Pràctica Forks

Processos i Comunicació en Sistemes Operatius

Alumne: Hamza El Haddad Sabri i Oscar Saborido Valdes
Data: 24/04/2025-11/05/2025

Índex

Activitat 1: Creació de processos

Aquesta activitat té com a objectiu entendre com es poden crear múltiples processos a partir del procés pare utilitzant fork().

1.1 Creació seqüencial de processos (1 pare, N fills)
Creeu un programa en C que:

- Creeu N processos fills mitjançant un bucle for executat pel procés pare.
- Cada fill ha de mostrar per pantalla:
  - L'ID de creació (la variable del bucle),
  - El seu PID i el PID del seu pare.
  - Recordeu evitar processos òrfes o zombies.
- El procés pare ha d'esperar que tots els fills acabin.

1.2 Exploració del creixement exponencial de processos
Ara creeu un nou codi a partir de l'anterior per tal que cada procés creat continuï executant el bucle for.

- Afegiu un sleep(1) (1s) per observar l'execució progressiva i deixeu l'execució del codi durant 5 segons, o menys en cas de problemes.
- Comproveu quants processos es creen en total. Quin comportament segueix el codi?
- Mostreu per pantalla el PID de cada procés i el nombre total de processos per a cada moment.

Atenció: aquest codi pot generar un nombre molt alt de processos si no es controla el nombre d'iteracions i el temps entre iteracions, cosa que pot provocar que el sistema operatiu col·lapsi.

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

int main() {
    int N = 5; // Nombre de processos fills a crear

    // Crear N processos fills
    for (int i = 1; i <= N; i++) {
        pid_t pid = fork(); // Crear un procés fill
        if (pid == -1) {
            // Si fork falla
            perror("Fork failed");
            exit(1);
        }

        if (pid == 0) {
            // Codi del procés fill
            printf("Fill %d: PID = %d, Pare PID = %d\n", i, getpid(), getppid());
            exit(0);
        }
    }

    // El procés pare espera que tots els fills acabin
    for (int i = 0; i < N; i++) {
        wait(NULL);
    }

    printf("Procés pare acabat. Tots els fills han acabat.\n");

    return 0;
}
```

1.1

Utilitzem un bucle for per crear N processos fills.

Cada vegada que s'executa fork(), es crea un nou procés. Si fork() en retorna 0, vol dir que estem en el procés fill; si retorna un valor positiu, estem en el procés pare.

Cada procés fill imprimeix el seu ID de creació (la variable del bucle), el seu PID i el PID del seu pare amb les funcions getpid() i getppid()

El procés pare espera que els fills acabin abans de finalitzar pel que no queden processos orfes.

```c
int main() {
    int N = 5; // Nombre de processos fills inicials a crear

    for (int x = 1; x <= N; x++) {
        pid_t pid = fork(); // Crear un procés fill

        if (pid == -1) {
            perror("Fork failed");
            exit(1);
        }

        if (pid == 0) {
            // Procés fills inicials
            printf("Procés fill 1r nivell: PID = %d, Pare PID = %d\n", getpid(), getppid());

            for (int y = 1; y <= N; y++) {
                pid_t pid2 = fork();
                if (pid2 == -1) {
                    perror("Fork failed");
                    exit(1);
                }

                if (pid2 == 0) {
                    // Procés net (2n nivell)
                    printf("Procés net: PID = %d, Pare PID = %d\n", getpid(), getppid());
                    sleep(1);
                    exit(0);
                }
            }

            // Esperar els fills del fill
            for (int j = 0; j < N; j++) wait(NULL);
            exit(0);
        }

        sleep(1); // Per evitar que tots els forks passin alhora
    }

    // Esperar a que els fills acabin
    for (int i = 0; i < N; i++) wait(NULL);

