

g

- Tout document autorisé
- Les réponses doivent être **claires et concises** : cela sera pris en compte lors de l'évaluation
- La durée de l'examen est 90 minutes

### Exercice 1 (6 pts)

Pour le programme ci-dessous, quelle sortie serait imprimée lors de son exécution ? Supposons que tous les appels de fonction, de bibliothèque et de système réussissent.

```

int x = 42;

main()
{
    int rc, status;
    rc = fork();
    if (rc == 0) {
        func1(); exit(1);
    }
    else {
        rc = waitpid(rc, &status, 0);
        printf("R: %d", WEXITSTATUS(status));
        printf("M: %d\n", x);
        x = 100;
        printf("P: %d\n", x);
        _exit(2);
    }
    func2();
}

void func1()
{
    int rc, status;
    printf("T: %d\n", x);
    x = 10;
    rc = fork();
    if (rc == 0) {
        x = 50;
        printf("Q: %d\n", x);
        exit(3);
    }
    rc = waitpid(rc, &status, 0)
    printf("A: %d\n", x);
    printf("D: %d", WEXITSTATUS(status));
    exit(4);
}

void func2()
{ printf("C: %d\n", x); }
```

T: 42

Q: 50

A: 10

D: 3

R: 4

M: 42

P: 100

### Exercice 2 (3 pts)

Supposons qu'un système d'exploitation utilise une stratégie à temps partagé avec priorité avec un quantum de 500 millisecondes. L'horloge matérielle génère une interruption de temporisateur chaque milliseconde. Supposons qu'un processus P est programmé pour s'exécuter et est distribué. Pendant son quantum, P ne fait aucun appel système. Cependant, les exceptions de traduction d'adresse se produisent une fois toutes les 10 millisecondes pendant que P est en cours d'exécution. Combien de fois au cours de son quantum le processus P entre-t-il dans le mode noyau ? Expliquez brièvement votre réponse.

**Chaque exception et interruption fait entrer le contrôle dans le noyau. Il y aura 500 interruptions de minuterie pendant le quantum de P, et il y aura  $500/10 = 50$  exceptions de traduction d'adresse, pour un total de 550 entrées de noyau.**

### Exercice 3 (11 pts)

*On se propose d'étudier une structuration pour des disques dont la capacité est comprise entre 100 Mo et 2 Go., les secteurs étant de 4096 octets.*

Les machines pour lesquelles ils sont destinés manipulent des octets, des entiers sur 16 bits ou des entiers sur 32 bits.

Les concepteurs du système de gestion de fichiers ont décidé de diviser chaque unité du disque de la façon suivante :

- *le descripteur du disque*, qui contient tous les renseignements nécessaires à son identification, les paramètres variables de la structure, et une table donnant accès aux fichiers.
- *la table d'allocation de l'espace*, représentant l'espace libre sur le disque.
- *la zone des fichiers*, qui contient les descripteurs de fichiers et l'espace de données de ces fichiers.

Un fichier est constitué de deux parties:

- *le descripteur de fichier* qui contient toutes les informations nécessaires à sa localisation, et qui seront détaillées ci-dessous.
- *l'ensemble des zones physiques*, qui contiennent les données du fichier. Pour un fichier, toutes les zones physiques du fichier ont la même taille, et sont constituées d'un nombre entier de secteurs contigus. Par contre les différentes zones d'un même fichier ne sont pas nécessairement contiguës.

Les descripteurs de fichiers doivent repérer l'ensemble des zones du fichier. Pour permettre à un fichier de contenir un nombre quelconque de zones, tout en ayant des descripteurs de taille fixe, les descripteurs sont décomposés en une partie principale (obligatoire) et des parties extensions (optionnelles). Les parties extensions éventuelles ont la même structure que les parties principales, les informations inutilisées étant simplement mises à 0 dans ce cas. Par la suite nous appellerons descripteur cette structure, qu'elle soit utilisée comme partie principale ou comme extension. Les descripteurs de fichiers sont regroupés dans des secteurs du disque. Ces secteurs sont chaînés entre eux, pour permettre la recherche des fichiers. Un descripteur de fichier a la structure suivante :

- le *nom* du fichier sur 16 octets,
- des informations diverses sur 12 octets,
- la *longueur du fichier*, c'est-à-dire le nombre total d'octets du fichier,
- le *nombre total de zones* du fichier,
- la *taille des zones* en nombre de secteurs,
- l'*"adresse"* de l'*extension suivante* (numéro de secteur et position dans le secteur),
- l'*"adresse"* de la dernière extension du fichier,
- la *table des numéros de premiers secteurs* des premières zones.

A.1- En considérant que tout entier sur disque a la même représentation qu'en mémoire centrale, c'est-à-dire 16 ou 32 bits, déterminer la taille en octets nécessaire à la représentation de chacune des informations suivantes:

- un numéro de secteur,
- la longueur du fichier, c'est-à-dire le nombre total d'octets du fichier,
- le nombre total de zones du fichier,
- la taille des zones en nombre de secteurs,
- l'*"adresse"* d'une extension (numéro de secteur et position dans le secteur).

- *numéro de secteur*. Un disque peut contenir jusqu'à  $2*2^{30} / 4*2^{10} = 2^{19}$  secteurs. Il faut donc au moins 19 bits pour représenter un numéro de secteur, soit un entier sur 32 bits, ou 4 octets.
- *longueur du fichier*. Un fichier ne peut dépasser la taille maximale d'un disque, donc  $2 * 2^{30} = 2^{31}$ . Il faut donc des entiers sur 32 bits, soit 4 octets.
- *nombre total de zones et taille des zones*. En l'absence de contrainte, une zone physique d'un fichier doit être d'au moins 1 secteur, mais pouvoir atteindre presque le disque complet (à

l'exception des informations structurelles). Un fichier peut donc avoir  $2^{19}$  zones de 1 secteur; il faut donc 32 bits ou 4 octets pour représenter le nombre total de zones du fichier. De même, un fichier peut avoir 1 zone de  $2^{19}$  secteurs; il faut donc 32 bits ou 4 octets pour représenter la taille des zones du fichier. Notons qu'il ne serait pas très contraignant et sans doute assez logique de limiter chacune de ces quantités à  $2^{16}$ , permettant ainsi d'utiliser des entiers sur 16 bits ou 2 octets, puisque ceci nous donne de toute façon  $2^{32}$  secteurs, valeur nettement supérieure au nombre maximal de secteurs d'un disque.

- *adresse d'une extension.* L'adresse d'une partie extension d'un descripteur doit permettre de désigner d'une part le numéro de secteur disque où la partie extension est placée (19 bits), et d'autre part, sa position dans ce secteur. Celle-ci peut être définie simplement par le numéro de son premier octet (12 bits), donnant ainsi un total de 31 bits. Cette position peut aussi être le numéro d'ordre du descripteur dans le secteur, puisque tous les descripteurs sont de même taille. Par exemple, s'il y a au plus 32 descripteurs par secteur, 5 bits suffiront alors pour ce numéro, et une adresse de partie extension nécessitera 24 bits. Enfin, en suivant la politique de représentation des entiers en 16 ou 32 bits, on constate alors qu'il faut 32 bits ou 4 octets dans tous les cas.

A.2- Déterminer la taille d'un descripteur en fonction du nombre d'entrées q de la table des premiers secteurs de zones dans un descripteur. En déduire la valeur maximale du nombre r de descripteurs dans les secteurs du disque qui les contiennent, en fonction de q, et la perte qui résulte d'un choix donné de r et q compatibles.

Pour déterminer le nombre de descripteurs de fichiers par secteur, il faut déterminer la taille d'un descripteur. Nous reprenons les données numériques de la question A.1, et les notons devant les informations du descripteur de fichier:

- 16 le nom du fichier sur 16 octets,
- 12 des informations diverses sur 12 octets,
- 4 le nombre total d'octets du fichier,
- 4 le nombre total de zones du fichier,
- 4 la taille des zones en nombre de secteurs,
- 4 l'“adresse” de l'extension suivante (numéro de secteur et position dans le secteur),
- 4 l'“adresse” de la dernière extension du fichier,
- $4 * q$  la table des numéros de premiers secteurs des premières zones.

Il s'ensuit que le descripteur occupe  $48 + 4 * q$  octets, où q est le nombre d'entrées de la table des premiers secteurs de blocs d'un descripteur. Les secteurs contenant des descripteurs étant chaînés entre eux, le lien de chaînage occupe 4 octets, laissant libres 4092 octets pour r descripteurs, et il s'ensuit que  $r \leq 4092 / (48 + 4 * q)$ . Pour une valeur de r compatible avec q, la perte est  $4092 - r * (48 + 4 * q)$ .

A.3- Étudier en particulier les cas où ce nombre q prend les valeurs 31, 32 et 33. Calculer la perte d'espace dans chaque cas. Commenter le choix de 32.

Nous avons les valeurs suivantes:

q	r maximum	perte
31	23	136

32	23	44
33	22	132

Le choix de 32 est le plus optimal, puisqu'il correspond à la plus petite perte. Cependant, même s'il induisait une perte supérieure aux autres, il pourrait être conservé car il a l'avantage de permettre de représenter, dans chaque descripteur, qu'il soit partie principale ou extension, un nombre de zones qui est une puissance de 2.

A.4- Expliquer l'intérêt des différentes informations (autres que diverses) qui sont présentent dans la partie principale d'un descripteur. On distinguera celles qui sont nécessaires et celles qui sont redondantes, et on justifiera cette redondance. Montrer que l'on peut déduire de ces informations le nombre d'extensions du descripteur de fichier.

Le *nom* du fichier est nécessaire pour distinguer les fichiers entre eux, et permet à l'utilisateur de désigner ce fichier par une chaîne de caractères. La *longueur du fichier* N est la seule information de taille qui permette de savoir exactement quels sont les octets de l'espace alloué au fichier qui en font partie. La *taille des zones* T est nécessaire pour déterminer la taille des zones à allouer au fur et à mesure des besoins au fichier; elle permet également de déterminer à quelle zone appartient le n<sup>ième</sup> secteur du fichier. L'*adresse de l'extension suivante* est nécessaire pour parcourir les extensions du descripteur. La *table des numéros de premiers secteurs* permet de savoir où commence chaque zone sur disque, et donc détermine l'espace alloué au fichier.

Le *nombre total de zones* du fichier est une information redondante, en ce sens qu'elle peut être reconstruite à partir des autres. En effet, le nombre de secteurs S est égal à  $(N-1)\%4096+1$ , où % désigne la division entière. À partir de la taille des zones T, on peut déduire le nombre de zones  $B = (S-1)\%T+1$ . Elle évite de refaire ce calcul en particulier pour déterminer la taille de l'espace occupé par le fichier sur le disque. Enfin l'*adresse de la dernière extension* du fichier est également une information redondante, puisqu'elle peut être connue en parcourant les extensions successives. Elle est utile car l'allocation d'une nouvelle zone lors d'un allongement du fichier, entraîne la modification de cette extension.

Par ailleurs  $(B-1)\%32$  nous donne le nombre d'extensions du fichier.

## Introduction

- Seuls les documents de cours sous format papier sont autorisés
- Répondez sur le sujet de l'examen
- Les réponses doivent être **claires et concises** : cela sera pris en compte lors de l'évaluation des réponses
- La durée de l'examen est 120 minutes.

