

# 1 TD1

## 1.1 Ex1

P1 à t = 0

Calcul pendant 3 unités

Impression sur imprimante pendant 2 unités

Calcul pendant 2 unités

Affichage à l'écran pendant 1 unité

Calcul pendant 3 unités

FIN.

P2 à t = 1

Calcul pendant 2 unités

Affichage à l'écran pendant 2 unités

Calcul pendant 1 unité

Impression sur imprimante pendant 2 unités

FIN.

### 1) Mono-programmé

CPU	$p_1$	$p_1$	$p_1$			$p_1$	$p_1$		$p_1$	$p_1$	$p_2$	$p_2$		$p_2$		
Impression				$p_1$	$p_1$										$p_2$	$p_2$
Écran								$p_1$					$p_2$	$p_2$		
Attente		$p_2$	$p_2$	$p_2$	$p_2$	$p_2$	$p_2$	$p_2$	$p_2$	$p_2$	$p_2$					

Occupation Processeur  $\frac{11}{18} \rightarrow 61\%$

### 2) Multi-programmé

CPU	$p_1$	$p_1$	$p_1$	$p_2$	$p_2$	$p_1$	$p_1$	$p_2$	$p_1$	$p_1$	$p_1$
Impression				$p_1$	$p_1$				$p_2$	$p_2$	
Écran						$p_2$	$p_2$	$p_1$			
Attente		$p_2$	$p_2$								

Occupation Processeur  $\frac{11}{11} \rightarrow 100\%$

### 3) Temps Partagé (1s)

CPU	$p_1$	$p_2$	$p_1$	$p_2$	$p_1$		$p_2$	$p_1$	$p_1$		$p_1$	$p_1$	$p_1$
Impression							$p_1$	$p_1$	$p_2$	$p_2$			
Écran					$p_2$	$p_2$					$p_1$		
Attente		$p_1$	$p_2$	$p_2$									

Occupation Processeur  $\frac{11}{13} \rightarrow 84\%$

## 1.2 Ex2

### 1.2.1 États possibles de processus

- Prêt : Attente du processeur
- En Exécution :
- Bloqué : Attente IO

### 1.2.2 Files d'attente nécessaires pour gérer les processus de ce système.

Plusieurs programmes coexistent à l'état d'attente (ou bloqué) → deux files

### 1.2.3 Comment gérer les priorités

Favoriser les processus demandant de l'I/O en augmentant la priorité du processus lors de l'accès à l'I/O (puis de le réduire si il n'en a pas fait depuis un certain temps)

### 1.2.4 PCB

- PID
- État
  - AX, BX, CX, DX, EX
  - CO
  - FLAGS
  - DS, CS
- Cause du blocage
- Priorité, quantum

## 1.3 Ex3

On représente RM avec

<b>EX</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>H</b>
x	x	x	x	x	x

Chaque interruption à un masque propre afin de représenter la priorité :

<b>EX</b>	0	0	0	0	0	0
<b>A</b>	1	0	0	0	0	0
<b>B</b>	1	1	0	0	0	0
<b>C</b>	1	1	1	0	0	0
<b>D</b>	1	1	1	1	0	0
<b>H</b>	1	1	1	1	1	0

## 1.4 Ex4

*Exemple foireux et inutile.*

## 2 TD2

### 2.1 Ex1

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

pid_t chld;

void usr1Handler() {
    printf(" Temperature %d\n", (rand() % 30) + 10);    fflush(stdout);
}

void alrmHandler() {
    kill(chld, SIGUSR1);    alarm(5);
}

int main(int argn, char* argv[]) {
    struct sigaction newUsr1Handler, newAlrmHandler;
    newUsr1Handler.sa_handler = usr1Handler;
    newAlrmHandler.sa_handler = alrmHandler;

    srand(time(NULL));

    if ((chld = fork()) == -1) {
        perror("fork");
        return -1;
    }

    /* Child's Process */
    if(chld == 0) {
        sigaction(SIGUSR1, &newUsr1Handler, NULL);
        while(1) pause();
    }

    /* Parent's Process */
    else {
        sigaction(SIGALRM, &newAlrmHandler, NULL);
        alarm(5);

        while(1) {
            sleep(1);    putchar('-');    fflush(stdout);
        }
    }

    return 0;
}
```

## 2.2 Ex2

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

pid_t chld;
char curChar;
char step = 1;

void child_handler() {
    for(int i= 0; i< step && curChar <= 'z'; i++) {
        printf("%c", curChar);
        curChar++;
    }
    step++;
    kill(getppid(), SIGUSR1);
}

void parent_handler() {
    for(int i= 0; i< step && curChar <= 'Z'; i++) {
        printf("%c", curChar);
        curChar++;
    }
    if(curChar <= 'Z') {
        step++;
        kill(chld, SIGUSR1);
    }
}

int main(int argn, char* argv[]) {
    struct sigaction sa;

    if ((chld = fork()) == -1) {
        perror("fork");
        return -1;
    }

    /* Child's Process */
    if(chld == 0) {
        sa.sa_handler = child_handler;
        curChar = 'a';
        sigaction(SIGUSR1, &chldUsr1Handler, NULL);
        while(1) pause();
    }

    /* Parent's Process */
    else {
        sa.sa_handler = parent_handler;
        curChar = 'A';
        sigaction(SIGUSR1, &prntUsr1Handler, NULL);
        kill(chld, SIGUSR1);
        while(1) pause();
    }

    return 0;
}
```

## 3 TD3

### 3.1 Ex1 – Programmation d'un Robot

Un robot dispose de 2 roues de circonférence 10cm. Pour faire avancer le robot on fait appel au SVC (appel système). Le contrôleur du robot dispose de deux registres :

- RE : Registre d'état mis à 1 quand le robot est prêt.
- RC : Registre faisant avancer le robot d'un tour de roue quand mis à 1.

