La classe `Process` ofereix:

- `getInputStream()` → llegeix la sortida estàndard del procés llançat.
- `getErrorStream()` → llegeix els errors del procés llançat.
- `getOutputStream()` → escriu a l'entrada estàndard del procés llançat.
- `waitFor()` → espera que el procés acabi (retorna 0 si ha finalitzat correctament).

```
import java.io.*;

public class ExecutaComanda {
    public static void main(String[] args) throws Exception {
        Runtime r = Runtime.getRuntime();
```

```

        Process p = r.exec("CMD /C DIR");

        BufferedReader br = new BufferedReader(
            new InputStreamReader(p.getInputStream()));
        String linia;
        while ((linia = br.readLine()) != null)
            System.out.println(linia);
        br.close();

        int exitVal = p.waitFor();
        System.out.println("Valor de sortida: " + exitVal);
    }
}

```

Si volem enviar dades a l'entrada estàndard del procés (per exemple, respondre a una ordre interactiva), es fa servir `getOutputStream()`:

```

ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DATE");
Process p = pb.start();

// Enviem la resposta a l'entrada estàndard del procés:
try (OutputStream os = p.getOutputStream()) {
    os.write("02-12-2024\n".getBytes());
    os.flush();
}

```

AVÍS

Com que `exec()` no és un intèrpret d'ordres, redirigir la sortida amb `>` dins de la cadena de l'ordre **no funciona**; cal implementar la redirecció manualment amb `FileOutputStream/PrintWriter` a partir de la sortida llegida amb `getInputStream()`.

6.3. ProcessBuilder

Des de JDK 1.5, la classe **ProcessBuilder** (paquet `java.lang`) ofereix una forma més flexible de crear i configurar processos, gestionant atributs com l'ordre, l'entorn, el directori de treball i les redireccions d'entrada/sortida/error.

```

Process p = new ProcessBuilder("CMD", "/C", "DIR").start();

```

És equivalent a `Runtime.exec()`, però permet, entre altres coses:

- Consultar i modificar les variables d'entorn amb `environment()`.
- Redirigir la sortida i l'error directament a fitxers, sense passar per Java, amb `redirectOutput()` i `redirectError()` (des de JDK 1.7).
- Redirigir l'entrada des d'un fitxer amb `redirectInput()`.

```
import java.io.File;

public class ExempleProcessBuilder {
    public static void main(String[] args) throws Exception {
        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
        pb.redirectOutput(new File("sortida.txt"));
        pb.redirectError(new File("error.txt"));
        pb.start();
    }
}
```

7. Fils d'execució

Un **fil** (*thread*) és una seqüència de control **dins d'un mateix procés** que s'executa de forma independent d'altres fils del mateix procés. A diferència dels processos:

	Processos	Fils
Espai de memòria	Independent (generalment)	Compartit dins del mateix procés
Cost de canvi de context	Elevat	Baix
Comunicació	Mecanismes explícits del SO (pipes, FIFO, senyals...)	Variables compartides directament

Diversos fils d'un mateix procés comparteixen el context (codi, dades globals, fitxers oberts) però cadascun manté la seva pròpia pila d'execució i el seu comptador de programa.

7.1. Creació de fils amb Java

Java proporciona la classe **Thread** per a la programació concurrent basada en fils. Hi ha dues maneres de crear-ne:

a) Heretant de Thread

```
class ElMeuFil extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++)
            System.out.println("Dins del fil...");
    }
}

public class UsaFil {
    public static void main(String[] args) {
        ElMeuFil f = new ElMeuFil();
        f.start(); // engega el fil (invoca run()
                  ⇨ internament)
        for (int i = 0; i < 5; i++)
            System.out.println("Fora del fil...");
    }
}
```

b) Implementant la interfície Runnable

```
class ElMeuFil2 implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++)
            System.out.println("Dins del fil...");
    }
}

public class UsaFil2 {
    public static void main(String[] args) {
        ElMeuFil2 tasca = new ElMeuFil2();
        Thread t = new Thread(tasca);
        t.start();
        for (int i = 0; i < 5; i++)
            System.out.println("Fora del fil...");
    }
}
```

En tots dos casos, el mètode `run()` conté el codi que s'executarà de forma independent. Un cop `run()` acaba, el fil finalitza. La sortida per pantalla es va intercalant entre el fil principal i el fil creat, i **no és determinista**: pot variar cada vegada que s'executa el programa, precisament perquè el sistema operatiu decideix com repartir el temps de CPU entre tots dos.