## Partie 1 (10 points) --- Copie 1

- 1) Citer deux inconvénients de la multiprogrammation et deux avantages des systèmes à temps partagé. **(1 point)**

- Inconvénients de la multiprogrammation : Gestion de la mémoire, le prob de sécurité des processus, il faut un ordonnanceur ... ect.
- Avantages des systèmes à temps partagé : Utilisation efficace du CPU, Temps de traitement petit pour les programmes courts.

- 2) L'accès direct à la mémoire est utilisé pour les périphériques d'E/S à grande vitesse afin d'éviter la surcharge du processeur.

- a) Comment le processeur communique-t-il avec le périphérique pour coordonner le transfert ? **(0,75 point)**

La CPU peut lancer une opération DMA en écrivant des valeurs dans des registres spéciaux auxquels le périphérique peut accéder indépendamment.

- b) Comment le processeur sait-il que les opérations d'E/S sont terminées? **(0,75 point)**

Lorsque le périphérique a terminé son opération, il interrompt la CPU pour indiquer la fin de l'opération.

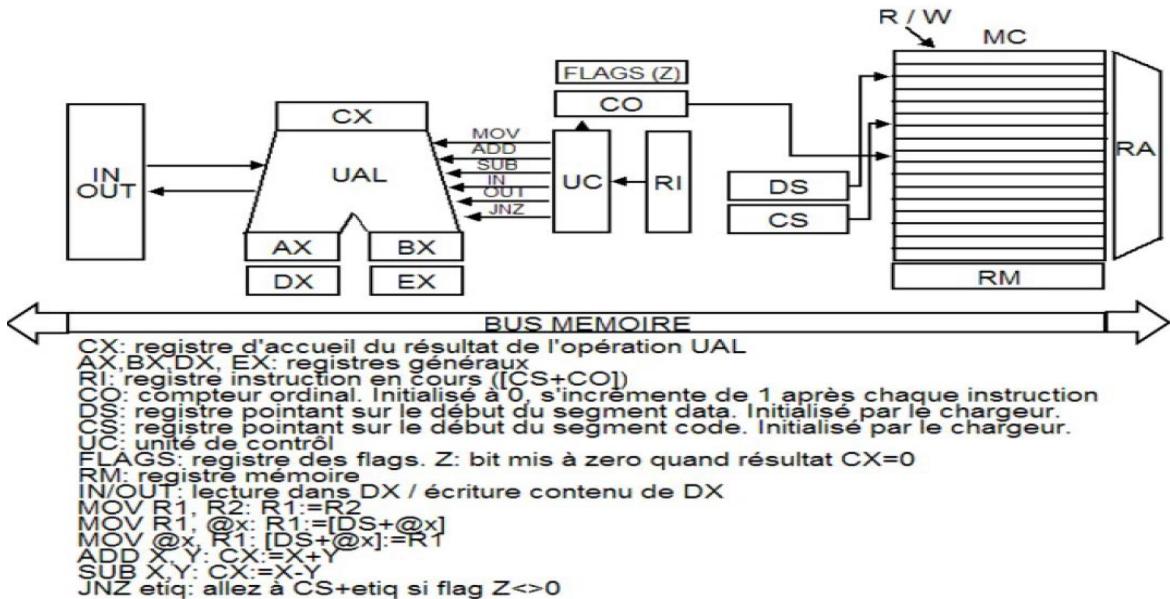
- 3) Un processeur ne traite que les interruptions et il a une capacité de lire ou écrire un mot mémoire de 4 octets en  $10\text{ns}$ . Lors d'une interruption il faut recopier 32 registres dans la pile du processus. Quel est le nombre maximal d'interruptions que ce processeur peut traiter en  $2\text{ms}$  ? **(1,5 points)**

*Le temps d'une interruption est  $2 \times 32 \times 10\text{ns} = 640\text{ ns}$ . Le nombre d'interruptions qu'il peut traiter en  $2\text{ms}$  est  $2000000 / 640 = 31225$ .*

- 4) Que pouvez-vous dire d'un système dans lequel lorsqu'un processus  $P$  crée un processus fils, l'exécution du processus  $P$  est suspendu jusqu'à la terminaison du processus fils ? (1 point)

*Système mono-programmé*

- 5) Soit l'architecture suivante :



- a) Dans cette architecture, peut-on exécuter un programme sans stocker ses données en mémoire centrale (MC) ? (0,5 point)

*Oui : on peut utiliser les registres si le nombre de données est petit.*

- b) Dans cette architecture, peut-on exécuter un programme sans stocker ses instructions en mémoire centrale (MC) ? (0,5 point)

*Non. Le processus est installé dans la MC.*

- c) Citer deux types d'entité de cette architecture qui peuvent envoyer un signal (0,5 point)

*Processus, processeur.*

- d) Citer un exemple d'entité de cette architecture qui peut générer une interruption (0,5 point)

*E/S*

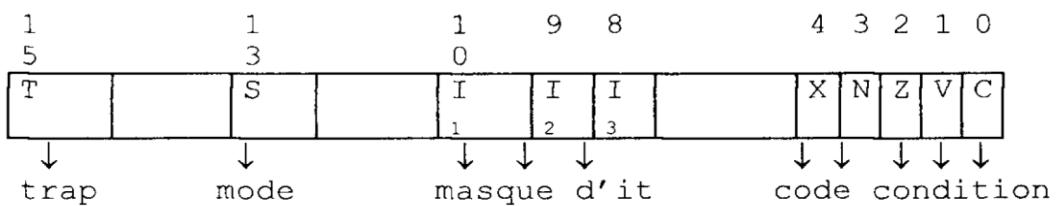
- a) Citer les avantages et les inconvénients de cette architecture (**0,5 point**)

*Il s'agit d'un type de l'architecture de Von Neumann. L'inconvénient est les goulots d'étranglements.*

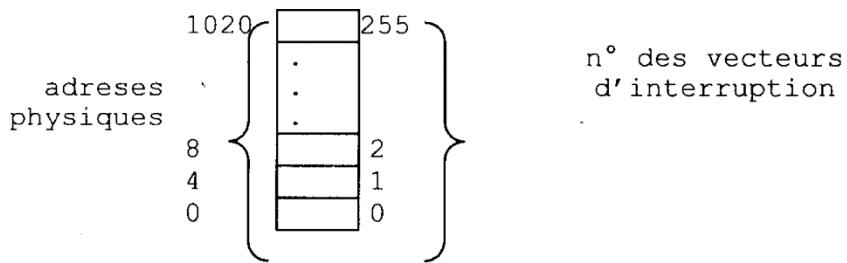
- b) Proposer une autre architecture qui répond aux inconvénients de cette architecture et discuter les caractéristiques de cette nouvelle architecture ? (**1 point**)

*Architecture de Harvard.*

- 6) Le contexte d'un programme s'exécutant sur une machine X est constitué de 16 registres généraux (R0 à R15), du compteur ordinal (CO) et du PSW dont la structure est la suivante :



Cette machine reconnaît 256 causes d'interruption (it) dont les vecteurs sont stockés dans la table suivante:



Le vecteur d'interruption contient le compteur ordinal et la routine d'interruption correspondante. Parmi ces 256 causes d'interruption huit causes sont dues à des organes externes. Chacune de ces huit causes possède un numéro d'interruption qui correspond à sa priorité (de 0 à 7). Une interruption de ce type arrive de la manière suivante :

- i. L'unité centrale ne prend en considération un signal d'interruption que si le numéro de celle-ci est supérieur à la valeur dumasque d'it qui se trouve dans le PSW
- ii. L'unité centrale envoie un signal d'acquittement au composant qui a déclenché l'interruption
- iii. Le composant envoie alors le numéro de son vecteur d'it
- iv. L'unité centrale récupère ce numéro et empile dans la pile système le CO et le PSW du programme courant
- v. L'unité centrale positionne le bit S du PSW à 1 et le masque d'it à la valeur correspondante au numéro d'it récupéré en I
- vi. L'unité centrale se branche sur la routine d'it correspondante

La machine dispose de l'instruction RTI qui charge le PSW et le CO à partir des deux mots de tête de la pile système.

- a) Comment le système se branche vers la routine d'it  $i$ ,  $0 \leq i \leq 255$  (**0,5 point**)

*Co = i x 4;*

- b) Donner le corps d'une routine d'it quelconque (**0,5 point**)

*SR0=R0, ..... SR15=R15 ;  
Traitement ;  
R0=SR0, ..... R15=SR15;  
RTI;*

- c) Donner les valeurs du masque d'it du PSW pour les 8 causes d'it (**0,5 point**)

*000  
001  
010  
011  
101  
110  
100  
111*

**Correction de la partie 2 du  
median SR02-P2019**

## 1 Solution Exercice 1 (3.5 points)

### 1.1 Question 1

Donner les bouts de code (3) et (4) nécessaires permettant d'installer des nouveaux handlers pour le traitement d'un signal SIGUSR1. **(1 point)**

### 1.2 Solution 1

```

1 struct sigaction old, new;
2 new.sa_handler=handler_pere;
3 sigaction(SIGUSR1,&new,&old);

```

Listing 1: installation du handler pour le père

```

1 struct sigaction new_fils;
2 new_fils.sa_handler = handler_fils;
3 sigaction(SIGUSR1,&new_fils,NULL);

```

Listing 2: installation du handler pour le fils

### 1.3 Question 2

Donner les bouts de code (5) et (6) nécessaires pour que les deux processus puissent synchroniser l'exécution tel qu'il est décrit dans l'énoncé. **(1 point)**

### 1.4 Solution 2

```

1 while(1){ // .... (5)
2     fibonacci(&n, &t1, &t2);
3     kill(getppid(), SIGUSR1); // envoi du signal SIGUSR1 au parent.
4     pause(); // se mettre en pause
5 }

```

Listing 3: Code du processus fils

```

1 while(1){ // .... (6)
2     pause();
3 }

```

Listing 4: Code du processus père

## 1.5 Question 3

Donner les bouts de codes (1) et (2) correspondants aux fonctions handler\_pere() et handler\_fils() qui serviront des handlers pour le traitement du signal SIGUSR1. (1.5 point)

## 1.6 Solution 3

```

1 void handler_fils(int sig){
2     nb_sig++;
3     if (nb_sig == MAX_SIGUSR1){
4         kill(getppid(), SIGINT);
5         exit(0);
6     }
7 }
8
9 void handler_pere(int sig){
10    sleep(5);
11    kill(pid_fils, SIGUSR1);
12 }
```

Listing 5: Exercice 3

## 2 Solution Exercice 2 (6.5 points)

### 2.1 Question 1

Expliquer pourquoi il faut partager le pid du processus p2 dans une mémoire partagée (0.5 points)

### 2.2 Solution 1

Le processus p1 a besoin du PID du processus p2 pour lui envoyer un signal. Le seul moyen pour que p1 connaisse le PID de p2 c'est d'utiliser une mémoire partagée pour stocker les PID de p2.