    int total_processos = 1 + N + N * N;
    printf("Procés pare acabat. Total de processos creats: %d\n", total_processos);
    return 0;
}
```

1.2

Procés pare: Crea N processos fills.

Procés fill: Cada fill crea N nous processos, seguint l'estructura de creixement exponencial.

Cada procés fill imprimeix el seu PID, el PID del seu pare.

Sleep(1): Espera un segon per observar l'execució. N'hem utilitzat 5, ja que amb 5 processos inicials en el cas d'aquest codi, es produeixen en total 5 sleeps d'1 segon, per tal que el codi duri els 5 segons demanats anteriorment.

El procés pare espera per acabar a què tots els fills hagin finalitzat, mostra el nombre total de processos creats amb el càlcul:

int total_processos = 1*(el pare)* **+ *N****(fills inicials)* **+ *N * N****(fills dels fills)****;*

fins al moment i finalitza el codi.

Activitat 2: Comunicació entre processos per parelles

Creeu un programa en C que permeti la comunicació entre processos:

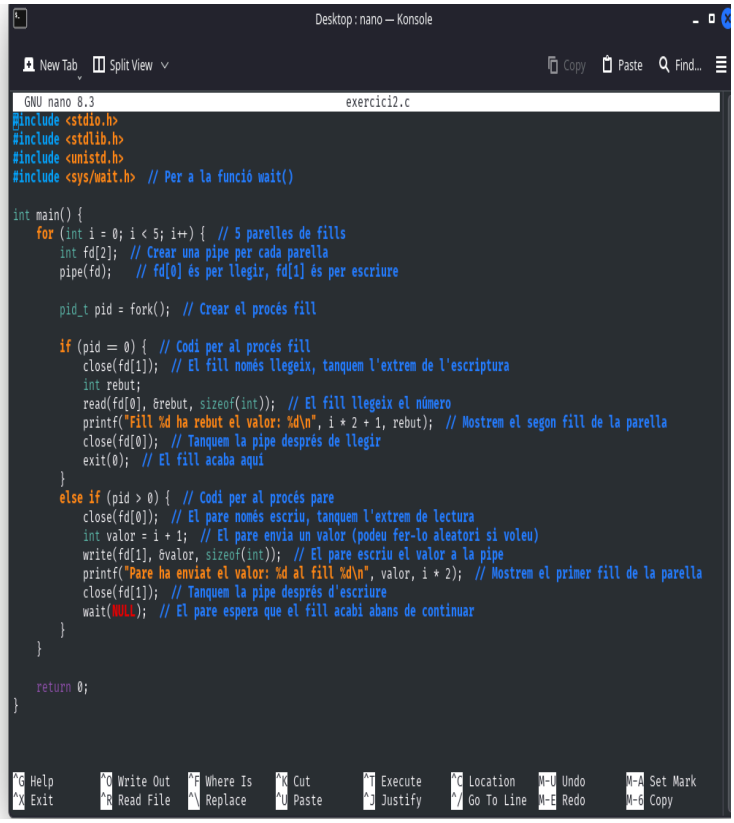
• Creeu 10 processos fills (del 0 al 9).
• El procés pare haurà d'enviar informació a la meitat dels processos fills i aquests hauran de compartir una informació entre ells.

Tanmateix, hi ha una restricció: la comunicació entre processos només pot ser per parelles d'id: 0↔1, 2↔3, ..., 8↔9. És a dir, el procés amb id 0 no pot comunicar amb el procés amb id 2.
• La idea seria establir una comunicació així: pare → fill 0 → fill 1, pare → fill 2 → fill 3...
• Podeu establir valors inicials totalment aleatoris des del pare als fills. Aquests han de ser diferents per a cada pack de processos i s'han de mostrar per pantalla per a cada pas: valor del pare, valor del procés X i valor del procés Y.

Exemple (pack 0-1):

1. El pare envia un número al procés amb id = 0.
2. El procés 0 fa una multiplicació i envia el resultat al procés 1.
3. Es mostra per pantalla el valor, per al procés pare, 0 i 1.

Recomanacions tècniques:
Utilitzeu `pipe()` i `fork()`. Per cada parella, creeu les pipes necessàries (intenteu evitar excés de pipes):
• pare → fill X
• fill X → fill Y



Aquest codi crea 10 processos fills amb un bucle for i fork(). El procés pare envia un número aleatori a cada primer procés de cada parella (0, 2, 4, 6 i 8). Aquests primers fills reben el número per una pipe, fan una multiplicació amb aquest número, i envien el resultat al segon procés de la parella (1, 3, 5, 7 i 9) per una altra pipe.

Cada parella només comunica entre si. El procés 0 no pot parlar amb el 2, ni el 2 amb el 4, etc. La comunicació sempre és així: pare → fill X → fill Y.

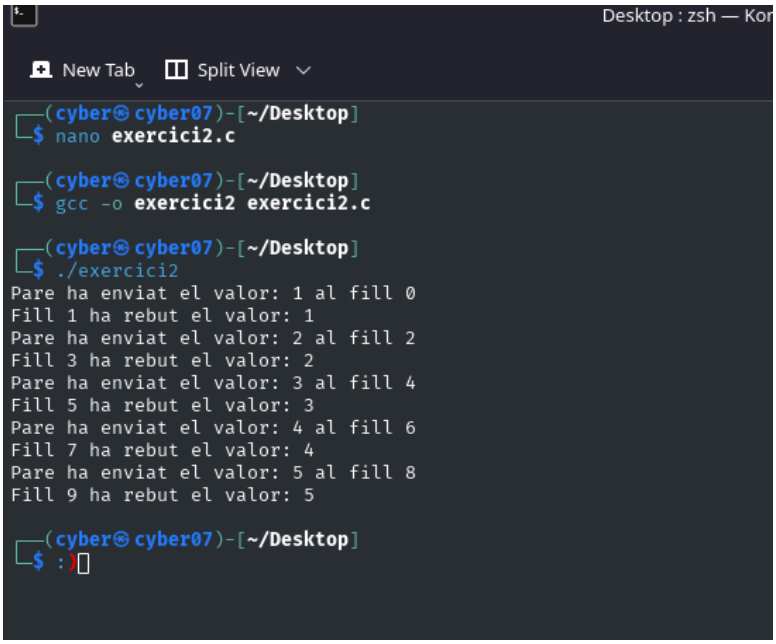
Es creen dues pipes per parella:

Una per enviar el valor del pare al primer fill (X).

Una altra per enviar el valor del primer fill (X) al segon (Y).

Els processos tanquen les pipes que no necessiten per evitar errors.