NOTA

Es recomana la interfície `Runnable` quan la classe ja hereta d'una altra, ja que Java no permet herència múltiple.

8. Programació concurrent

La **concurrència** és l'existència simultània de diversos processos (o fils) en execució. En un sistema **monoprocessador**, la concurrència s'aconsegueix repartint el temps de CPU entre processos (**multiprogramació**): en cada instant només n'hi ha un en execució, però l'alternança és tan ràpida que fa la sensació de simultaneïtat. En un sistema **multiprocessador**, hi pot haver paral·lelisme real, amb un procés diferent executant-se físicament a cada processador (**multiprocés**).

Dos processos són **concurrents** quan hi ha encavalcament o intercalat en l'execució de les seves instruccions (la primera instrucció d'un s'executa després de la primera de l'altre i abans que acabi). Això no s'ha de confondre amb l'**execució simultània** pròpia de la **programació paral·lela**.

8.1. Condicions de Bernstein

Bernstein va formular unes condicions per determinar si dos conjunts d'instruccions es poden executar concurrentment sense risc. Per a cada conjunt d'instruccions es defineixen:

- **Conjunt de lectura L:** variables llegides.
- **Conjunt d'escriptura E:** variables escrites.

Dos conjunts d'instruccions I_i i I_j es poden executar concurrentment si es compleixen **totes tres** condicions:

$$L(I_i) \cap E(I_j) = \emptyset \quad E(I_i) \cap L(I_j) = \emptyset \quad E(I_i) \cap E(I_j) = \emptyset$$

És a dir: cap de les dues instruccions pot llegir el que l'altra escriu, ni ambdues poden escriure la mateixa variable.

Exemple:

Instrucció	Lectura L	Espectura E
I1: $x := y + 1$	y	x
I2: $y := x + 2$	x	y
I3: $z := a + b$	a, b	z

I1 i I2 **no** es poden executar concurrentment (la lectura d'una coincideix amb l'espectura de l'altra en ambdós sentits), mentre que I3 sí que és independent de les altres dues i es pot executar en paral·lel amb qualsevol d'elles.

8.2. Problemes inherents a la concurrència

Executar processos concurrentment introdueix dos problemes fonamentals:

1. **Exclusió mútua:** quan diversos processos accedeixen a la vegada a una variable compartida, cal evitar que un la llegeixi mentre un altre l'està modificant (inconsistència de dades). La solució clàssica és la **regió crítica**: el tram de codi que accedeix a la variable compartida, dins del qual només pot entrar-hi un procés cada vegada; la resta han d'esperar.
2. **Condicció de sincronització:** de vegades un procés no pot continuar fins que un altre hagi arribat a un cert punt de la seva execució (per exemple, fins que li arribin dades). Cal un mecanisme de bloqueig/desbloqueig coordinat amb l'esdeveniment esperat (com els senyals vistos al punt 5).

Per gestionar aquests problemes existeixen diverses eines, que s'estudiaran més endavant: regions crítiques, semàfors, regions crítiques condicionals, bústies, monitors i sincronització per *rendez-vous*.

9. Programació paral·lela i distribuïda

9.1. Programació paral·lela

Un **programa paral·lel** és un programa concurrent dissenyat perquè s'executi en un **sistema multiprocessador**, on diversos elements de procés treballen simultàniament per resoldre un mateix problema, dividit en parts independents.

Segons com s'intercanvia la informació entre processadors, hi ha dos models principals:

- **Memòria compartida:** tots els processadors accedeixen físicament al mateix espai d'adreces; un valor escrit per un és directament visible per als altres.
- **Pas de missatges:** cada processador té la seva pròpia memòria, inaccessible per als altres; l'intercanvi de dades requereix peticions i enviaments explícits. **PVM** es basa en aquest model.