### 2.3 Question 2

Ecrire la fonction initab(int n, int fd) qui permet d'initialiser aléatoirement un tableau de n entiers et copier son contenu dans le fichier représenté par le descripteur fd. (0.5 points)

## 2.4 Solution 2

```

1 void initab(int n, int fd){
2     srand(time(NULL));
3     int i;
4     for(i=0; i<n; i++){
5         int a = rand()%30+10;
6         write(fd,&a,sizeof(a));
7     }
8 }
```

Listing 6: initab()

## 2.5 Question 3

Ecrire la fonction showtab(int n, int \* tab) qui permet d'afficher le contenu du tableau tab (projeté en mémoire par la primitive mmap). **(0.5 points)**

## 2.6 Solution 3

```

1 void showtab(int n, int *tab){
2     int i;
3     for(i=0; i<n; i++){
4         printf("\n\ttab[%d]=%d",i, tab[i]); fflush(stdout);
5     }
6 }
```

Listing 7: showtab()

## 2.7 Question 4

Ecrire les routines de traitement du signal SIGUSR1 handlers\_pere(), handler\_fils1(), et handlers\_fils2() liées aux processus parent, p1 et p2 respectivement. **(2.5 points)**

## 2.8 Solution 4

```

1 void handler_fils1(int sig){
2     showtab(n, tab);
3     int i;
4     for(i=0; i<n; i++){
5         if (!(tab[i]%2)) tab[i]+= 1;
6     }
7     kill(pids[1], SIGUSR1);
8 }
```

```

10 /* fonction utilisateur de comparaison fournie a qsort() */
11 static int compare (void const *a, void const *b)
12 {
13     int const *pa = a;
14     int const *pb = b;
15     return *pa - *pb;
16 }
17
18 void handler_fils2(int sig){
19     showtab(n, tab);
20     qsort(tab, n, sizeof(int), compare);
21     kill(getppid(), SIGUSR1);
22 }
23
24 void handler_pere(int sig){
25     showtab(n, tab);
26     shmdt(pids); // detacher le segment memoire du processus parent
27     shmctl(shmid, IPC_RMID, NULL); // supprimer le segment memoire
28     munmap((void *) tab, n*sizeof(int));
29 }
```

Listing 8: Les handlers

## 2.9 Question 5

Ecrire le programme principal permettant de créer les deux processus p1 et p2 et réaliser l'objectif décrit ci-dessus, en faisant appel aux différentes fonctions : initab, showtab, etc. (2.5 points)

## 2.10 Solution 5

```

1 // Les variables globales
2 int *tab, *pids, n, fd, shmid;
3
4 int main(int argc, char* argv[]){
5     if(argc < 3) {perror("\nUsage: <filepath> <n>"); exit(-1);}
6     struct sigaction new;
7     new.sa_handler=handler_pere;
8     sigaction(SIGUSR1,&new,NULL);
9     n = atoi(argv[2]);
10    fd = open(argv[1], O_RDWR|O_CREAT, 0666);
11    initab(n, fd);
12    tab = (int *)mmap(NULL, n*sizeof(int), PROT_READ|PROT_WRITE,
13                      MAP_SHARED, fd, 0);
14    if(tab == (int*)MAP_FAILED){
15        perror("Erreur mmap!!"); exit(1);
16    }
17    close(fd);
```

```

17    showtab(n,tab);
18    int p;
19    if ((shmid = shmget(IPC_PRIVATE, 2*sizeof(int), PERM)) == -1) {
20        perror("Echec de creation du segment");
21    return 1;
22}
23    if ((pids = (int *)shmat(shmid, NULL, 0)) == (void *)-1) {
24        perror("Echec de l'attachement du segment");
25    }
26    if((p=fork())==-1){ perror("\n(parent) Echec de creation du
27        processus p1"); return 1;}
28    if(p > 0){ //parent
29        pids[0] = p; // sauvegarder le pid du fils 1 dans la memoire
30        partagee.
31        if((p=fork())==-1){ perror("\nEchec de creation du processus p2"
32        ); return 1;}
33        if(!p){ //fils 2
34            struct sigaction new;
35            new.sa_handler=handler_fils2;
36            sigaction(SIGUSR1,&new,NULL);
37            sleep(3);
38            pause();
39            exit(0);
40        }
41        pids[1]=p;
42        sleep(1);
43        kill(pids[0], SIGUSR1);
44        int i;
45        pause();
46    }
47    else{ //fils 1
48        struct sigaction new;
49        new.sa_handler=handler_fils1;
50        sigaction(SIGUSR1,&new,NULL);
51        sleep(3);
52        pause();
53        exit(0);
54    }
55    return 0;
56}

```

Listing 9: Programme principal

## Introduction

- Les documents de cours sont autorisés
- Les réponses doivent être **claires et concises** : cela sera pris en compte lors de l'évaluation des réponses
- La durée de l'examen est 90 minutes

**Q1)** Donner cinq cas dans lesquels une interruption peut se produire ? **(1,5pts)**

- Interruption horloge
- Fin E/S
- Débordement,
- Division par zéro
- Fin de Quantum
- Etc.

**Q2)** Écrire un programme dans lequel un processus père crée 3 processus fils et attend leurs retours et les affiche à l'écran ? **(1,5pts)**

Il faut que les programmes créés soient des fils de même père. Il faut aussi que le père les attend et récupère leurs retours.

**Q3)** Donner la différence entre un appel système et une fonction utilisateur. Lequel entre eux s'exécute de façon plus rapide ? **(1pt)**

L'appel system fait partie des fonctions de la bibliothèque POSIX (c'est-à-dire des fonctions préfinies développées par les concepteurs du système d'exploitation), qui s'exécutent en mode système. Les fonctions utilisateurs sont des programmes développés par les utilisateurs du système d'exploitation.

**Q4)** Un processeur ne traite que les interruptions et il a une capacité de lire ou écrire un mot mémoire de 4 octets en  $10\text{ns}$ . Lors d'une interruption, il faut recopier 32 registres dans la pile du processus. Quel est le nombre maximal d'interruptions que ce processeur peut traiter en  $2\text{ms}$  ? **(2 pt)**

Le temps d'une interruption est  $2 \times 32 \times 10\text{ns} = 640\text{ ns}$ . Le nombre d'interruptions qu'il peut traiter en  $2\text{ms}$  est  $2000000/640 = 3125$

**Q5)** Donnez deux raisons pour lesquelles un processus en cours d'exécution peut être interrompu **(1pt)**.

- Déroulement
- Fin de quantum

**Q6)** Un disque comporte 100 pistes numérotées de 0 à 99. La dernière requête traitée concernait la piste 28 et une requête pour la piste 30 est en cours. La liste des nouvelles requêtes dans l'ordre d'arrivée est la suivante : 60,10,80,20,70,35,0,90

- a) Calculer le déplacement total de la tête pour les algorithmes suivants en illustrant votre réponse par un diagramme des déplacements effectués : **(2 pt)**  
 - FIFO, PCTR, SCAN, LOOK
- b) Pourquoi est-il dangereux d'autoriser le traitement immédiat de nouvelles requêtes ? **(1pt)**

Cela crée le problème de famine

- c) Citez, parmi ces algorithmes, un algorithme qui souffre de cette insuffisance et expliquer pourquoi **(1 pt)**

- d) Le traitement immédiat de nouvelles requêtes risque d'entraîner une famine. PCTR est un algorithme qui risque d'entraîner une famine.

**Pour les questions suivantes choisir la bonne réponse**

**Q7)** Le tube ne permet pas de faire communiquer des processus de machines différentes. (1pt)

- A) Vrai  
B) Faux

**Q8)** Lorsqu'un processus bloque un signal, le signal est délivré et le processus le gère en le jetant (1pt)

- A) Vrai  
B) Faux

**Q9)** Voici le code (une partie) d'un programme qui permet d'initialiser un fichier via un segment de mémoire partagé :

```
1. fd= open(argv[1], O_RDWR);  
2. stat (argv[1], &etat_fichier );  
3. long taille_fichier = etat_fichier.st_size ;  
4. projection = ( char * ) mmap(NULL, taille_fichier, PROT_READ |  
    PROT_WRITE,MAP_SHARED, fd 0);  
5. memset(projection, 'A'+1, taille_fichier);  
6. close( fichier );  
7. munmap((void *) projection, taille_fichier )
```

Si on suppose que la taille du fichier est de 10 octets, alors son contenu après l'exécution de ce programme sera : (1.5 pt)

- A) BBBBBBBBBB  
B) BBBB  
C) AAAAAAAA  
D) AAAA

**Q10)** Dans le système UNIX, les véritables appels système sont effectués à partir d'un programme utilisateur, d'une commande Shell, d'une procédure de la bibliothèque standard. Elles sont exécutées en : (1pt)

- A) Mode superviseur  
B) Aucune de ces réponses  
C) Mode utilisateur

**Q11)** Un lot est composé de 50 travaux, que pour simplifier, on suppose tous constitués de 3 phases indépendantes :

- lecture des cartes (20 secondes)
- calcul (15 secondes)
- impression des résultats (5 secondes).

Le temps mis pour passer d'un travail à un autre est négligeable.

Calculer le taux d'utilisation de l'unité centrale pour le calcul dans le cas où l'unité centrale gère les périphériques d'entrée-sortie (2 pt)

- A) 37,5 %  
B) Aucune de ces réponses  
C) 7,5%  
D) 6,6%

**Q12)** Soit le programme suivant :

```
main(int argc, char **argv)
{int child = fork();
int c = 5;
if(child == 0)
{c += 5;}
else
{child = fork();
c += 10;
if(child)c += 5;}}
```

Combien de copies différentes de la variable c existe-t-il ? **(1pt)**

- A) 1
- B) 2
- C) 3
- D) 4
- E) aucune de ces réponses

**Q13)** On considère le morceau de programme suivant :

```
int main(void) {
int n = 0;
while(n < 3) {
int pid = fork();
if(pid == -1) exit(EXIT_FAILURE);
if(pid > 0) {
if(pid % 5 == 0) n++;
else
if(waitpid(pid, NULL, 0) == -1) exit(EXIT_FAILURE);
}
if(pid == 0) {
if(getpid() % 5 == 0) sleep(10);
exit(EXIT_SUCCESS);
}
}
for(int i = 0; i < 3; i++)
if(waitpid(-1, NULL, 0) == -1) exit(EXIT_FAILURE);
exit(EXIT_SUCCESS);
}
```

Dans le programme précédent, le processus principal crée : **(1pt)**

- A) exactement 3 fils
- B) des fils jusqu'à en avoir 3 dont le PID est un multiple de 5
- C) des fils jusqu'à en avoir 5 dont le PID est un multiple de 3

**Q14)** Pour deux processus accédant à une variable partagée, l'algorithme de Peterson garantit : **(1pt)**

- A) exclusion mutuelle
- B) progrès
- C) attente limitée
- D) aucune de ces réponses

**Q15)** Un processus Zombie est un processus : **(1pt)**

- A) qui a terminé son exécution et qui attend la prise en compte de cette fin par son père

- B) qui a perdu son père
- C) qui a terminé son exécution en erreur
- D) aucune de ces réponses

## Réponses aux questions

### Partie I

- 1) Les problèmes de gestion des entrées/sorties, gestion de la mémoire, et gestion des processus, etc. se poseraient.
- 2)
  - a) A l'état bloqué, on ne sait pas si la cause du blocage a été entièrement traitée par le SE. Si entièrement traitée, on ne sait pas si le processeur est libre et le cas échéant, si le processus a la priorité d'exécution.
  - b) Le CO indique l'adresse de la prochaine instruction à exécuter. Le PSW indique l'état ou une image du processeur.
- 3) Un système monoprogrammation sera généralement plus facile à implémenter qu'un système multiprogrammation
- 4) Interruption : évènement extérieur provenant généralement d'une E/S (interruption matérielle). Déroutement : évènement interne lié à une exception provoquée par le programme en cours d'exécution. Appel au superviseur : évènement interne à un processus demandant au SE la réalisation d'une tâche en mode superviseur.

Note : Tous les trois entraînent des commutations de contexte.