Al final, tots els processos mostren per pantalla els valors rebuts i enviats, indicant el seu ID de procés. El pare espera que tots els fills acabin amb wait().



Quan s'executa el programa, es pot veure com el pare envia un valor al primer procés de cada parella, i aquest primer procés fa una operació amb aquest valor (com multiplicar-lo per 2) i envia el resultat al seu company. El segon procés de la parella rep aquest nou valor i el mostra per pantalla. Aquest patró es repeteix per a totes les parelles, sempre seguint l'ordre establert: el pare parla amb el 0, el 2, el 4, el 6 i el 8; aquests, al seu torn, parlen amb els seus respectius companys: 1, 3, 5, 7 i 9.

Activitat 3: Exclusió mútua amb mutex (fils amb pthread)

Creeu un programa en C que utilitzi fils amb pthread per simular accés concurrent a una variable compartida. Aquesta activitat té com a objectiu entendre el concepte de condició de competència i com evitar-la mitjançant exclusió mútua.

• Creeu 5 fils amb pthread.
• Cada fil executa una funció que incrementa la variable global `contador` 10000 vegades. Aquesta variable comença amb un valor 0.
• Protegiu l'accés a `contador`.
• Mostreu el valor final de `contador` i comproveu si és igual a 50000 o diferent. Comenteu el perquè.
• Què passaria si no es protegeix la variable global correctament? Demostreu els resultats.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NTHREADS 5
#define NITER 10000

int contador = 0;               // variable global compartida
pthread_mutex_t mutex;          // declaració del mutex

void* incrementa(void* arg) {
    for (int i = 0; i < NITER; ++i) {
        // Secció crítica: incrementar el contador
        pthread_mutex_lock(&mutex);
        contador += 1;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NTHREADS];
    // Inicialitzar el mutex
    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("Error inicialitzant mutex");
        exit(1);
    }
    // Crear els fils
    for (int i = 0; i < NTHREADS; ++i) {
        if (pthread_create(&threads[i], NULL, incrementa, NULL) != 0) {
            perror("Error creant fil");
            exit(1);
        }
    }
    // Esperar que tots els fils acabin
    for (int i = 0; i < NTHREADS; ++i) {
        pthread_join(threads[i], NULL);
    }
    // Destruir el mutex
    pthread_mutex_destroy(&mutex);

    printf("Valor final del contador: %d\n", contador);
    if (contador == NTHREADS * NITER) {
        printf("Correcte: el valor esperat %d coincideix amb el valor final.\n", NTHREADS * NITER);
    } else {
        printf("Advertència: valor final %d diferent de l'esperat %d (condició de competència?)\n",
               contador, NTHREADS * NITER);
    }
    return 0;
}
```

```
Valor final del contador: 50000
Correcte: el valor esperat 50000 coincideix amb el valor final.
```