#### 3.1.1 Sans mécanisme d'interruption – Mono-processus

```
def rouler(n):
    save_context(PCB[actif])

    for i in range(n):
        while RE != 1:
            wait()
        RC = 1

    load_context(PCB[actif]) ;
    load_psw(PCB[actif].psw)
```

#### 3.1.2 Avec mécanisme d'interruption – Multi-processus

En admettant qu'une interruption est envoyée quand les roues ont fait un tour complet :

```
def rouler(n):
    save_context(PCB[active]) ; blocked_queue.push(active)

    p = packet_io(pid = PCB[active].pid, nb = n, type = 1)
    request_queue.push(p)
    new_pr = ready_queue.pop()

    load_context(PCB[active]) ; load_psw(PCB[active].psw)

def driver_robot():
    while True:
        if not request_queue.empty:
            p = request_queue.pop()
            while(RE != 1):
                wait()
            RCC = p.type
            p.nb--
            pause()
            ready_queue.push(blocked_queue.pop())

def robot_int_routine(p):
    save_context(PCB[active]);
    if p.nb == 0:
        # envoie d'un signal pour reveiller le driver
    else:
        RC = p.type
        p.nb--

    load_context(PCB[active]) ; load_psw(PCB[active].psw)
```

## 3.2 Ex2

Un disque comporte 200 pistes ([0..199]). La tête de lecture est en 112. File d'attente :

138 91 165 67 158 43 132 28 106 84

### 3.2.1 Durée de déplacement par algos

<b>FIFO</b>	First-in First-out	732
<b>PCTR</b>	Plus court temps de recherche	202
<b>SCAN</b>	Va d'un bout à l'autre en répondant au passage puis demi tour	258
<b>LOOK</b>	Pareil que SCAN mais va pas au bout si inutile	190

### 3.2.2 Danger d'autoriser traitement immédiat nouvelles requêtes ? Quels algorithmes sont sujets à ces famines ?

Peut stagner si toutes les opérations se font au même endroit, oubliant les autres. Le PCTR.

## 3.3 Ex3

Un fichier occupant 100 blocs et un répertoire ou un index occupant un bloc, combien de transferts disque-ram sont nécessaires pour ajouter un bloc au début du fichier et enlever un bloc à la fin du fichier :

### Contiguë

—  
—

### Chaînage

- Accès au répertoire, écriture sur le disque : 2
- Accès au répertoire, parcours de tous les blocs : 101

### Chaînage Indexé

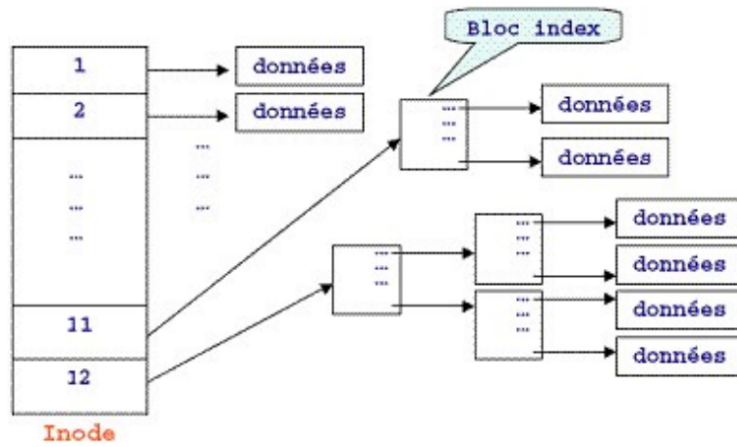
- Accès au répertoire, écriture sur le disque : 2
- Accès au répertoire : 1

### Indexé

- Accès au répertoire, écriture sur le disque : 2
- Accès au répertoire : 1

### 3.4 Ex4

Avec un i-node comme ça :



Chaque bloc d'index pouvant contenir 256 adresse de blocs, combien d'accès sont nécessaires pour accéder au 583ème bloc d'un fichier qui en compte 896 ?

## 4 TD4

### 4.1 Ex1

#### 4.1.1

Fonction `copyFile(int f1, int f2)` copiant le contenu d'un fichier dans un autre et retournant le nombre d'octets copiés. La copie se fait par blocs de 1024o. Programme affichant deux fichiers puis la taille lue grâce à deux processus parent et fils :

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

#define READSIZE 1024

int copyFile(int f1, int f2) {
    int totalByteRead= 0, lastByteRead= READSIZE;
    char buffer[READSIZE];

    while(lastByteRead == READSIZE) {
        lastByteRead = read(f1, buffer, READSIZE);
        write(f2, buffer, lastByteRead);
        totalByteRead += lastByteRead;
    }

    return totalByteRead;
}

int main(int argc, char* argv[]) {
    int file;
    pid_t chld = fork();

    if(chld == 0)    { file = open((argc > 2) ? argv[2] : "file2",
        O_RDONLY);    }
    else            { file = open((argc > 1) ? argv[1] : "file1",
        O_RDONLY); wait(NULL); }

    printf("\n%d bytes\n", copyFile(file, fileno(stdout)));
    close(file);
    return 0;
}
```



### 4.1.2

Pareil mais en utilisant une mémoire partagée pour que le fils indique au père son nombre d'octets lus :