Avantatges del processament paral·lel: execució simultània de tasques, reducció del temps total, capacitat de resoldre problemes de gran complexitat, aprofitament de recursos distribuïts en xarxa i reducció de costos respecte a un supercomputador dedicat.

Inconvenients: entorns de programació més difícils de desenvolupar i d'escriure, consum energètic més elevat, més complexitat en l'accés a dades i necessitat de mecanismes de comunicació i sincronització entre subtasques.

9.2. Programació distribuïda

Un **sistema distribuït** és aquell en què components de maquinari o programari, situats en ordinadors diferents units per una xarxa, es comuniquen i coordinen mitjançant el **pas de missatges**. Internet n'és l'exemple més conegut. Això comporta tres conseqüències importants:

- **Concurrencia:** en una xarxa d'ordinadors, l'execució concurrent de programes és la norma.
- **Absència de rellotge global:** la coordinació s'ha de fer via missatges, no per referència a un temps compartit.
- **Fallades independents:** cada component pot fallar sense que els altres deixin de funcionar.

L'arquitectura **client-servidor** és el patró més habitual: el client demana un servei, el servidor l'atén i respon. Els models de comunicació típics són:

- **Sockets:** la base de baix nivell de la comunicació en xarxa.
- **RPC (Remote Procedure Call):** permet cridar un procediment d'un programa remot com si fos local.
- **Invocació remota d'objectes (RMI):** en Java, Java RMI permet que un objecte en un procés invoqui mètodes d'un objecte que viu en un altre procés.

Avantatges: compartició de recursos i dades, creixement incremental, flexibilitat en el repartiment de càrrega, alta disponibilitat i caràcter obert i heterogeni.

Inconvenients: augment de la complexitat del programari, problemes propis de la xarxa (pèrdua de missatges, saturació) i riscos de seguretat (per exemple, atacs de denegació de servei).

9.3. Introducció a PVM (Parallel Virtual Machine)

PVM és un conjunt d'eines que permeten emular un entorn de computació concurrent, distribuïda i de propòsit general, fent que un conjunt heterogeni d'ordinadors connectats en xarxa es comporti com una única màquina paral·lela virtual.

El sistema PVM es compon de tres parts:

- El **dimoni** `pvmd3`, present a totes les màquines de la màquina virtual (una fa de mestra, la resta d'esclaves).
- La **biblioteca de desenvolupament (API)**: funcions per crear tasques, enviar i rebre missatges i reconfigurar la màquina virtual.
- La **consola de PVM**: un intèrpret d'ordres que fa d'interfície entre l'usuari i el dimoni.

Instal·lació (Debian/Ubuntu):

```
$ sudo apt install mpich libmpich-dev`
```

Cal instal·lar-ho a **totes** les màquines que formaran part de la màquina virtual, i tenir-hi també **SSH** configurat (amb claus sense contrasenya, mitjançant `ssh-keygen` i `ssh-copy-id`) perquè la màquina mestra pugui connectar-se a les esclaves sense demanar contrasenya cada vegada.

Configuració mínima a cada màquina (fitxer `~/ .bashrc`):

```
export PVM_ROOT=/usr/lib/pvm3
export PVM_ARCH=$(PVM_ROOT/lib/pvmgetarch)
export PVM_RSH=/usr/bin/ssh
```

I la variable `PVM_PATH` apuntant al directori on hi ha els executables de les tasques (perquè spawn els trobi):

```
export PVM_PATH=/home/usuari
```

Un cop dins la consola (`pvm`), les ordres bàsiques són:

Ordre	Funció
<code>add <màquina></code>	Incorpora una màquina a la màquina virtual
<code>conf</code>	Mostra la configuració actual
<code>spawn <tasca></code>	Llança una tasca
<code>spawn -> <tasca></code>	Per veure la sortida directament a la consola
<code>ps</code>	Llista els processos actius a PVM
<code>halt</code>	Apaga la màquina virtual i surt