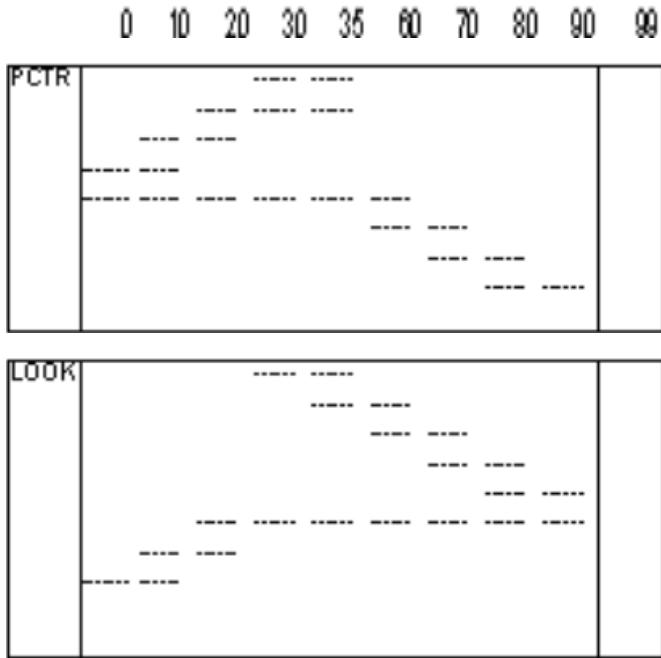
- 5)
  - a) Le DMA est en concurrence avec le processeur sur le bus de données.
  - b) Le processeur via le pilote du périphérique initialise le DMA qui lance l'E/S. Le DMA signale la fin d'entrée/sortie par le déclenchement d'une interruption que le processeur traite en exécutant une routine d'interruption.
- 6) Cela facilite la gestion modulaire des périphériques par le SE. Au niveau physique le contrôleur de périphérique assure l'interopérabilité matérielle. Le niveau logique est la couche de programmation utilisant des primitives de la couche physique.
- 7) Capacité maximale théorique = nombre de faces\*nombre de pistes par faces\*nombre de secteur par piste\*taille secteur.

Taille maximale d'un fichier =  $(10 + 256 + 256*256)*4 \text{ Ko}$

8) Interruptions, DMA, etc.

9)

- a) PCTR : 130
- b) LOOK : 150



10)

- a) faux (c'est possible mais cela peut être complexe)
- b) faux (ca dépend du contexte d'utilisation)
- c) faux (ca dépend du masque d'interruption et la priorité)
- d) faux (taille maximale fichier =  $10 + 2^8 + 2^{16} + 2^{24}$  Ko >  $2^4 * 2^{20}$  Ko)

## Partie II

1)

- (i) désactiver toutes les interruptions : mode noyau, bien que ce soit un moyen basique de faire de la synchronisation, si un utilisateur le fait mal, alors le système d'exploitation cessera de fonctionner et n'aura aucun moyen de sortir.
  - (ii) lire l'horloge du jour : mode utilisateur, il s'agit d'une opération en lecture seule, et un utilisateur ne peut pas faire de mal. Cependant, certains systèmes d'exploitation le conservent dans le noyau
  - (iii) régler l'horloge du jour : mode noyau, si l'utilisateur a changé le temps, les autres utilisateurs seraient affectés négativement
  - (iv) tuer un processus : mode noyau, les utilisateurs ne peuvent pas tuer les processus arbitrairement. Ils peuvent tuer leurs propres processus, mais le noyau doit vérifier que le paramètre appartient effectivement à l'utilisateur appelant.
- 2) Un signal est généré par le processeur ou un processus. Il est envoyé à un processus. Une interruption est générée par une dispositif d'E/S. Elle est traitée par le processeur.
- 3) Si un processus attend une opération qui se produira bientôt, cela ne vaut pas la peine de faire le changement de contexte impliqué dans le blocage d'une interruption. Généralement, lorsque le temps d'attente est inférieur au temps système pour le changement de contexte, il est préférable d'interroger.

- 4) La séquence de Fibonacci est la série de nombres 0, 1, 1, 2, 3, 5, 8, .... Formellement, elle peut être exprimée comme suit:

$$\begin{aligned}fib(0) &= 0 \\fib(1) &= 1 \\fib(n) &= fib(n-1) + fib(n-2), n>1\end{aligned}$$

a)

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[])
{
    int k, pid,i=0,j=0;
    int sum=1;
    int num=atoi(argv[1]);
    if (argc == 0)
    {
        printf ("Entrer un nombre à la séquence de fibonacci ");
    }

    pid = fork();
    if (pid == 0)
    {
        for(k=1;k<num;k++)
        {
            i = j;
            j= sum;
            sum = i + j;
            printf("Fibonacci (%d) = %d\n",k,sum);
        }
    }
    else
    {
        wait();
        printf ("Le parent a attendu son fils \n");
    }
    return 0;
}
```

- b) Le fils passe à l'état Zombie  
c) En utilisant la communication interprocessus : pipe ou segment de mémoire partagé ou file de message ou socket.

5)

- a) 256 couleurs ==> 8 bits par point ==> 1 octet par point

- b) Les processus doivent partager leurs PID. *struct shared\_data* permet de les partager.  
 Contre-exemple : Si on utilise des variables globales, il se peut que ces variables ne soient pas initialisées avant un fork, donc le premier processus créé ne saura pas le PID de chacun des deux autres processus créés après sa création.
- c) (4) shmid=shmget(IPC\_PRIVATE,sizeof(struct shared\_data),IPC\_CREAT|0666);  
 (5) shared\_d=(struct shared\_data\*)shmat(shmid,0,0);
- d) (6) sigact\_lecteur.sa\_handler=handler\_lecteur;  
 sigaction(SIGUSR1,&sigact\_lecteur,NULL);
- (7) sigact\_rotation.sa\_handler=handler\_rotation;  
 sigaction(SIGUSR1, &sigact\_rotation, NULL);
- (8) sigact\_redacteur.sa\_handler=handler\_redacteur;  
 sigaction(SIGUSR1, &sigact\_redacteur, NULL);

e)

```

void handler_lecteur(int x){
    int i,j;
    image=mmap(0,size*size,PROT_READ|PROT_WRITE,MAP_SHARED,shared_d->fd,0);
    for(i=0;i<size;i++){
        for(j=0;j<size;j++)
            printf("%c",*(image+i*size+j));
        printf("\n");
    }
    kill(shared_d->p_rotation, SIGUSR1);
}

void handler_rotation(int x){
    struct iovec vec[size];
    int i,j;
    close(shared_d->piped[0]);
    image=mmap(0,size*size,PROT_READ|PROT_WRITE,MAP_SHARED,shared_d->fd,0);
    for(j=0;j<size;j++){
        for(i=0;i<size;i++){
            vec[i].iov_base=image+i*size+j;
            vec[i].iov_len=1;
        }
        writev(shared_d->piped[1],vec,size);
    }
    close(shared_d->piped[1]);
    kill(shared_d->p_redacteur,SIGUSR1);
}

void handler_redacteur(int x){
    struct iovec vec[size];
    int i;
    close(shared_d->piped[1]);
}

```

```
image=mmap(0,size*size,PROT_READ|PROT_WRITE,MAP_SHARED,shared_d->fd,0);

    for(i=0;i<size;i++){
        vec[i].iov_base=image+i*size;
        vec[i].iov_len=size;
    }
    readv(shared_d->piped[0],vec,size);
    close(shared_d->piped[0]);
}
```

Nom : .....

Prénom : .....

## Partie II

### Exercice 2 (5 pts)

La suite de Syracuse est définie par :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

La conjecture de Syracuse affirme que, quel que soit le terme initial  $u_0$  (non nul) de la suite, celle-ci finit par valoir 1 (puis boucler sur 4, 2, 1).

Etant donnée une valeur initiale  $u_0$  de la suite, on appelle *temps de vol*, le plus petit indice  $k$  pour lequel  $u_k = 1$  pour la première fois, et *altitude maximale* la valeur maximale prise par la suite durant ce temps de vol.

Par exemple, pour  $u_0 = 14$ , on construit la suite des nombres :

14, 7, 22, 11, 34, 17, **52**, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, **1**, 4, 2, 1, ...

Le temps de vol dans cet exemple est de 17 et l'altitude maximale est de 52.

1. Ecrire un programme qui exécute une boucle infinie dans laquelle on fixe le terme initial  $u_0$  de manière aléatoire, puis on calcule les termes de la suite de Syracuse jusqu'à ce qu'elle vaille 1. Le programme boucle indéfiniment sans rien afficher.
2. En utilisant l'interface POSIX de gestion des signaux :
  - Faire en sorte qu'à la réception du signal SIGTSTP (Ctrl-Z), le programme affiche :
    - la valeur initiale courante,
    - la valeur et l'indice du dernier terme calculé.
  - Faire en sorte qu'à la réception du signal SIGINT (Ctrl-C), le programme affiche :
    - la plus grande valeur initiale testée ;
    - le plus grand temps de vol atteint et la valeur initiale pour laquelle ce dernier est atteint ;
    - la plus grande altitude maximale atteinte et la valeur initiale pour laquelle cette dernière est atteinte.

Si le programme reçoit dans les trois secondes un autre signal SIGINT et si le signal SIGTSTP a été reçu au moins 10 fois, alors le programme doit se terminer.

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

/*****************/
/** Variables globales ***/ (0.25 pt)
/*****************/

unsigned long int u0, uk, k;
unsigned long int tvol_max=0, u0_tvol_max;
unsigned long int alt_max=0, u0_alt_max;
struct sigaction act_sigint, act_sigstop, act_alarm;
unsigned int nb_sigstop = 0;
int in_sigint = 0;
```

Nom : .....

Prénom : .....

```
*****  
*** Prototypes des fonctions *** (0.25 pt)  
*****  
void handler_sigstop(int signum);  
void handler_sigin(int signum);  
void handler_alarm(int signum);  
  
*****  
*** Programme principal *** (1.5 pts)  
*****  
main(){  
    act_sigstop.sa_handler = handler_sigstop;  
    sigaction(SIGTSTP, &act_sigstop, NULL);  
    act_sigin.sa_handler = handler_sigin;  
    sigaction(SIGINT, &act_sigin, NULL);  
    act_alarm.sa_handler = handler_alarm;  
    sigaction(SIGALRM, &act_alarm, NULL);  
  
    while(1){  
        u0 = random()%2000+1;  
        uk = u0; k = 0;  
        while(uk != 1){  
            if(uk > alt_max){  
                alt_max = uk;  
                u0_alt_max = u0;  
            }  
            if(uk%2) uk = 3*uk+1;  
            else uk = uk/2;  
            k++;  
        }  
        if(k > tvol_max){  
            tvol_max = k;  
            u0_tvol_max = u0;  
        }  
    }  
}  
  
*****  
*** Définition des fonctions *** (3 pts)  
*****  
void handler_sigstop(int signum){  
    printf("\n\n u(0) = %d \t u(%d) = %d\n", u0,k,uk);  
    nb_sigstop++;  
}  
void handler_sigin(int signum){  
    if(!in_sigin){  
        printf("\n\nDernier u(0) = %d \n", u0);  
        printf("Temps de vol max = %d \t u(0) = %d\n", tvol_max, u0_tvol_max);  
        printf("Altitude max = %d \t u(0) = %d\n", alt_max, u0_alt_max);  
        in_sigin = 1;  
        alarm(2);  
    }  
    else if(nb_sigstop >= 5) exit(0);  
}  
void handler_alarm(int signum){  
    in_sigin = 0;  
}
```

## Réponses aux questions

### Partie I

- 1) Les problèmes de gestion des entrées/sorties, gestion de la mémoire, et gestion des processus, etc. se poseraient.
- 2)
  - a) A l'état bloqué, on ne sait pas si la cause du blocage a été entièrement traitée par le SE. Si entièrement traitée, on ne sait pas si le processeur est libre et le cas échéant, si le processus a la priorité d'exécution.
  - b) Le CO indique l'adresse de la prochaine instruction à exécuter. Le PSW indique l'état ou une image du processeur.
- 3) Un système monoprogrammation sera généralement plus facile à implémenter qu'un système multiprogrammation
- 4) Interruption : évènement extérieur provenant généralement d'une E/S (interruption matérielle). Déroutement : évènement interne lié à une exception provoquée par le programme en cours d'exécution. Appel au superviseur : évènement interne à un processus demandant au SE la réalisation d'une tâche en mode superviseur.