```
int main(int argc, char* argv[]) {
    int file;
    id_t id = shmget(IPC_PRIVATE, sizeof(int), 0666);
    int* shmArea = (int*) shmat(id, NULL, 0);

    pid_t chld = fork();

    if(chld == 0)    file = open((argc > 2) ? argv[2] : "file2", O_RDONLY);
    else            file = open((argc > 1) ? argv[1] : "file1", O_RDONLY);

    int byteCopied = copyFile(file, fileno(stdout));
    close(file);

    if(chld == 0) *shmArea = byteCopied;
    else
    {
        wait(NULL);
        printf("Bytes copied: %d & %d\n", byteCopied, *shmArea);

        shmdt((void*) shmArea);
        shmctl(id, IPC_RMID, NULL);
    }

    return 0;
}
```

### 4.1.3

```
#include <unistd.h>

int main(int, char**) {
    int pid, pip[2];
    char instring[20];
    pipe(pip);
    pid = fork();

    // Child
    if (pid == 0) {
        close(pip[0]);
        write(pip[1], "Salut !", 7);
    }

    // Parent
    else {
        close(pip[1]);
        read(pip[0], instring, 7);
    }

    return 0;
}
```

## 4.2 Ex2

Application serveur – client échangeant a-z :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27

void server() {
    char c;
    int shmid;
    key_t key = 5678;
    char *shm, *s;

    // Creer le segment
    shmid = shmget(key, SHMSZ, IPC_CREAT | 0666);
    if(shmid < 0) { perror("shmget"); exit(1); }

    // Attacher le segment
    shm = shmat(shmid, NULL, 0);
    if (shm == (char *) -1) { perror("shmat"); exit(1); }

    // Mettre quelques choses dans la memoire pour l'autre processus
    s = shm;
    for (c = 'a'; c <= 'z'; c++) *s++ = c;
    *s = NULL;

    // On attend que le client lise en mettant en premier caractere '*'
    while (*shm != '*') sleep(1);

    shmdt((void*) shm);
    shmctl(shmid, IPC_RMID, NULL);
    exit(0);
}

void client() {
    int shmid;
    char *shm, *s;

    // Obtenir le segment "5678" cree par le serveur
    key_t key = 5678;
    shmid = shmget(key, SHMSZ, 0666);
    if(shmid < 0) { perror("shmget"); exit(1); }

    // Attacher le segment a notre espace de donnees
    shm = shmat(shmid, NULL, 0);
    if (shm == (char*) -1) { perror("shmat"); exit(1); }

    // Lire ce que le serveur a mis dans la memoire
    for (s = shm; *s != NULL; s++) putchar(*s);
    putchar('\n');

    // Changez le premier caractere du segment en '*' pour indiquer la
    // lecture du segment
    *shm = '*';
    exit(0);
}
```

## 4.3 Ex3

Programmes modifiant et affichant un fichier binaire composé de 10 entiers à l'aide d'une mémoire partagée :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#define FILESIZE 10

int main(int argc, char* argv[]) {
    int i= 0, fd = open("titi.dat", O_RDWR, 0666);

    int* fileMap = (int*) mmap(NULL, FILESIZE*sizeof(int), PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, 0);

    if(fileMap == (int*) MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);

    while(1) {
        scanf("%d", &i);
        if(i == 99) break;
        if(i < 10 && i >= 0) fileMap[i] = fileMap[i]+1;
    }

    munmap((void*) fileMap, FILESIZE*sizeof(int));
    return 0;
}
```

```
int main(int argc, char* argv[]) {
    int i= 0, fd = open("titi.dat", O_RDONLY, 0666);

    int* fileMap = (int*) mmap(NULL, FILESIZE*sizeof(int), PROT_READ,
        MAP_PRIVATE, fd, 0);

    if(fileMap == (int*) MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);

    while(1) {
        scanf("%d", &i);
        if(i == 99) break;
        for(int j= 0; j< 10; j++) printf("\t%d\n", fileMap[j]);
    }

    munmap((void*) fileMap, FILESIZE*sizeof(int));
    return 0;
}
```

## 4.4 Ex4

Intervertir les caractères d'un fichier :

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

int main(int argc, char* argv[]) {
    int i= 0, fd = open(filename, O_RDWR, 0666);
    char* filename = "file.txt";
    struct stat st;

    stat(filename, &st);
    long fileSize = st.st_size;
    char* fileMap = (char*) mmap(NULL, fileSize, PROT_READ | PROT_WRITE,
        MAP_SHARED, fd, 0);

    if(fileMap == (char*) MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);

    while(i < fileSize/2) {
        char c = fileMap[i];
        fileMap[i] = fileMap[fileSize - i-1];
        fileMap[fileSize - i-1] = c;
        i++;
    }

    printf("%s\n", fileMap);
    munmap((void*) fileMap, fileSize);
    return 0;
}
```

## 5 TD5 – Sémaphores

### 5.1 Ex1

Soit 5 philosophes autour d'une table, chacun ayant son assiette et une fourchette. Entre deux pensées un philosophe mange, et pour cela il à besoin de deux fourchettes.

**Version Naïve :**

```
sem_t fourchettes[4];
for(int i= 0; i< 4; i++)
    init(fourchettes[i], 1)

void philosophe(int i) {
    while(1) {
        // Penser
        pause(rand()%1000);

        // Manger
        p(fourchettes[i]);
        p(fourchettes[(i+1)%5]);
        pause(rand()%10);
        v(fourchettes[i]);
        v(fourchettes[(i+1)%5]);
    }
}
```

→ Risque d'interblocage si chacun prend sa gauche en même temps

**Version Améliorée :**

```
sem_t table;
sem_t fourchettes[4];

init(table, 4);
for(int i= 0; i< 4; i++)
    init(fourchettes[i], 1)

void philosophe(i) {
    while(1) {
        // Penser
        pause(rand()%1000);

        // Manger
        p(table);
        p(fourchettes[i]);
        p(fourchettes[(i+1)%5]);
        pause(rand()%10);
        v(fourchettes[i]);
        v(fourchettes[(i+1)%5]);
        v(table);
    }
}
```