9.4. Programació amb l'API de PVM

Les funcions més habituals per programar en C amb PVM (biblioteca `pvm3.h`, s'ha de compilar amb `-lpvm3`) són:

```
int    pvm_mytid(void);           // TID de la
    ↪ tasca actual
int    pvm_parent(void);          // TID del pare
int    pvm_spawn(char *tasca, char **argv, int flag,
                char *on, int ntasks, int *tids); // crea tasques
    ↪ noves
int    pvm_initsend(int codificacio); // buida el
    ↪ buffer d'enviament
int    pvm_pkstr(char *cp);        // empaqueta
    ↪ una cadena
int    pvm_pkint(int *ip, int nitem, int stride); // empaqueta
    ↪ enters
int    pvm_send(int tid, int etiqueta); // envia el
    ↪ missatge
int    pvm_recv(int tid, int etiqueta); // rep un
    ↪ missatge
int    pvm_upkstr(char *cp);        // desempaqueta
    ↪ una cadena
int    pvm_upkint(int *ip, int nitem, int stride); //
    ↪ desempaqueta enters
void    pvm_exit(void);            // surt de
    ↪ l'entorn PVM
```

El flux típic d'una aplicació PVM és:

1. El procés **mestre** crida `pvm_spawn()` per llançar una o diverses còpies d'un procés **esclau**.
2. Per **enviar** un missatge: `pvm_initsend()` → `pvm_pkXXX()` (un cop per cada dada) → `pvm_send()`.
3. Per **rebre**: `pvm_recv()` → `pvm_upkXXX()` (en el mateix ordre en què es va empaquetar).

Exemple: trobar el valor màxim d'un vector repartint-lo entre dos nodes

A diferència d'una simple suma, calcular un màxim obliga el mestre a comparar els resultats parcials en lloc de només acumular-los; és un patró habitual de *reducció* en càlcul distribuït. En aquest cas el vector es reparteix en dos blocs de mida diferent (4 i 6 elements) per il·lustrar que el repartiment no cal que sigui simètric.

Procés mestre (nodemestre.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>

#define TOTAL 10
#define ETQ_DADES 10
#define ETQ_RESULT 20
```

```

int main() {
    int vector[TOTAL] = {7, 2, 9, 4, 15, 1, 8, 23, 6, 3};
    int mida_bloc[2] = {4, 6};
    int desplaç = 0;
    int fills[2], parcials[2];
    int i, maxim;

    pvm_spawn("nodeescrau", (char**)0, 0, "", 2, fills);

    for (i = 0; i < 2; i++) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&mida_bloc[i], 1, 1);
        pvm_pkint(vector + desplaç, mida_bloc[i], 1);
        pvm_send(fills[i], ETQ_DADES);
        desplaç += mida_bloc[i];
    }

    for (i = 0; i < 2; i++) {
        pvm_recv(fills[i], ETQ_RESULT);
        pvm_upkint(&parcials[i], 1, 1);
        printf("Node %d retorna el màxim parcial: %d\n", i,
            ↪ parcials[i]);
    }

    maxim = (parcials[0] > parcials[1]) ? parcials[0] : parcials[1];
    printf("El valor màxim del vector complet és: %d\n", maxim);

    pvm_exit();
    return 0;
}

```

Procés esclau (nodeescrau.c)

```

#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>

#define ETQ_DADES 10
#define ETQ_RESULT 20

int main() {
    int bloc[10];
    int mida, i, maxim;
    int origen = pvm_parent();

    pvm_recv(origen, ETQ_DADES);
    pvm_upkint(&mida, 1, 1);
    pvm_upkint(bloc, mida, 1);

    maxim = bloc[0];
    for (i = 1; i < mida; i++)

```

```
        if (bloc[i] > maxim) maxim = bloc[i];

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&maxim, 1, 1);
    pvm_send(origen, ETQ_RESULT);

    pvm_exit();
    return 0;
}
```

Cal copiar l'executable de l'esclau (nodeesc`lau`) al directori `~/pvm3/bin/$PVM_ARCH` (o al directori definit a `$PVM_PATH`, com per exemple `/home/usuari`) de **totes** les màquines esclaves, que és on PVM cerca els executables d'usuari per defecte.

Índex de figures

1	Estats d'un procés	2
---	------------------------------	---

Versions d'aquest document

- [HTML](#)
- [PDF](#)
- [ODT](#)
- [MD](#)

[Domini Públic \(CC0\)](#)