Note : Tous les trois entraînent des commutations de contexte.

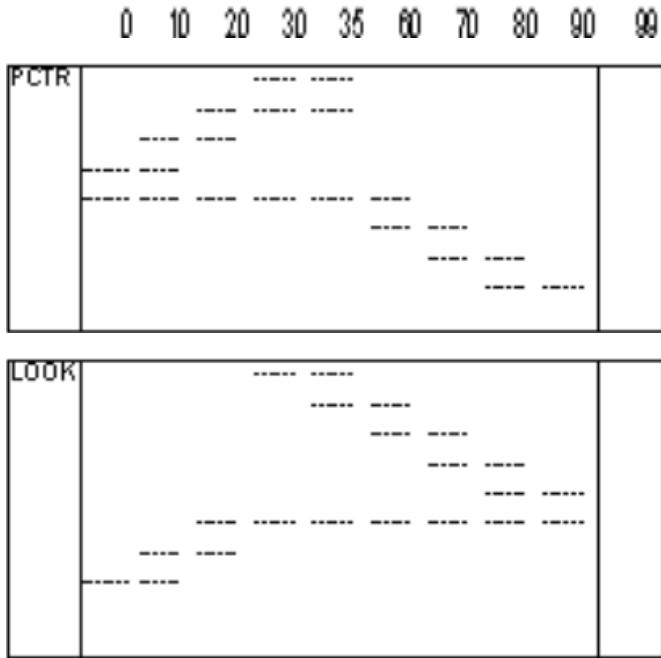
- 5)
  - a) Le DMA est en concurrence avec le processeur sur le bus de données.
  - b) Le processeur via le pilote du périphérique initialise le DMA qui lance l'E/S. Le DMA signale la fin d'entrée/sortie par le déclenchement d'une interruption que le processeur traite en exécutant une routine d'interruption.
- 6) Cela facilite la gestion modulaire des périphériques par le SE. Au niveau physique le contrôleur de périphérique assure l'interopérabilité matérielle. Le niveau logique est la couche de programmation utilisant des primitives de la couche physique.
- 7) Capacité maximale théorique = nombre de faces\*nombre de pistes par faces\*nombre de secteur par piste\*taille secteur.

Taille maximale d'un fichier =  $(10 + 256 + 256*256)*4 \text{ Ko}$

8) Interruptions, DMA, etc.

9)

- a) PCTR : 130
- b) LOOK : 150



10)

- a) faux (c'est possible mais cela peut être complexe)
- b) faux (ca dépend du contexte d'utilisation)
- c) faux (ca dépend du masque d'interruption et la priorité)
- d) faux (taille maximale fichier =  $10 + 2^8 + 2^{16} + 2^{24}$  Ko >  $2^4 * 2^{20}$  Ko)

## Partie II

1)

- (i) désactiver toutes les interruptions : mode noyau, bien que ce soit un moyen basique de faire de la synchronisation, si un utilisateur le fait mal, alors le système d'exploitation cessera de fonctionner et n'aura aucun moyen de sortir.
  - (ii) lire l'horloge du jour : mode utilisateur, il s'agit d'une opération en lecture seule, et un utilisateur ne peut pas faire de mal. Cependant, certains systèmes d'exploitation le conservent dans le noyau
  - (iii) régler l'horloge du jour : mode noyau, si l'utilisateur a changé le temps, les autres utilisateurs seraient affectés négativement
  - (iv) tuer un processus : mode noyau, les utilisateurs ne peuvent pas tuer les processus arbitrairement. Ils peuvent tuer leurs propres processus, mais le noyau doit vérifier que le paramètre appartient effectivement à l'utilisateur appelant.
- 2) Un signal est généré par le processeur ou un processus. Il est envoyé à un processus. Une interruption est générée par une dispositif d'E/S. Elle est traitée par le processeur.
- 3) Si un processus attend une opération qui se produira bientôt, cela ne vaut pas la peine de faire le changement de contexte impliqué dans le blocage d'une interruption. Généralement, lorsque le temps d'attente est inférieur au temps système pour le changement de contexte, il est préférable d'interroger.

- 4) La séquence de Fibonacci est la série de nombres 0, 1, 1, 2, 3, 5, 8, .... Formellement, elle peut être exprimée comme suit:

$$\begin{aligned}fib(0) &= 0 \\fib(1) &= 1 \\fib(n) &= fib(n-1) + fib(n-2), n>1\end{aligned}$$

a)

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[])
{
    int k, pid,i=0,j=0;
    int sum=1;
    int num=atoi(argv[1]);
    if (argc == 0)
    {
        printf ("Entrer un nombre à la séquence de fibonacci ");
    }

    pid = fork();
    if (pid == 0)
    {
        for(k=1;k<num;k++)
        {
            i = j;
            j= sum;
            sum = i + j;
            printf("Fibonacci (%d) = %d\n",k,sum);
        }
    }
    else
    {
        wait();
        printf ("Le parent a attendu son fils \n");
    }
    return 0;
}
```

- b) Le fils passe à l'état Zombie  
c) En utilisant la communication interprocessus : pipe ou segment de mémoire partagé ou file de message ou socket.

5)

- a) 256 couleurs ==> 8 bits par point ==> 1 octet par point

- b) Les processus doivent partager leurs PID. *struct shared\_data* permet de les partager.  
 Contre-exemple : Si on utilise des variables globales, il se peut que ces variables ne soient pas initialisées avant un fork, donc le premier processus créé ne saura pas le PID de chacun des deux autres processus créés après sa création.
- c) (4) shmid=shmget(IPC\_PRIVATE,sizeof(struct shared\_data),IPC\_CREAT|0666);  
 (5) shared\_d=(struct shared\_data\*)shmat(shmid,0,0);
- d) (6) sigact\_lecteur.sa\_handler=handler\_lecteur;  
 sigaction(SIGUSR1,&sigact\_lecteur,NULL);
- (7) sigact\_rotation.sa\_handler=handler\_rotation;  
 sigaction(SIGUSR1, &sigact\_rotation, NULL);
- (8) sigact\_redacteur.sa\_handler=handler\_redacteur;  
 sigaction(SIGUSR1, &sigact\_redacteur, NULL);

e)

```

void handler_lecteur(int x){
    int i,j;
    image=mmap(0,size*size,PROT_READ|PROT_WRITE,MAP_SHARED,shared_d->fd,0);
    for(i=0;i<size;i++){
        for(j=0;j<size;j++)
            printf("%c",*(image+i*size+j));
        printf("\n");
    }
    kill(shared_d->p_rotation, SIGUSR1);
}

void handler_rotation(int x){
    struct iovec vec[size];
    int i,j;
    close(shared_d->piped[0]);
    image=mmap(0,size*size,PROT_READ|PROT_WRITE,MAP_SHARED,shared_d->fd,0);
    for(j=0;j<size;j++){
        for(i=0;i<size;i++){
            vec[i].iov_base=image+i*size+j;
            vec[i].iov_len=1;
        }
        writev(shared_d->piped[1],vec,size);
    }
    close(shared_d->piped[1]);
    kill(shared_d->p_redacteur,SIGUSR1);
}

void handler_redacteur(int x){
    struct iovec vec[size];
    int i;
    close(shared_d->piped[1]);
}

```

```
image=mmap(0,size*size,PROT_READ|PROT_WRITE,MAP_SHARED,shared_d->fd,0);

    for(i=0;i<size;i++){
        vec[i].iov_base=image+i*size;
        vec[i].iov_len=size;
    }
    readv(shared_d->piped[0],vec,size);
    close(shared_d->piped[0]);
}
```

## Introduction

- Les documents de cours sont autorisés
- Les réponses doivent être **claires et concises** : cela sera pris en compte lors de l'évaluation des réponses
- La durée de l'examen est 90 minutes

**Q1)** Donner cinq cas dans lesquels une interruption peut se produire ? **(1,5pts)**

- Interruption horloge
- Fin E/S
- Débordement,
- Division par zéro
- Fin de Quantum
- Etc.

**Q2)** Écrire un programme dans lequel un processus père crée 3 processus fils et attend leurs retours et les affiche à l'écran ? **(1,5pts)**

Il faut que les programmes créés soient des fils de même père. Il faut aussi que le père les attend et récupère leurs retours.

**Q3)** Donner la différence entre un appel système et une fonction utilisateur. Lequel entre eux s'exécute de façon plus rapide ? **(1pt)**

L'appel system fait partie des fonctions de la bibliothèque POSIX (c'est-à-dire des fonctions préfinies développées par les concepteurs du système d'exploitation), qui s'exécutent en mode système. Les fonctions utilisateurs sont des programmes développés par les utilisateurs du système d'exploitation.

**Q4)** Un processeur ne traite que les interruptions et il a une capacité de lire ou écrire un mot mémoire de 4 octets en  $10\text{ns}$ . Lors d'une interruption, il faut recopier 32 registres dans la pile du processus. Quel est le nombre maximal d'interruptions que ce processeur peut traiter en  $2\text{ms}$  ? **(2 pt)**

Le temps d'une interruption est  $2 \times 32 \times 10\text{ns} = 640\text{ ns}$ . Le nombre d'interruptions qu'il peut traiter en  $2\text{ms}$  est  $2000000/640 = 3125$

**Q5)** Donnez deux raisons pour lesquelles un processus en cours d'exécution peut être interrompu **(1pt)**.

- Déroulement
- Fin de quantum

**Q6)** Un disque comporte 100 pistes numérotées de 0 à 99. La dernière requête traitée concernait la piste 28 et une requête pour la piste 30 est en cours. La liste des nouvelles requêtes dans l'ordre d'arrivée est la suivante : 60,10,80,20,70,35,0,90

- a) Calculer le déplacement total de la tête pour les algorithmes suivants en illustrant votre réponse par un diagramme des déplacements effectués : **(2 pt)**  
 - FIFO, PCTR, SCAN, LOOK
- b) Pourquoi est-il dangereux d'autoriser le traitement immédiat de nouvelles requêtes ? **(1pt)**

Cela crée le problème de famine

- c) Citez, parmi ces algorithmes, un algorithme qui souffre de cette insuffisance et expliquer pourquoi **(1 pt)**

- d) Le traitement immédiat de nouvelles requêtes risque d'entraîner une famine. PCTR est un algorithme qui risque d'entraîner une famine.

**Pour les questions suivantes choisir la bonne réponse**

**Q7)** Le tube ne permet pas de faire communiquer des processus de machines différentes. (1pt)

- A) Vrai  
B) Faux

**Q8)** Lorsqu'un processus bloque un signal, le signal est délivré et le processus le gère en le jetant (1pt)

- A) Vrai  
B) Faux

**Q9)** Voici le code (une partie) d'un programme qui permet d'initialiser un fichier via un segment de mémoire partagé :