## 5.2 Ex2

Problème de lecteurs/rédacteurs

### Priorité lecture

```
Init(file , 1);
Init(mutex, 1);
int nb= 0;

void Writer()
{
    while(1) {
        P(file);
        // Write
        V(file);
    }
}

void Reader()
{
    while(1) {
        P(mutex);
        if(++nb == 1) P(file);
        V(mutex);
        // Read
        P(mutex);
        if(--nb == 0) V(file);
        v(mutex);
    }
}
```

### Priorité Égale

```
Init(file , 1);
Init(mutex, 1);
Init(order, 1);
int nb= 0;

void Writer()
{
    while(1) {
        P(order);
        P(file);
        // Write
        V(file);
        V(order);
    }
}

void Reader()
{
    while(1) {
        P(order);
        P(mutex);
        if(++nb == 1) P(file);
        V(mutex);
        V(order);
        // Read
        P(mutex);
        if(--nb == 0) V(file);
        v(mutex);
    }
}
```

## Priorité Lecture

```
Init(file, 1);
Init(read, 1);
Init(write, 1);
int nb= 0;

void Writer()
{
    while(1) {
        P(write);
        P(file);
        // Write
        V(file);
        V(write);
    }
}

void Reader()
{
    while(1) {
        P(read);
        if(++nb == 1) P(file);
        V(read);
        // Read
        P(read);
        if(--nb == 0) V(file);
        v(read);
    }
}
```

## Priorité Écriture

```
Init(file , 1);
Init(mutex, 1);
Init(order, 1);
int nb= 0;

void Writer()
{
    while(1) {
        P(order);
        P(file);
        // Write
        V(file);
        V(order);
    }
}

void Reader()
{
    while(1)
    {
        P(order);
        P(mutex);
        if(++nb == 1) P(file);
        V(mutex);
        P(order);
        // Read
        P(mutex);
        if(--nb == 0) V(file);
        v(mutex);
    }
}
```

### 5.3 Ex3

Synchronisation de deux processus avec deux sémaphores initialisées à 0.  
Ici A1 et A2 s'exécutent avant B1 et B2 :

Procedure A1;	Procedure A2;
V(S2)	V(S1)
P(S1)	P(S2)
Procedure B1;	Procedure B2;

Pour synchroniser trois processus :

Procedure A1;	Procedure A2;	Procedure A3;
V(S3)	V(S1)	V(S2)
V(S2)	V(S3)	V(S1)
P(S1)	P(S2)	P(S3)
P(S1)	P(S2)	P(S3)
Procedure B1;	Procedure B2;	Procedure B3;

Pour synchroniser N processus :

**Avec N sémaphores**

```
For(i: 1 to N)
  Init(S[i], 0)

Process(i)
  Procedure A[i]

  For(j: 1 to N)
    If(i!=j) V(S[j])

  For(k: 1 to N-1)
    P(S[i])

  Procedure B[i];
```

**Avec 2 sémaphores et un compteur**

```
Init(Mutex, 1)
Init(S, 0)
Compteur = 0

Process(i)
  Procedure A[i]

  P(Mutex)
  Compteur += 1
  If(Compteur = N)
    For(j: 1 to N) V(S)
  V(Mutex)
  P(S)

  Procedure B[i];
```



## 5.4 Ex4

Consommation / Production ordonnée avec tampon circulaire de taille N :

```
Objet tampon[N];
Init(nb_free, N);
Init(nb_full, 0);
int i_free= 0, i_full= 0;

void Producer()
{
    while(1) {
        P(nb_free);

        Produce(buffer[i_full]);
        i_full = (i_full + 1) % N;

        V(nb_full);
    }
}

void Consumer()
{
    while(1) {
        P(nb_full);

        Consume(buffer[i_free]);
        i_free = (i_free + 1) % N;

        V(nb_free);
    }
}
```

## 6 TD6 – Implémentation de Sémaphores

```
#include "sem_pv.h" // stdio.h, stdlib.h, unistd.h
                      // sys/types.h, sys/ipc.h, sys/sem.h

union semun {
    int val;
    struct semids* buf;
    ushort* array;
};

int semid= -1;

int init_sem() {
    if(semid >= 0) return 1;

    if((semid = semget(IPC_PRIVATE, N_SEM, 0600)) < 0) return 2;
    for(int i= 0; i< N_SEM; i++)
        val_sem(i, 0);

    return 0;
}

int del_sem() {
    if(semid < 0) return -1;
    int status = semctl(semid, 0, IPC_RMID);
    if(status >= 0) semid = -1;

    return status;
}

int val_sem(uint sem, int val) {
    if(semid < 0) return -1;
    if(sem >= N_SEM) return -2;

    union semun arg;
    arg.val = val;
    return semctl(semid, sem, SETVAL, arg);
}

int _semop(uint sem, int op) {
    if(semid < 0) return -1;
    if(sem >= N_SEM) return -2;

    struct sembuf buf;
    buf.sem_num = sem;
    buf.sem_op = op;
    buf.sem_flg = 0;
    return semop(semid, &buf, 1);
}


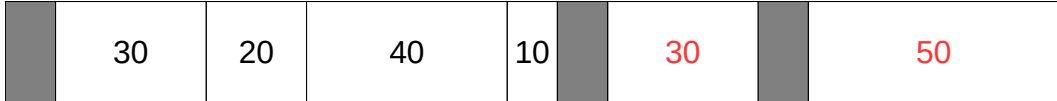

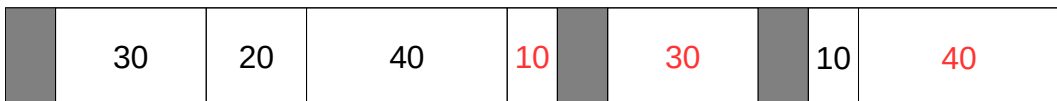
int P(uint sem) { return _semop(sem, -1); }
int V(uint sem) { return _semop(sem, 1); }
```

```
int main(int argc, char* argv[]) {
    init_sem(); val_sem(2, 1);
    P(2);    sleep(10);    V(2);
    del_sem();
}
```