Codi utilitzant mutex

Utilitzem fils (pthread) per crear concurrència dins d'un mateix procés.

Creem 5 fils que comparteixen una variable global contador.

Cada fil incrementa aquesta variable 10000 vegades.

Protegim l'accés a la secció crítica utilitzant un mutex, amb les funcions pthread_mutex_lock() i pthread_mutex_unlock().

D'aquesta manera, evitem que dos fils modifiquin el valor de contador al mateix temps.
Inicialitzem el mutex abans de crear els fils i l'alliberem un cop tots han finalitzat.

El procés principal espera que tots els fils acabin i imprimeix el valor final del contador.
Si el valor final és 50000, vol dir que l'exclusió mútua ha funcionat correctament.

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

#define NTHREADS 5
#define NITER 10000

int contador = 0;               // variable global compartida

void* incrementa(void* arg) {
    for (int i = 0; i < NITER; ++i) {
        // Secció crítica: incrementar el contador
        contador += 1;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NTHREADS];

    // Crear els fils
    for (int i = 0; i < NTHREADS; ++i) {
        if (pthread_create(&threads[i], NULL, incrementa, NULL) != 0) {
            perror("Error creant fil");
            exit(1);
        }
    }
    // Esperar que tots els fils acabin
    for (int i = 0; i < NTHREADS; ++i) {
        pthread_join(threads[i], NULL);
    }


    printf("Valor final del contador: %d\n", contador);
    if (contador == NTHREADS * NITER) {
        printf("Correcte: el valor esperat %d coincideix amb el valor final.\n", NTHREADS * NITER);
    } else {
        printf("Advertència: valor final %d diferent de l'esperat %d (condició de competència?)\n",
               contador, NTHREADS * NITER);
    }
    return 0;
}
```