```
1. fd= open(argv[1], O_RDWR);  
2. stat (argv[1], &etat_fichier );  
3. long taille_fichier = etat_fichier.st_size ;  
4. projection = ( char * ) mmap(NULL, taille_fichier, PROT_READ |  
    PROT_WRITE,MAP_SHARED, fd 0);  
5. memset(projection, 'A'+1, taille_fichier);  
6. close( fichier );  
7. munmap((void *) projection, taille_fichier )
```

Si on suppose que la taille du fichier est de 10 octets, alors son contenu après l'exécution de ce programme sera : (1.5 pt)

- A) BBBBBBBBBB  
B) BBBB  
C) AAAAAAAA  
D) AAAA

**Q10)** Dans le système UNIX, les véritables appels système sont effectués à partir d'un programme utilisateur, d'une commande Shell, d'une procédure de la bibliothèque standard. Elles sont exécutées en : (1pt)

- A) Mode superviseur  
B) Aucune de ces réponses  
C) Mode utilisateur

**Q11)** Un lot est composé de 50 travaux, que pour simplifier, on suppose tous constitués de 3 phases indépendantes :

- lecture des cartes (20 secondes)
- calcul (15 secondes)
- impression des résultats (5 secondes).

Le temps mis pour passer d'un travail à un autre est négligeable.

Calculer le taux d'utilisation de l'unité centrale pour le calcul dans le cas où l'unité centrale gère les périphériques d'entrée-sortie (2 pt)

- A) 37,5 %  
B) Aucune de ces réponses  
C) 7,5%  
D) 6,6%

**Q12)** Soit le programme suivant :

```
main(int argc, char **argv)
{int child = fork();
int c = 5;
if(child == 0)
{c += 5;}
else
{child = fork();
c += 10;
if(child)c += 5;}}
```

Combien de copies différentes de la variable c existe-t-il ? **(1pt)**

- A) 1
- B) 2
- C) 3
- D) 4
- E) aucune de ces réponses

**Q13)** On considère le morceau de programme suivant :

```
int main(void) {
int n = 0;
while(n < 3) {
int pid = fork();
if(pid == -1) exit(EXIT_FAILURE);
if(pid > 0) {
if(pid % 5 == 0) n++;
else
if(waitpid(pid, NULL, 0) == -1) exit(EXIT_FAILURE);
}
if(pid == 0) {
if(getpid() % 5 == 0) sleep(10);
exit(EXIT_SUCCESS);
}
}
for(int i = 0; i < 3; i++)
if(waitpid(-1, NULL, 0) == -1) exit(EXIT_FAILURE);
exit(EXIT_SUCCESS);
}
```

Dans le programme précédent, le processus principal crée : **(1pt)**

- A) exactement 3 fils
- B) des fils jusqu'à en avoir 3 dont le PID est un multiple de 5
- C) des fils jusqu'à en avoir 5 dont le PID est un multiple de 3

**Q14)** Pour deux processus accédant à une variable partagée, l'algorithme de Peterson garantit : **(1pt)**

- A) exclusion mutuelle
- B) progrès
- C) attente limitée
- D) aucune de ces réponses

**Q15)** Un processus Zombie est un processus : **(1pt)**

- A) qui a terminé son exécution et qui attend la prise en compte de cette fin par son père

- B) qui a perdu son père
- C) qui a terminé son exécution en erreur
- D) aucune de ces réponses

## Introduction

- Seuls les documents de cours sous format papier sont autorisés
- Répondez sur le sujet de l'examen
- Les réponses doivent être **claires et concises** : cela sera pris en compte lors de l'évaluation des réponses
- La durée de l'examen est 120 minutes.

## Partie 1 (10 points) --- Copie 1

- 1) Citer deux inconvénients de la multiprogrammation et deux avantages des systèmes à temps partagé. **(1 point)**

- Inconvénients de la multiprogrammation : Gestion de la mémoire, le prob de sécurité des processus, il faut un ordonnanceur ... ect.
- Avantages des systèmes à temps partagé : Utilisation efficace du CPU, Temps de traitement petit pour les programmes courts.

- 2) L'accès direct à la mémoire est utilisé pour les périphériques d'E/S à grande vitesse afin d'éviter la surcharge du processeur.

- a) Comment le processeur communique-t-il avec le périphérique pour coordonner le transfert ? **(0,75 point)**

La CPU peut lancer une opération DMA en écrivant des valeurs dans des registres spéciaux auxquels le périphérique peut accéder indépendamment.

- b) Comment le processeur sait-il que les opérations d'E/S sont terminées? **(0,75 point)**

Lorsque le périphérique a terminé son opération, il interrompt la CPU pour indiquer la fin de l'opération.

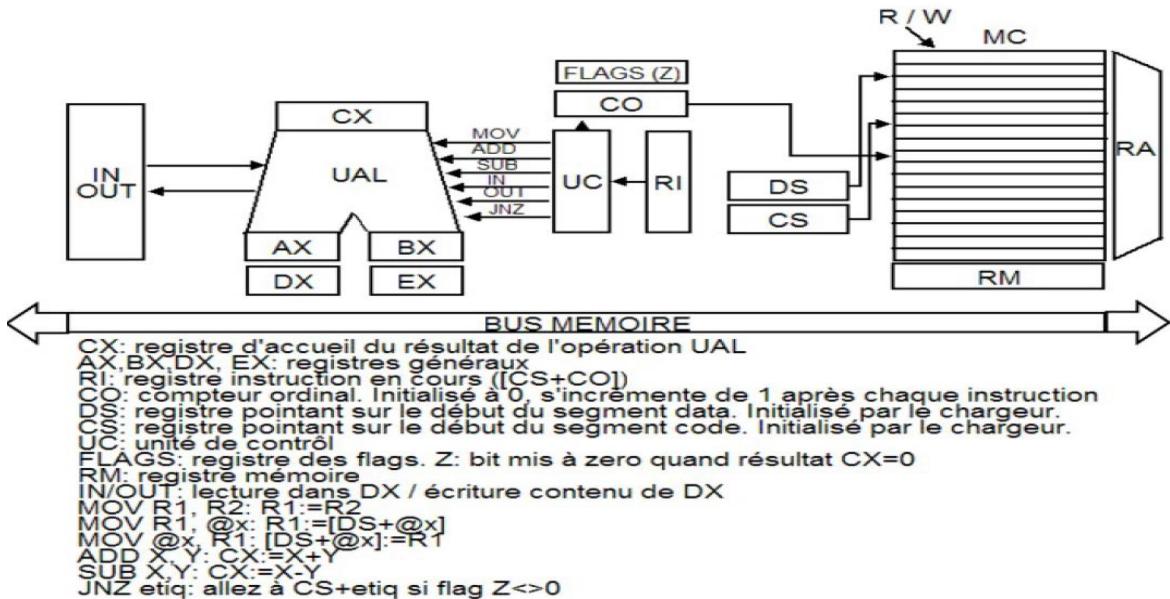
- 3) Un processeur ne traite que les interruptions et il a une capacité de lire ou écrire un mot mémoire de 4 octets en  $10\text{ns}$ . Lors d'une interruption il faut recopier 32 registres dans la pile du processus. Quel est le nombre maximal d'interruptions que ce processeur peut traiter en  $2\text{ms}$  ? **(1,5 points)**

*Le temps d'une interruption est  $2 \times 32 \times 10\text{ns} = 640\text{ ns}$ . Le nombre d'interruptions qu'il peut traiter en  $2\text{ms}$  est  $2000000 / 640 = 31225$ .*

- 4) Que pouvez-vous dire d'un système dans lequel lorsqu'un processus  $P$  crée un processus fils, l'exécution du processus  $P$  est suspendu jusqu'à la terminaison du processus fils ? (1 point)

*Système mono-programmé*

- 5) Soit l'architecture suivante :



- a) Dans cette architecture, peut-on exécuter un programme sans stocker ses données en mémoire centrale (MC) ? (0,5 point)

*Oui : on peut utiliser les registres si le nombre de données est petit.*

- b) Dans cette architecture, peut-on exécuter un programme sans stocker ses instructions en mémoire centrale (MC) ? (0,5 point)

*Non. Le processus est installé dans la MC.*

- c) Citer deux types d'entité de cette architecture qui peuvent envoyer un signal (0,5 point)

*Processus, processeur.*

- d) Citer un exemple d'entité de cette architecture qui peut générer une interruption (0,5 point)

*E/S*

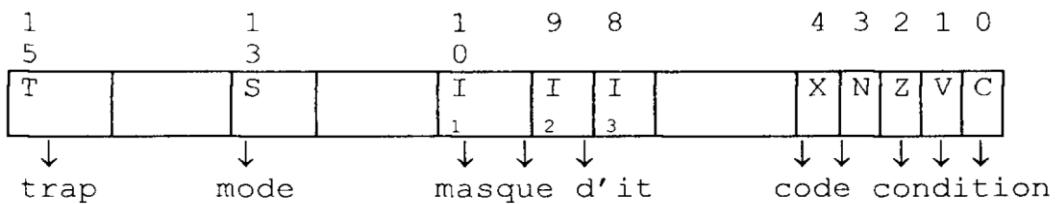
- a) Citer les avantages et les inconvénients de cette architecture (**0,5 point**)

*Il s'agit d'un type de l'architecture de Von Neumann. L'inconvénient est les goulots d'étranglements.*

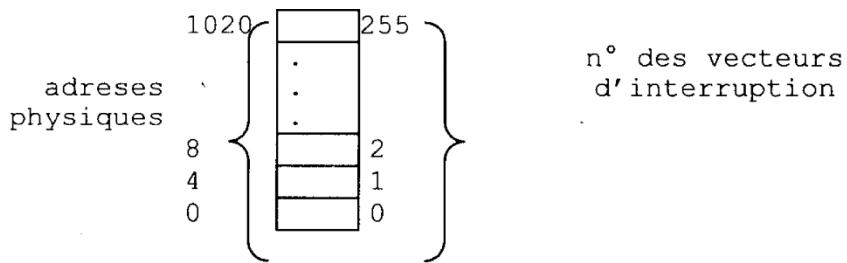
- b) Proposer une autre architecture qui répond aux inconvénients de cette architecture et discuter les caractéristiques de cette nouvelle architecture ? (**1 point**)

*Architecture de Harvard.*

- 6) Le contexte d'un programme s'exécutant sur une machine X est constitué de 16 registres généraux (R0 à R15), du compteur ordinal (CO) et du PSW dont la structure est la suivante :



Cette machine reconnaît 256 causes d'interruption (it) dont les vecteurs sont stockés dans la table suivante:



Le vecteur d'interruption contient le compteur ordinal et la routine d'interruption correspondante. Parmi ces 256 causes d'interruption huit causes sont dues à des organes externes. Chacune de ces huit causes possède un numéro d'interruption qui correspond à sa priorité (de 0 à 7). Une interruption de ce type arrive de la manière suivante :

- i. L'unité centrale ne prend en considération un signal d'interruption que si le numéro de celle-ci est supérieur à la valeur dumasque d'it qui se trouve dans le PSW
- ii. L'unité centrale envoie un signal d'acquittement au composant qui a déclenché l'interruption
- iii. Le composant envoie alors le numéro de son vecteur d'it
- iv. L'unité centrale récupère ce numéro et empile dans la pile système le CO et le PSW du programme courant
- v. L'unité centrale positionne le bit S du PSW à 1 et le masque d'it à la valeur correspondante au numéro d'it récupéré en I
- vi. L'unité centrale se branche sur la routine d'it correspondante

La machine dispose de l'instruction RTI qui charge le PSW et le CO à partir des deux mots de tête de la pile système.

- a) Comment le système se branche vers la routine d'it  $i$ ,  $0 \leq i \leq 255$  (**0,5 point**)

*Co = i x 4;*

- b) Donner le corps d'une routine d'it quelconque (**0,5 point**)

*SR0=R0, ..... SR15=R15 ;  
Traitement ;  
R0=SR0, ..... R15=SR15;  
RTI;*

- c) Donner les valeurs du masque d'it du PSW pour les 8 causes d'it (**0,5 point**)

*000  
001  
010  
011  
101  
110  
100  
111*

## Correction de la partie 2 du median SR02-P2019

# 1 Solution Exercice 1 (3.5 points)

## 1.1 Question 1

Donner les bouts de code (3) et (4) nécessaires permettant d'installer des nouveaux handlers pour le traitement d'un signal SIGUSR1. **(1 point)**

## 1.2 Solution 1