## 7 TD7 – Mémoire

### 7.1 Ex1

Soit mémoire par zones de tailles variables ayant des zones de 100, 30 et 50 de libres. Différentes techniques d'allocation pour  $\{30, 20, 40, 60, 10\}$  :

Base	
First Fit	
Best Fit	
Worst Fit	

### 7.2 Ex2

Soit mémoire paginée de 20 bits, séparée en 2 tel que les bits de 19 à 12 soient de page, et de 11 à 0 d'offset. Elle peut contenir :

- $2^{20}$  mots
- $2^8$  pages
- $2^{12}$  mots par page

### 7.3 Ex3

Sur un système de pagination simple de  $2^{14}$  octets de mémoire physique,  $2^8$  pages d'espace d'adressage logique et une taille de page de  $2^{10}$  octets :

- Une adresse logique fait  $10 + 8 = 18$  bits
- Une case fait  $2^{10}$  bits
- $\frac{2^{14}}{2^{10}} = 2^4$  donc 4 bits de l'adresse physique spécifient la case
- La table de pages contient  $2^8$  entrées
- Une entrée faisant 4 bits, la table de page fait  $4 + 2^8 = 1024$  bits

## 7.4 Ex4

Soit une séquence de références mémoires d'un programme de 460 mots {10, 11, 104, 170, 73, 309, 185, 245, 246, 434, 408, 364}

### 7.4.1

Si une page fait 100 mots, la chaîne de références est {0, 0, 1, 1, 0, 3, 1, 2, 2, 4, 4, 3}

### 7.4.2

FIFO	0	0	0	0	0	3	3	3	3	4	4	4	6 défauts de pages
			1	1	1	1	1	2	2	2	2	3	
	*		*			*		*		*		*	
LRU	0	0	0	0	0	0	1	1	1	4	4	4	7 défauts de pages
			1	1	1	3	3	2	2	2	2	3	
	*		*			*	*	*		*		*	

### 7.4.3

Si un page fait maintenant 200 mots, on a : {0, 0, 0, 0, 0, 1, 0, 1, 1, 2, 2, 1}  
Et 3 défauts de pages avec FIFO et LRU.

## 7.5 Ex5

Dans une mémoire de 600 octets avec des pages de 200 octets :

### Algorithme 1

```
char A[10][100];
for(int i= 0; i< 9; i++)
    for(int j= 0; j< 99; j++)
        A[i][j] = 0;
```

### Algorithme 2

```
char A[10][100];
for(int i= 0; i< 99; i++)
    for(int j= 0; j< 9; j++)
        A[i][j] = 0;
```

Chaîne réf : 0, 1, 2, 3, 4

Défauts Pages : 5 en LRU et Optimal

Chaîne réf : (0, 1, 2, 3, 4) × 100

Défauts Pages : 500 en LRU  
 $4 + \frac{496}{4} \times 2 = 252$  en Optimal

## 7.6 Ex6

Soit un système de gestion de mémoire en siamois de 1Mio :

Action	Mémoire				
Départ	1024				
A = 70K	A	128		256	512
B = 35K	A	B	64	256	512
C = 80K	A	B	64	C	128
Fin A	128	B	64	C	128
D = 60K	128	B	D	C	128
Fin B	128	64	D	C	128
Fin D		256		C	128
Fin C			512		512
Fin					1024

## 8 TD8 – Threads

### 8.1 Ex1

Trois Threads comptent jusque 5 puis se terminent :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define nth 3
#define ifer(is, msg) if(is < 0) perror(msg)
pthread_t threads[nth];

void* th_fonc (void* arg) {
    int n = (int) arg;

    for (int i= 1; i<= 5; i++) {
        printf("From thread %d: i=%d\n", n, i);
        sleep(1);
    }

    return (void*) (n+100);
}

int main(int argn, char* argv[]) {
    // Thread creation
    for(int i=0; i<nth; i++)
        ifer(pthread_create(&threads[i], NULL, th_fonc, (void*) i),
            "Thread creation: ");

    // Thread joining
    for(int i=0; i< nth; i++) {
        void* retval;
        ifer(pthread_join(threads[i], &retval), "Thread join: ");
        printf("Thread ended with %d\n", (int) retval);
    }
}
```

### 8.2 Ex2

Trois threads doivent se retrouver à un point de rdv avant de continuer :

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <time.h>
#include "barx.h"
#define nth 3

pthread_t threads[nth];

int iterationRdv= 10, step= 10, nFinishedThreads=0, curFinishedThread= 0;

pthread_mutex_t mutRdv;
pthread_mutex_t mutDraw;
pthread_mutex_t mutSync;
```

```

pthread_mutex_t mutEnd;

pthread_cond_t condRdv;
pthread_cond_t condFinished;

/* Checks whether all threads are at the meeting point or not */
void rdv()
{
    pthread_mutex_lock(&mutRdv);
    if(++nFinishedThreads < nth)
        pthread_cond_wait(&condRdv, &mutRdv);

    pthread_cond_broadcast(&condRdv);
    pthread_mutex_unlock(&mutRdv);
}

void* th_fonc(void* arg) {
    int n = (int)arg;
    int ypos = 0, mi = 0;
    char *name = "_X_", *color = "yellow";

    switch(n) {
        case 0: ypos = 100;
                mi = 20;
                name = "_0_";
                color = "yellow";
                break;

        case 1: ypos = 135;
                mi = 35;
                name = "_1_";
                color = "white";
                break;

        case 2: ypos = 170;
                mi = 35;
                name = "_2_";
                color = "green";
                break;
    }

    // Drawing the rectangle
    pthread_mutex_lock(&mutDraw);
    drawstr(30, ypos+25, name, strlen(name));
    drawrec(100, ypos, mi*step, 30);
    pthread_mutex_unlock(&mutDraw);

    // Drawing steps
    for(int j = 1; j <= mi; j++) {
        sleep(1 + rand() % 2);

        pthread_mutex_lock(&mutDraw);
        fillrec(100, ypos+2, j*step, 26, color);
        pthread_mutex_unlock(&mutDraw);

        if(j == iterationRdv) {
            printf("thread-%d-rdv\n", n);
            rdv();
            printf("thread-%d-fin-rdv\n", n);
        }
    }
}