```
Valor final del contador: 48820
Advertència: valor final 48820 diferent de l'esperat 50000 (condició de competència?)
```

Codi sense utilitzar mutex

Aquest codi és igual que l'anterior però no utilitza cap mutex per protegir la secció crítica.
Creem 5 fils que comparteixen una variable global contador.

Cada fil incrementa aquesta variable 10000 vegades.

En no utilitzar sincronització, diversos fils poden accedir a contador al mateix temps.

Això provoca condicions de competència i pèrdues d'increments.

El valor final del contador és inferior a 50000 i pot variar entre execucions.

Demostrem així que és necessari protegir l'accés concurrent a dades compartides.

Activitat 4: Control de concurrència amb semàfors

Creeu un programa en C que simuli l'accés concurrent a un recurs limitat, com per exemple una impressora, on només 3 processos poden utilitzar-la alhora.

Simuleu un entorn amb 10 processos. Cada un d'ells representa un usuari que vol fer servir una impressora compartida. Només poden accedir 3 usuaris simultàniament.

• Utilitzeu les crides de semàfors de POSIX.
• Cada procés ha de:
– Mostrar un missatge quan vol accedir a la impressora (esperant torn). Aquesta esperarà fins que el recurs estigui lliure.
– Mostrar un missatge quan entra (imprimint) i fer un `sleep(2)` per simular el temps d'impressió.
– Alliberar recurs i mostrar un missatge de sortida (ha acabat d'imprimir).
• Executeu el programa i comproveu que mai imprimeixen més de 3 processos a la vegada. Mostreu els resultats.

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

int main() {
    const char *SEM_NAME = "/sem_impressora";
    sem_t *sem = sem_open(SEM_NAME, O_CREAT | O_EXCL, 0644, 3);
    if (sem == SEM_FAILED) {
        perror("Error creant semàfor");
        sem_unlink(SEM_NAME);
        exit(1);
    }

    for (int i = 1; i <= 10; ++i) {
        pid_t pid = fork();
        if (pid < 0) {
            perror("Error en fork");
            exit(1);
        }
        if (pid == 0) {
            // Codi de cada procés fill (usuari)
            printf("Usuari %d: vol accedir a la impressora (esperant torn)\n", i);
            sem_wait(sem);  // espera fins que hi hagi un "lloc" disponible (valor semàfor >0)
            // Secció crítica: ús de la impressora
            printf("Usuari %d: ha entrat a la impressora (imprimint)\n", i);
            sleep(2);       // simular que està imprimint durant 2 segons
            printf("Usuari %d: ha acabat d'imprimir (alliberant impressora)\n", i);
            sem_post(sem);  // allibera un lloc en acabar
            exit(0);
        }
        // El procés pare continua creant la resta de processos
    }
    // Procés pare espera tots els fills
    for (int j = 1; j <= 10; ++j) {
        wait(NULL);
    }
    // Tancar i eliminar el semàfor
    sem_close(sem);
    sem_unlink(SEM_NAME);
    return 0;
}
```

Utilitzem processos per simular l'accés concurrent de diversos usuaris a una impressora compartida.

Creem 10 processos fills.
Inicialitzem un semàfor POSIX amb valor 3, que representa el nombre màxim de processos que poden accedir a la impressora alhora.

Cada procés mostra un missatge quan vol imprimir, espera el seu torn amb sem_wait(), imprimeix durant 2 segons (sleep(2)) i allibera el recurs amb sem_post().
El procés pare crea tots els fills i espera que finalitzin.

El semàfor garanteix que com a màxim 3 processos estiguin imprimint simultàniament.
Finalment, tanquem i desemllacem el semàfor.

```
Usuari 1: ha entrat a la impressora (imprimint)
Usuari 2: vol accedir a la impressora (esperant torn)
Usuari 2: ha entrat a la impressora (imprimint)
Usuari 3: vol accedir a la impressora (esperant torn)
Usuari 3: ha entrat a la impressora (imprimint)
Usuari 4: vol accedir a la impressora (esperant torn)
Usuari 5: vol accedir a la impressora (esperant torn)
Usuari 6: vol accedir a la impressora (esperant torn)
Usuari 7: vol accedir a la impressora (esperant torn)
Usuari 8: vol accedir a la impressora (esperant torn)
Usuari 9: vol accedir a la impressora (esperant torn)
Usuari 10: vol accedir a la impressora (esperant torn)
Usuari 1: ha acabat d'imprimir (alliberant impressora)
Usuari 4: ha entrat a la impressora (imprimint)
Usuari 2: ha acabat d'imprimir (alliberant impressora)
Usuari 5: ha entrat a la impressora (imprimint)
Usuari 3: ha acabat d'imprimir (alliberant impressora)
Usuari 6: ha entrat a la impressora (imprimint)
Usuari 4: ha acabat d'imprimir (alliberant impressora)
Usuari 5: ha acabat d'imprimir (alliberant impressora)
Usuari 7: ha entrat a la impressora (imprimint)
Usuari 6: ha acabat d'imprimir (alliberant impressora)
Usuari 8: ha entrat a la impressora (imprimint)
Usuari 9: ha entrat a la impressora (imprimint)
Usuari 7: ha acabat d'imprimir (alliberant impressora)
Usuari 8: ha acabat d'imprimir (alliberant impressora)
Usuari 10: ha entrat a la impressora (imprimint)
Usuari 9: ha acabat d'imprimir (alliberant impressora)
Usuari 10: ha acabat d'imprimir (alliberant impressora)
```

Com es pot veure en l'output, només 3 usuaris poden imprimir al mateix temps.