```

1 struct sigaction old, new;
2 new.sa_handler=handler_pere;
3 sigaction(SIGUSR1,&new,&old);

```

Listing 1: installation du handler pour le père

```

1 struct sigaction new_fils;
2 new_fils.sa_handler = handler_fils;
3 sigaction(SIGUSR1,&new_fils,NULL);

```

Listing 2: installation du handler pour le fils

## 1.3 Question 2

Donner les bouts de code (5) et (6) nécessaires pour que les deux processus puissent synchroniser l'exécution tel qu'il est décrit dans l'énoncé. **(1 point)**

## 1.4 Solution 2

```

1 while(1){ // .... (5)
2     fibonacci(&n, &t1, &t2);
3     kill(getppid(), SIGUSR1); // envoi du signal SIGUSR1 au parent.
4     pause(); // se mettre en pause
5 }

```

Listing 3: Code du processus fils

```

1 while(1){ // .... (6)
2     pause();
3 }

```

Listing 4: Code du processus père

## 1.5 Question 3

Donner les bouts de codes (1) et (2) correspondants aux fonctions handler\_pere() et handler\_fils() qui serviront des handlers pour le traitement du signal SIGUSR1. (1.5 point)

## 1.6 Solution 3

```

1 void handler_fils(int sig){
2     nb_sig++;
3     if (nb_sig == MAX_SIGUSR1){
4         kill(getppid(), SIGINT);
5         exit(0);
6     }
7 }
8
9 void handler_pere(int sig){
10    sleep(5);
11    kill(pid_fils, SIGUSR1);
12 }
```

Listing 5: Exercice 3

## 2 Solution Exercice 2 (6.5 points)

### 2.1 Question 1

Expliquer pourquoi il faut partager le pid du processus p2 dans une mémoire partagée (0.5 points)

### 2.2 Solution 1

Le processus p1 a besoin du PID du processus p2 pour lui envoyer un signal. Le seul moyen pour que p1 connaisse le PID de p2 c'est d'utiliser une mémoire partagée pour stocker les PID de p2.

### 2.3 Question 2

Ecrire la fonction initab(int n, int fd) qui permet d'initialiser aléatoirement un tableau de n entiers et copier son contenu dans le fichier représenté par le descripteur fd. (0.5 points)

## 2.4 Solution 2

```

1 void initab(int n, int fd){
2     srand(time(NULL));
3     int i;
4     for(i=0; i<n; i++){
5         int a = rand()%30+10;
6         write(fd,&a,sizeof(a));
7     }
8 }
```

Listing 6: initab()

## 2.5 Question 3

Ecrire la fonction showtab(int n, int \* tab) qui permet d'afficher le contenu du tableau tab (projeté en mémoire par la primitive mmap). **(0.5 points)**

## 2.6 Solution 3

```

1 void showtab(int n, int *tab){
2     int i;
3     for(i=0; i<n; i++){
4         printf("\n\ttab[%d]=%d",i, tab[i]); fflush(stdout);
5     }
6 }
```

Listing 7: showtab()

## 2.7 Question 4

Ecrire les routines de traitement du signal SIGUSR1 handlers\_pere(), handler\_fils1(), et handlers\_fils2() liées aux processus parent, p1 et p2 respectivement. **(2.5 points)**

## 2.8 Solution 4

```

1 void handler_fils1(int sig){
2     showtab(n, tab);
3     int i;
4     for(i=0; i<n; i++){
5         if (!(tab[i]%2)) tab[i]+= 1;
6     }
7     kill(pids[1], SIGUSR1);
8 }
```

```

10 /* fonction utilisateur de comparaison fournie a qsort() */
11 static int compare (void const *a, void const *b)
12 {
13     int const *pa = a;
14     int const *pb = b;
15     return *pa - *pb;
16 }
17
18 void handler_fils2(int sig){
19     showtab(n, tab);
20     qsort(tab, n, sizeof(int), compare);
21     kill(getppid(), SIGUSR1);
22 }
23
24 void handler_pere(int sig){
25     showtab(n, tab);
26     shmdt(pids); // detacher le segment memoire du processus parent
27     shmctl(shmid, IPC_RMID, NULL); // supprimer le segment memoire
28     munmap((void *) tab, n*sizeof(int));
29 }
```

Listing 8: Les handlers

## 2.9 Question 5

Ecrire le programme principal permettant de créer les deux processus p1 et p2 et réaliser l'objectif décrit ci-dessus, en faisant appel aux différentes fonctions : initab, showtab, etc. (2.5 points)

## 2.10 Solution 5

```

1 // Les variables globales
2 int *tab, *pids, n, fd, shmid;
3
4 int main(int argc, char* argv[]){
5     if(argc < 3) {perror("\nUsage: <filepath> <n>"); exit(-1);}
6     struct sigaction new;
7     new.sa_handler=handler_pere;
8     sigaction(SIGUSR1,&new,NULL);
9     n = atoi(argv[2]);
10    fd = open(argv[1], O_RDWR|O_CREAT, 0666);
11    initab(n, fd);
12    tab = (int *)mmap(NULL, n*sizeof(int), PROT_READ|PROT_WRITE,
13                      MAP_SHARED, fd, 0);
14    if(tab == (int*)MAP_FAILED){
15        perror("Erreur mmap!!"); exit(1);
16    }
17    close(fd);
```

```

17    showtab(n,tab);
18    int p;
19    if ((shmid = shmget(IPC_PRIVATE, 2*sizeof(int), PERM)) == -1) {
20        perror("Echec de creation du segment");
21    return 1;
22    }
23    if ((pids = (int *)shmat(shmid, NULL, 0)) == (void *)-1) {
24        perror("Echec de l'attachement du segment");
25    }
26    if((p=fork())==-1){ perror("\n(parent) Echec de creation du
27        processus p1"); return 1;}
28    if(p > 0){ //parent
29        pids[0] = p; // sauvegarder le pid du fils 1 dans la memoire
30        partagee.
31        if((p=fork())==-1){ perror("\nEchec de creation du processus p2"
32        ); return 1;}
33        if(!p){ //fils 2
34            struct sigaction new;
35            new.sa_handler=handler_fils2;
36            sigaction(SIGUSR1,&new,NULL);
37            sleep(3);
38            pause();
39            exit(0);
40        }
41        pids[1]=p;
42        sleep(1);
43        kill(pids[0], SIGUSR1);
44        int i;
45        pause();
46    }
47    else{ //fils 1
48        struct sigaction new;
49        new.sa_handler=handler_fils1;
50        sigaction(SIGUSR1,&new,NULL);
51        sleep(3);
52        pause();
53        exit(0);
54    }
55    return 0;
56}

```

Listing 9: Programme principal

g

- Tout document autorisé
- Les réponses doivent être **claires et concises** : cela sera pris en compte lors de l'évaluation
- La durée de l'examen est 90 minutes

### Exercice 1 (6 pts)

Pour le programme ci-dessous, quelle sortie serait imprimée lors de son exécution ? Supposons que tous les appels de fonction, de bibliothèque et de système réussissent.

```

int x = 42;

main()
{
    int rc, status;
    rc = fork();
    if (rc == 0) {
        func1(); exit(1);
    }
    else {
        rc = waitpid(rc, &status, 0);
        printf("R: %d", WEXITSTATUS(status));
        printf("M: %d\n", x);
        x = 100;
        printf("P: %d\n", x);
        _exit(2);
    }
    func2();
}

void func1()
{
    int rc, status;
    printf("T: %d\n", x);
    x = 10;
    rc = fork();
    if (rc == 0) {
        x = 50;
        printf("Q: %d\n", x);
        exit(3);
    }
    rc = waitpid(rc, &status, 0)
    printf("A: %d\n", x);
    printf("D: %d", WEXITSTATUS(status));
    exit(4);
}