```

```

    }
}

pthread_mutex_lock(&mutDraw);
    flushdis();
pthread_mutex_unlock(&mutDraw);

pthread_mutex_lock(&mutSync);

// Signaling main process of the thread's ending
pthread_mutex_lock(&mutEnd);
    curFinishedThread = n;
    pthread_cond_signal(&condFinished);
pthread_mutex_unlock(&mutEnd);

return (void*) (n + 42 + mi + ypos);
}

int main(int argn, char* argv[]) {
    // Initializing the window
    srand(time(NULL));
    initrec();
    drawrec(100 + iterationRdv*step, 50, 4, 180);

    // Initializing mutexes and conds
    pthread_mutex_init(&mutSync, NULL);
    pthread_mutex_init(&mutEnd, NULL);
    pthread_mutex_init(&mutDraw, NULL);
    pthread_mutex_init(&mutRdv, NULL);
    pthread_cond_init (&condFinished, NULL);
    pthread_cond_init (&condRdv, NULL);

    // Creating threads
    for(int i= 0; i< nth; i++) {
        printf("main-debut-thread-%d\n", i);
        pthread_create(&threads[i], NULL, th_fonc, (void*) i);
    }

    // Waiting for threads (they signal it themselves with condFinished
    // and curFinishedThread)
    for(int i= 0; i< nth; i++) {
        pthread_mutex_lock(&mutEnd);
        if(i > 0)
            pthread_mutex_unlock(&mutSync);

        pthread_cond_wait(&condFinished, &mutEnd);
        int j = curFinishedThread;
        pthread_mutex_unlock(&mutEnd);

        // Getting the return value
        void* val;
        pthread_join(threads[j], &val);
        printf("main-fin-thread-%d-avec-val=%d\n", j, (int) val);
    }

    printf("main-tout-est-terme\n");
    detruitrec();
    return 0;
}

```



## 8.3 Bonus

Exemple de cours :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define BUFFERSIZE 4

int buf[BUFFERSIZE];
int readpos= 0, writepos= 0, count= 0;

pthread_t pth, cth;
pthread_cond_t notempty;
pthread_cond_t notfull;
pthread_mutex_t lck;

void produce(int* x) {
    sleep(1);
    x++;
    printf("Produce: %3d\tcount= %3d\n", *x, count);
}

void consume(int* x) {
    sleep(2);
    printf("Consume: %3d\tcount= %3d\n", *x, count);
}

void* producer(void* data) {
    int item= 0;

    while(1) {
        produce(&item);
        pthread_mutex_lock(&lck);

        while(count == BUFFERSIZE) {
            printf("Producer locked\n");
            pthread_cond_wait(&notfull, &lck);
            printf("Producer unlocked");
        }

        buf[writepos] = item;
        count++;
        writepos = (writepos+1) % BUFFERSIZE;

        pthread_cond_signal(&notempty);
        pthread_mutex_unlock(&lck);
    }
}

void* consumer(void* data) {
    int item;

    while(1) {
        pthread_mutex_lock(&lck);

        while(count == 0) {
```

```

        printf("Consumer locked\n");
        pthread_cond_wait(&notempty, &lck);
        printf("Consumer unlocked\n");
    }

    item = buf[readpos];
    count--;
    readpos = (readpos+1) % BUFFERSIZE;

    if(count == BUFFERSIZE-1)
        pthread_cond_signal(&notfull);

    pthread_mutex_unlock(&lck);
    consume(&item);
}

}

int main(int argn, char* argv[]) {
    pthread_cond_init(&notempty, NULL);
    pthread_cond_init(&notfull, NULL);
    pthread_mutex_init(&lck, NULL);

    pthread_create(&pth, NULL, producer, NULL);
    pthread_create(&cth, NULL, consumer, NULL);