Els altres mostren el missatge "vol accedir a la impressora (esperant torn)" fins que un lloc queda lliure.

Quan un usuari "ha acabat d'imprimir (alliberant impressora)", immediatament un altre "ha entrat a la impressora (imprimint)".

Això confirma que el semàfor limita correctament l'accés a 3 processos simultanis, tal com estava previst en l'activitat.

Activitat 5: Comunicació + sincronització

Objectiu: Implementar un sistema on diversos processos comparteixen dades, s'esperen entre ells i controlen l'accés a recursos.

- El pare crea 10 processos fills.
- Cada 2 processos formen un equip (5 equips en total).
- El pare envia un temps d'espera (valor) a cada primer membre de cada equip amb `pipe()`.
- Aquest primer fill fa un `sleep()` amb el valor rebut (simulant feina), suma un valor i l'envia al segon del seu equip.
- El segon fa un `sleep()` amb el valor rebut.
- Només poden estar 2 equips treballant simultàniament.
- El pare no ha de rebre cap resultat.

Reptes: combinació de `fork()`, `pipe()`, `sleep()` i semàfors.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <fcntl.h>
#include <time.h>

int main() {
    int pipes_pf[5][2];  // pipes pare->primer fill equip
    int pipes_ff[5][2];  // pipes primer fill->segon fill equip
    const char *SEM_NAME = "/sem_equips";
    sem_t *sem = sem_open(SEM_NAME, O_CREAT | O_EXCL, 0644, 2);
    if (sem == SEM_FAILED) {
        perror("Error creant semàfor d'equips");
        sem_unlink(SEM_NAME);
        exit(1);
    }

    // Crear pipes per a cada equip
    for (int e = 0; e < 5; ++e) {
        if (pipe(pipes_pf[e]) == -1 || pipe(pipes_ff[e]) == -1) {
            perror("Error creant pipes");
            exit(1);
        }
    }

    // Crear processos fills (10 en total)
    for (int i = 0; i < 10; ++i) {
        pid_t pid = fork();
        if (pid < 0) {
            perror("Error en fork");
            exit(1);
        }
        if (pid == 0) {
            int id = i;
            // Tancar extrems de pipes no necessaris segons el rol del procés fill
            for (int e = 0; e < 5; ++e) {
                if (!(id % 2 == 0 && e == id/2)) {
                    close(pipes_pf[e][0]);    // tanca lectura pare->fill de la resta d'equips
                }
                close(pipes_pf[e][1]);        // tanca tots els extrems d'escriptura pare->fill (només el pare els utilitza)
                if (!(id % 2 == 1 && e == id/2)) {
                    close(pipes_ff[e][0]);    // tanca lectura fill->fill de la resta d'equips
                }
                if (!(id % 2 == 0 && e == id/2)) {
                    close(pipes_ff[e][1]);    // tanca escriptura fill->fill de la resta d'equips
                }
            }
            if (id % 2 == 0) {
                // Primer membre de l'equip (id parell)
                sem_wait(sem);  // Esperar permís per iniciar (només 2 equips simultanis)
                int temps;
                read(pipes_pf[id/2][0], &temps, sizeof(temps));  // Llegir valor del pare
                printf("Procés %d (primer de l'equip %d-%d) rep temps %d\n", id, id, id+1, temps);
                // Treballar: simular feina dormint 'temps' segons
                sleep(temps);
                // Sumar un valor fix, per exemple 1, abans d'enviar al company
                int enviat = temps + 1;
                write(pipes_ff[id/2][1], &enviat, sizeof(enviat));
                printf("Procés %d envia valor %d al procés %d\n", id, enviat, id+1);
                exit(0);
            } else {
                // Segon membre de l'equip (id imparell)
                int rebut;
```

Procés pare: Crea 10 processos fills, agrupats en 5 equips de 2 processos (procés 0 i 1 formen l'equip 0, el 2 i 3 l'equip 1, etc.).

Per a cada equip, el pare envia un valor aleatori (entre 1 i 5 segons) al primer membre amb un pipe().

Aquest primer membre llegeix el valor, fa un sleep() amb aquest temps, suma 1 al valor i l'envia al seu company d'equip amb un altre pipe().

El segon membre rep el valor modificat, fa sleep() amb aquest nou valor i finalitza.
Utilitzem un semàfor inicialitzat a 2 per limitar a només 2 equips treballant simultàniament.

El primer membre fa sem_wait() abans de començar, i el segon membre fa sem_post() en acabar, alliberant l'accés per un altre equip.

El procés pare espera que tots els fills acabin i allibera el semàfor un cop finalitza tot el procés.