    sleep(15);
}

```

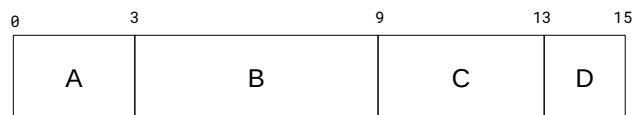
## 9 TD9 – Ordonnancement

### 9.1 Ex1

Graphes d'exécution pour les processus suivants :

Processus	Arrivée	Exécution
A	0	3
B	1.001	6
C	4.001	4
D	6.001	2

#### 9.1.1 First Come First Served



Temps Rotation Moyen :  $\frac{3+6+4+2}{4} = 3.75$

Temps Traitement Moyen :  $\frac{(3-0)+(9-1.001)+(13-4.001)+(15-6.001)}{4} = 7.25$

#### 9.1.2 Shortest Job First



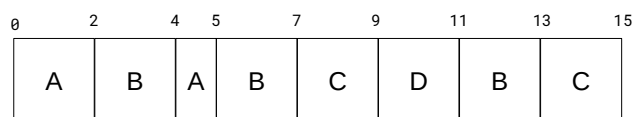
Temps Rotation Moyen :  $\frac{3+6+2+4}{4} = 3.75$

Temps Traitement Moyen :  $\frac{(3-0)+(9-1.001)+(15-4.001)+(13-6.001)}{4} = 7.25$

#### 9.1.3 Shortest Remaining Time (Quantum = 1)

Pareil que SJF

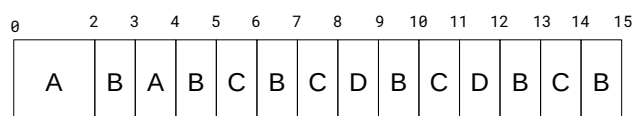
#### 9.1.4 Round Robin (Quantum = 2)



Temps Rotation Moyen :  $\frac{2+2+1+2+2+2+2+2}{8} = 1.875$

Temps Traitement Moyen :  $\frac{(5-0)+(13-1.001)+(15-4.001)+(11-6.001)}{4} = 8.25$

#### 9.1.5 Round Robin (Quantum = 1)



Temps Rotation Moyen :  $\frac{2+1+1+1+1+1+1+1+1+1+1+1+1+1}{14} = 1.14$

Temps Traitement Moyen :  $\frac{(4-0)+(15-1.001)+(14-4.001)+(12-6.001)}{4} = 8.5$

## 9.2 Ex2

Soit 7 processus possédant une priorité :  $F_1 < F_2 < F_3$

On passe aux processus de la file inférieure une fois la courante terminée ( $\rightarrow$  Risque de famine)

Ici les algorithmes s'appliquent aux 3 files

Processus	Date Arrivée	Temps Exécution	Priorité
$P_1$	0	7	2
$P_2$	0	4	3
$P_3$	1	6	1
$P_4$	1	1	2
$P_5$	1	2	3
$P_6$	2	4	1
$P_7$	2	1	2

$F_1$  :  $P_3, P_6$   
 $F_2$  :  $P_1, P_4, P_7$   
 $F_3$  :  $P_2, P_5$

### 9.2.1 Shortest Job First

0	4	6	7	8	15	19	25
2	5	4	7	1	6	3	

Temps Rotation Moyen :  $\frac{4+2+1+1+7+4+6}{7} = 3.57$

Temps Traitement Moyen :  $\frac{(4-0)+(6-1)+(7-1)+(8-2)+(15-0)+(19-2)+(25-1)}{7} = 11$

### 9.2.2 Round Robin (Quantum = 2)

0	2	4	6	8	9	10	15	17	19	21	23	25
2	5	2	1	4	7	1	3	6	3	6	3	

Temps Rotation Moyen :  $\frac{2+2+2+2+1+1+5+2+2+2+2+2}{12} = 2.08$

Temps Traitement Moyen :  $\frac{(6-0)+(4-1)+(15-0)+(9-1)+(10-2)+(23-2)+(25-1)}{7} = 12.14$

### 9.2.3 Shortest Remaining Time (Quantum = 1)

0	1	3	6	7	8	15	19	25
2	5	2	4	7	1	6	3	

Temps Rotation Moyen :  $\frac{1+2+3+1+1+7+4+6}{8} = 3.125$

Temps Traitement Moyen :  $\frac{(6-0)+(3-1)+(7-1)+(8-2)+(15-0)+(19-2)+(25-1)}{7} = 10.86$

### 9.3 Ex3

Soit un système d'ordonnancement tel que chaque processus ait une priorité de base. Toutes les secondes, la priorité est recalculée selon

$$Prio = Prio_{Base} + \frac{t_{occupation\ cpu}}{2}$$

Les processus ayant la plus haute priorité sont exécutés d'abord (et sont départagés par FCFS) :

Processus	Date Arrivée	Temps Exécution	Priorité Base
$P_1$	0	4	2
$P_2$	1	4	3
$P_3$	1	3	1
$P_4$	4	2	5
$P_5$	5	2	1
$P_6$	6	2	1

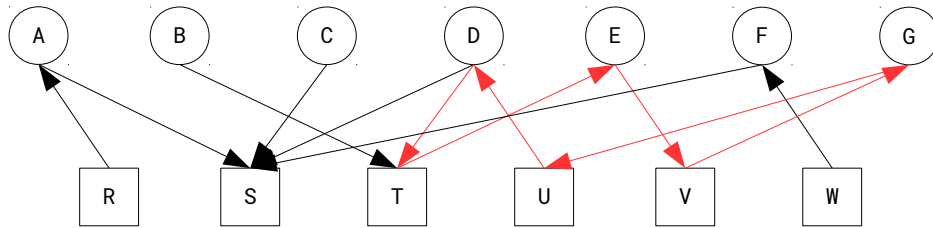
0	1	4	6	7	8	13	15	17
1	2	4	2	1	3	5	6	

Temps Rotation Moyen :  $\frac{1+3+2+1+1+3+3+2+2}{8} = 3.125$

Temps Traitement Moyen :  $\frac{(6-0)+(3-1)+(7-1)+(8-2)+(15-0)+(19-2)+(25-1)}{7} = 10.86$

## 10 TD10 – Interblocage

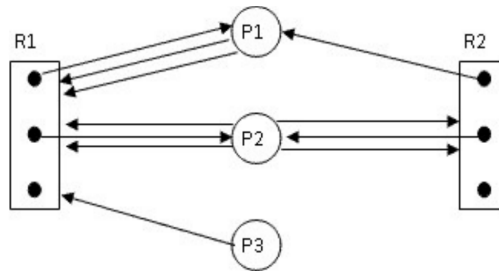
### 10.1



Cycle  $D \rightarrow T \rightarrow E \rightarrow V \rightarrow G \rightarrow U \rightarrow D$  Donc interblocage

### 10.2

Y'a t-il interblocage pour le graphe de dépendance suivant ?



Demande			Allocation			Disponible		Capacité	
	R1	R2		R1	R2	R1	R2	R1	R2
P1	2	0	P1	1	1	1	1	3	3
P2	2	2	P2	1	1				
P3	1	0	P3	0	0				

Soit le sous ensemble non-vide  $D$  de  $P_i$ ,  $D = \{P_1, P_2\}$

D'après

$$Disponible[i] + \sum_{j \notin D} Allocation[j] \leq Demande[i] \quad \text{ou} \quad Disponible \neq N - \sum_{j=1}^n Allocation[j]$$

$$demande[1] = [2, 0] \not\leq [1, 1] + [0, 0]$$

$$demande[2] = [2, 2] \not\leq [1, 1] + [0, 0]$$

→ Interblocage pour  $D = \{P_1, P_2\}$

## 10.3

Max					Allocation					Besoin				
	R0	R1	R2	R3		R0	R1	R2	R3		R0	R1	R2	R3
P0	0	0	1	2	P0	0	0	1	2	P0	0	0	0	0
P1	1	7	5	0	P1	1	0	0	0	P1	0	7	5	0
P2	2	3	5	6	P2	1	3	5	4	P2	1	0	0	2
P3	0	6	5	2	P3	0	6	3	2	P3	0	0	2	0
P4	1	6	5	6	P4	0	0	1	4	P4	1	6	4	2

### Disponible

R0	R1	R2	R3
1	5	2	0

### 10.3.1 Algorithmme

```
def sane_state():
    work = list(free)
    finished = [False] * N

    while True:
        cur = [i for i in range(N) if not finished[i] and need[i] <= work]
        if len(cur) == 0:
            break

        i = cur[0]
        work += allocated[i]
        finished[i] = True;

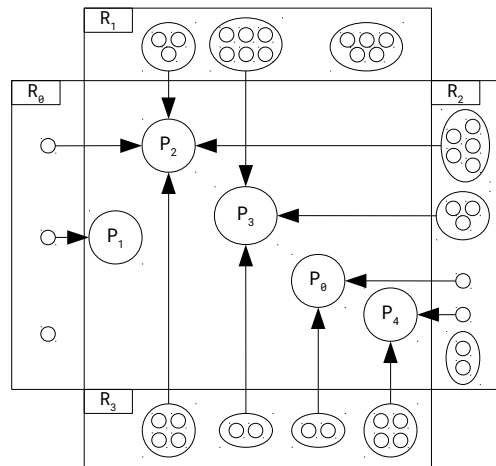
    return all(finished)

def request(req, need):
    if req[i] > need[i]:
        raise Error("Max request exeeded")

    if req[i] > free:
        wait()

    free -= req[i]
    allocated[i] += req[i]
    need[i] -= req[i]

    if sane_state():
        return True
    else:
        # Backtrack
        avaitable += req[i]
        allocated[i] -= req[i]
        need[i] += req[i]
```



Demande[1] = [0, 4, 2, 0]

algo\_requ :

- 1 : Ok
  - 2 : Ok
  - 3 : accepter et verifier
- Travail = [1, 1, 0, 0]  
Fini = [F, F, F, F, F]

i = 0    Travail = [1, 1, 1, 2]  
          Fini[0] = V

i = 2    Travail = [2, 4, 6, 6]  
          Fini[1] = V

i = 1    Travail = [3, 8, 8, 6]  
          Fini[2] = V

i = 3    Travail = [3, 14, 11, 8]  
          Fini[2] = V

i = 4    Travail = [3, 14, 12, 12]  
          Fini[2] = V

Demande[4] = [1, 0, 0, 0]

algo\_requ :

- 1 : Ok
  - 2 : Ok
  - 3 : accepter et verifier
- Travail = [0, 1, 0, 0]  
Fini = [F, F, F, F, F]

i = 0    Travail = [0, 1, 1, 2]  
          Fini[0] = V

Pas possible  
Retirer la demande



## 10.4

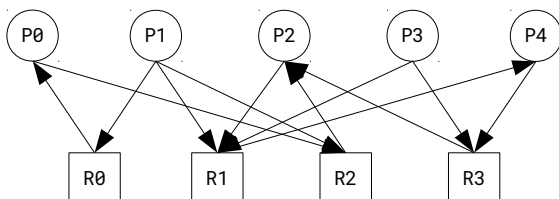
Soit l'état des allocations de ressources d'un système :

Allocation					Demande					Disponible				
	R0	R1	R2	R3		R0	R1	R2	R3	R0	R1	R2	R3	
P0	1	0	0	0	P0	0	0	1	0	0	0	0	0	
P1	0	0	0	0	P1	1	1	1	0					
P2	0	0	1	1	P2	0	1	0	0					
P3	0	0	0	0	P3	0	1	0	1					
P4	0	1	0	0	P4	0	0	0	1					

### 10.4.1 Algorithmme

```
def coffman(need, allocated, free):  
    work = list(avaitable)  
    finished = [all([y==0 for y in x]) for x in allocated]  
  
    while True:  
        cur = [i for i in range(N) if not free[i] and need[i] <= work]  
        if len(cur) == 0:  
            break  
  
        i = cur[0]  
  
        work += allocated[i]  
        # Work and end of process  
        finished[i] = True  
  
    if all(finished):  
        return True  
    else:  
        raise Error("Interblocage")
```

Graphe Allocation



Graphe Attentes