```
            } else {
                // Segon membre de l'equip (id imparell)
                int rebut;
                read(pipes_ff[id/2][0], &rebut, sizeof(rebut));  // LLegir valor del primer del seu equip
                printf("Procés %d (segon de l'equip %d-%d) rep valor %d\n", id, id-1, id, rebut);
                // Simular feina dormint 'rebut' segons
                sleep(rebut);
                printf("Procés %d (segon de l'equip %d-%d) finalitza\n", id, id-1, id);
                sem_post(sem);  // Alliberar permís perquè un altre equip comenci
                exit(0);
            }
        }
        // el pare continua creant processos
    }

    // El procés pare tanca els extrems que no usa i envia els valors inicials
    srand(time(NULL));
    for (int e = 0; e < 5; ++e) {
        // tancar extrems no usats pel pare
        close(pipes_pf[e][0]);
        close(pipes_ff[e][0]);
        close(pipes_ff[e][1]);
        // Enviar un temps d'espera aleatori (1-5 segons) a cada equip
        int t = 1 + rand() % 5;
        write(pipes_pf[e][1], &t, sizeof(t));
        printf("Pare envia temps %d al procés %d\n", t, 2*e);
        close(pipes_pf[e][1]);  // tancar l'extrem d'escriptura després d'enviar
    }

    // Esperar que acabin tots els fills
    for (int j = 0; j < 10; ++j) {
        wait(NULL);
    }
    // Tancar i eliminar el semàfor
    sem_close(sem);
    sem_unlink(SEM_NAME);
    return 0;
}
```

```
Pare envia temps 1 al procés 0
Pare envia temps 2 al procés 2
Pare envia temps 5 al procés 4
Procés 0 (primer de l'equip 0-1) rep temps 1
Pare envia temps 3 al procés 6
Pare envia temps 3 al procés 8
Procés 2 (primer de l'equip 2-3) rep temps 2
Procés 0 envia valor 2 al procés 1
Procés 1 (segon de l'equip 0-1) rep valor 2
Procés 2 envia valor 3 al procés 3
Procés 3 (segon de l'equip 2-3) rep valor 3
Procés 1 (segon de l'equip 0-1) finalitza
Procés 4 (primer de l'equip 4-5) rep temps 5
Procés 3 (segon de l'equip 2-3) finalitza
Procés 6 (primer de l'equip 6-7) rep temps 3
Procés 4 envia valor 6 al procés 5
Procés 5 (segon de l'equip 4-5) rep valor 6
Procés 6 envia valor 4 al procés 7
Procés 7 (segon de l'equip 6-7) rep valor 4
Procés 7 (segon de l'equip 6-7) finalitza
Procés 8 (primer de l'equip 8-9) rep temps 3
Procés 5 (segon de l'equip 4-5) finalitza
Procés 8 envia valor 4 al procés 9
Procés 9 (segon de l'equip 8-9) rep valor 4
Procés 9 (segon de l'equip 8-9) finalitza
```

Com es pot veure en l'output, el procés pare envia un temps a cada primer membre dels equips (procés 0, 2, 4, 6 i 8).

Cada un d'aquests primers processos rep el valor, fa sleep() i envia un nou valor al seu company d'equip.

El segon membre (procés 1, 3, 5, 7 i 9) rep aquest valor, fa també sleep() i finalitza.

Només hi ha 2 equips treballant alhora, ja que la resta esperen fins que un equip allibera el semàfor.

Això confirma que la sincronització amb semàfor i la comunicació amb pipes funciona correctament.