void func2()
{ printf("C: %d\n", x); }
```

T: 42

Q: 50

A: 10

D: 3

R: 4

M: 42

P: 100

### Exercice 2 (3 pts)

Supposons qu'un système d'exploitation utilise une stratégie à temps partagé avec priorité avec un quantum de 500 millisecondes. L'horloge matérielle génère une interruption de temporisateur chaque milliseconde. Supposons qu'un processus P est programmé pour s'exécuter et est distribué. Pendant son quantum, P ne fait aucun appel système. Cependant, les exceptions de traduction d'adresse se produisent une fois toutes les 10 millisecondes pendant que P est en cours d'exécution. Combien de fois au cours de son quantum le processus P entre-t-il dans le mode noyau ? Expliquez brièvement votre réponse.

**Chaque exception et interruption fait entrer le contrôle dans le noyau. Il y aura 500 interruptions de minuterie pendant le quantum de P, et il y aura  $500/10 = 50$  exceptions de traduction d'adresse, pour un total de 550 entrées de noyau.**

### Exercice 3 (11 pts)

*On se propose d'étudier une structuration pour des disques dont la capacité est comprise entre 100 Mo et 2 Go., les secteurs étant de 4096 octets.*

Les machines pour lesquelles ils sont destinés manipulent des octets, des entiers sur 16 bits ou des entiers sur 32 bits.

Les concepteurs du système de gestion de fichiers ont décidé de diviser chaque unité du disque de la façon suivante :

- *le descripteur du disque*, qui contient tous les renseignements nécessaires à son identification, les paramètres variables de la structure, et une table donnant accès aux fichiers.
- *la table d'allocation de l'espace*, représentant l'espace libre sur le disque.
- *la zone des fichiers*, qui contient les descripteurs de fichiers et l'espace de données de ces fichiers.

Un fichier est constitué de deux parties:

- *le descripteur de fichier* qui contient toutes les informations nécessaires à sa localisation, et qui seront détaillées ci-dessous.
- *l'ensemble des zones physiques*, qui contiennent les données du fichier. Pour un fichier, toutes les zones physiques du fichier ont la même taille, et sont constituées d'un nombre entier de secteurs contigus. Par contre les différentes zones d'un même fichier ne sont pas nécessairement contiguës.

Les descripteurs de fichiers doivent repérer l'ensemble des zones du fichier. Pour permettre à un fichier de contenir un nombre quelconque de zones, tout en ayant des descripteurs de taille fixe, les descripteurs sont décomposés en une partie principale (obligatoire) et des parties extensions (optionnelles). Les parties extensions éventuelles ont la même structure que les parties principales, les informations inutilisées étant simplement mises à 0 dans ce cas. Par la suite nous appellerons descripteur cette structure, qu'elle soit utilisée comme partie principale ou comme extension. Les descripteurs de fichiers sont regroupés dans des secteurs du disque. Ces secteurs sont chaînés entre eux, pour permettre la recherche des fichiers. Un descripteur de fichier a la structure suivante :

- le *nom* du fichier sur 16 octets,
- des informations diverses sur 12 octets,
- la *longueur du fichier*, c'est-à-dire le nombre total d'octets du fichier,
- le *nombre total de zones* du fichier,
- la *taille des zones* en nombre de secteurs,
- l'*"adresse"* de l'*extension suivante* (numéro de secteur et position dans le secteur),
- l'*"adresse"* de la dernière extension du fichier,
- la *table des numéros de premiers secteurs* des premières zones.

A.1- En considérant que tout entier sur disque a la même représentation qu'en mémoire centrale, c'est-à-dire 16 ou 32 bits, déterminer la taille en octets nécessaire à la représentation de chacune des informations suivantes:

- un numéro de secteur,
- la longueur du fichier, c'est-à-dire le nombre total d'octets du fichier,
- le nombre total de zones du fichier,
- la taille des zones en nombre de secteurs,
- l'*"adresse"* d'une extension (numéro de secteur et position dans le secteur).

- *numéro de secteur*. Un disque peut contenir jusqu'à  $2*2^{30} / 4*2^{10} = 2^{19}$  secteurs. Il faut donc au moins 19 bits pour représenter un numéro de secteur, soit un entier sur 32 bits, ou 4 octets.
- *longueur du fichier*. Un fichier ne peut dépasser la taille maximale d'un disque, donc  $2 * 2^{30} = 2^{31}$ . Il faut donc des entiers sur 32 bits, soit 4 octets.
- *nombre total de zones et taille des zones*. En l'absence de contrainte, une zone physique d'un fichier doit être d'au moins 1 secteur, mais pouvoir atteindre presque le disque complet (à

l'exception des informations structurelles). Un fichier peut donc avoir  $2^{19}$  zones de 1 secteur; il faut donc 32 bits ou 4 octets pour représenter le nombre total de zones du fichier. De même, un fichier peut avoir 1 zone de  $2^{19}$  secteurs; il faut donc 32 bits ou 4 octets pour représenter la taille des zones du fichier. Notons qu'il ne serait pas très contraignant et sans doute assez logique de limiter chacune de ces quantités à  $2^{16}$ , permettant ainsi d'utiliser des entiers sur 16 bits ou 2 octets, puisque ceci nous donne de toute façon  $2^{32}$  secteurs, valeur nettement supérieure au nombre maximal de secteurs d'un disque.

- *adresse d'une extension.* L'adresse d'une partie extension d'un descripteur doit permettre de désigner d'une part le numéro de secteur disque où la partie extension est placée (19 bits), et d'autre part, sa position dans ce secteur. Celle-ci peut être définie simplement par le numéro de son premier octet (12 bits), donnant ainsi un total de 31 bits. Cette position peut aussi être le numéro d'ordre du descripteur dans le secteur, puisque tous les descripteurs sont de même taille. Par exemple, s'il y a au plus 32 descripteurs par secteur, 5 bits suffiront alors pour ce numéro, et une adresse de partie extension nécessitera 24 bits. Enfin, en suivant la politique de représentation des entiers en 16 ou 32 bits, on constate alors qu'il faut 32 bits ou 4 octets dans tous les cas.

A.2- Déterminer la taille d'un descripteur en fonction du nombre d'entrées q de la table des premiers secteurs de zones dans un descripteur. En déduire la valeur maximale du nombre r de descripteurs dans les secteurs du disque qui les contiennent, en fonction de q, et la perte qui résulte d'un choix donné de r et q compatibles.

Pour déterminer le nombre de descripteurs de fichiers par secteur, il faut déterminer la taille d'un descripteur. Nous reprenons les données numériques de la question A.1, et les notons devant les informations du descripteur de fichier:

- 16 le nom du fichier sur 16 octets,
- 12 des informations diverses sur 12 octets,
- 4 le nombre total d'octets du fichier,
- 4 le nombre total de zones du fichier,
- 4 la taille des zones en nombre de secteurs,
- 4 l'“adresse” de l'extension suivante (numéro de secteur et position dans le secteur),
- 4 l'“adresse” de la dernière extension du fichier,
- $4 * q$  la table des numéros de premiers secteurs des premières zones.

Il s'ensuit que le descripteur occupe  $48 + 4 * q$  octets, où q est le nombre d'entrées de la table des premiers secteurs de blocs d'un descripteur. Les secteurs contenant des descripteurs étant chaînés entre eux, le lien de chaînage occupe 4 octets, laissant libres 4092 octets pour r descripteurs, et il s'ensuit que  $r \leq 4092 / (48 + 4 * q)$ . Pour une valeur de r compatible avec q, la perte est  $4092 - r * (48 + 4 * q)$ .

A.3- Étudier en particulier les cas où ce nombre q prend les valeurs 31, 32 et 33. Calculer la perte d'espace dans chaque cas. Commenter le choix de 32.

Nous avons les valeurs suivantes:

q	r maximum	perte
31	23	136

32	23	44
33	22	132

Le choix de 32 est le plus optimal, puisqu'il correspond à la plus petite perte. Cependant, même s'il induisait une perte supérieure aux autres, il pourrait être conservé car il a l'avantage de permettre de représenter, dans chaque descripteur, qu'il soit partie principale ou extension, un nombre de zones qui est une puissance de 2.

A.4- Expliquer l'intérêt des différentes informations (autres que diverses) qui sont présentent dans la partie principale d'un descripteur. On distinguera celles qui sont nécessaires et celles qui sont redondantes, et on justifiera cette redondance. Montrer que l'on peut déduire de ces informations le nombre d'extensions du descripteur de fichier.

Le *nom* du fichier est nécessaire pour distinguer les fichiers entre eux, et permet à l'utilisateur de désigner ce fichier par une chaîne de caractères. La *longueur du fichier* N est la seule information de taille qui permette de savoir exactement quels sont les octets de l'espace alloué au fichier qui en font partie. La *taille des zones* T est nécessaire pour déterminer la taille des zones à allouer au fur et à mesure des besoins au fichier; elle permet également de déterminer à quelle zone appartient le *n<sup>ième</sup>* secteur du fichier. L'*adresse de l'extension suivante* est nécessaire pour parcourir les extensions du descripteur. La *table des numéros de premiers secteurs* permet de savoir où commence chaque zone sur disque, et donc détermine l'espace alloué au fichier.

Le *nombre total de zones* du fichier est une information redondante, en ce sens qu'elle peut être reconstruite à partir des autres. En effet, le nombre de secteurs S est égal à  $(N-1)\%4096+1$ , où % désigne la division entière. À partir de la taille des zones T, on peut déduire le nombre de zones  $B = (S-1)\%T+1$ . Elle évite de refaire ce calcul en particulier pour déterminer la taille de l'espace occupé par le fichier sur le disque. Enfin l'*adresse de la dernière extension* du fichier est également une information redondante, puisqu'elle peut être connue en parcourant les extensions successives. Elle est utile car l'allocation d'une nouvelle zone lors d'un allongement du fichier, entraîne la modification de cette extension.

Par ailleurs  $(B-1)\%32$  nous donne le nombre d'extensions du fichier.

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define SHMSZ 27
6
7 void server() {
8     char c;
9     int shmid;
10    key_t key = 5678;
11    char *shm, *s;
12
13    // Creer le segment
14    shmid = shmget(key, SHMSZ, IPC_CREAT | 0666);
15    if(shmid < 0) { perror("shmget"); exit(1); }
16
17    // Attacher le segment
18    shm = shmat(shmid, NULL, 0);
19    if (shm == (char *) -1) { perror("shmat"); exit(1); }
20
21    // Mettre quelques choses dans la memoire pour l'autre processus
22    s = shm;
23    for (c = 'a'; c <= 'z'; c++) *s++ = c;
24    *s = NULL;
25
26    // On attend que le client lise en mettant en premier caractere '*'
27    while (*shm != '*') sleep(1);
28
29    shmdt((void*) shm);
30    shmctl(shmid, IPC_RMID, NULL);
31    exit(0);
32}
33
34 void client() {
35     int shmid;
36     char *shm, *s;
37
38     // Obtenir le segment "5678" cree par le serveur
39     key_t key = 5678;
40     shmid = shmget(key, SHMSZ, 0666);
41     if(shmid < 0) { perror("shmget"); exit(1); }
42
43     // Attacher le segment a notre espace de donnees
44     shm = shmat(shmid, NULL, 0)
45     if (shm == (char*) -1) { perror("shmat"); exit(1); }
46
47     // Lire ce que le serveur a mis dans la memoire
48     for (s = shm; *s != NULL; s++) putchar(*s);
49     putchar('\n');
50
51     // Changez le premier caractere du segment en '*' pour indiquer la lecture du segment
52     *shm = '*';
53     exit(0);
54 }

```

---

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define SHMSZ 27
6
7 int main(int, char**) {
8     int shmid;
9     char *shm, *s;
10
11    // Obtenir le segment "5678" cree par le serveur
12    key_t key = 5678;
13    shmid = shmget(key, SHMSZ, 0666);
14    if(shmid < 0) { perror("shmget"); exit(1); }
15
16    // Attacher le segment a notre espace de donnees
17    shm = shmat(shmid, NULL, 0)
18    if (shm == (char*) -1) { perror("shmat"); exit(1); }
19
20    // Lire ce que le serveur a mis dans la memoire
21    for (s = shm; *s != NULL; s++) putchar(*s);
22    putchar('\n');
23
24    // Changez le premier caractere du segment en '*' pour indiquer la lecture du segment
25    *shm = '*';
26    exit(0);
27 }

```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <unistd.h>
7 #include <sys/mman.h>
8
9 int main(int argc, char* argv[]) {
10     int i= 0, fd = open(filename, O_RDWR, 0666);
11     char* filename = "file.txt";
12     struct stat st;
13
14     stat(filename, &st);
15     long fileSize = st.st_size;
16     char* fileMap = (char*) mmap(NULL, fileSize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
17
18     if(fileMap == (char*) MAP_FAILED) { perror("mmap"); exit(1); }
19     close(fd);
20
21     while(i < fileSize/2) {
22         char c = fileMap[i];
23         fileMap[i] = fileMap[fileSize - i-1];
24         fileMap[fileSize - i-1] = c;
25         i++;
26     }
27
28     printf("%s\n", fileMap);
29     munmap((void*) fileMap, fileSize);
30     return 0;
31 }
32
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <unistd.h>
7 #include <sys/mman.h>
8 #define FILESIZE 10
9
10 int main(int argc, char* argv[]) {
11     int i= 0, fd = open("titi.dat", O_RDWR, 0666);
12
13     int* fileMap = (int*) mmap(NULL, FILESIZE*sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
14
15     if(fileMap == (int*) MAP_FAILED) { perror("mmap"); exit(1); }
16     close(fd);
17
18     while(1) {
19         scanf("%d", &i);
20         if(i == 99) break;
21         if(i < 10 && i >= 0) fileMap[i] = fileMap[i]+1;
22     }
23
24     munmap((void*) fileMap, FILESIZE*sizeof(int));
25     return 0;
26 }
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <unistd.h>
7 #include <sys/mman.h>
8 #define FILESIZE 10
9
10 int main(int argc, char* argv[]) {
11     int i= 0, fd = open("titi.dat", O_RDONLY, 0666);
12
13     int* fileMap = (int*) mmap(NULL, FILESIZE*sizeof(int), PROT_READ, MAP_PRIVATE, fd, 0);
14
15     if(fileMap == (int*) MAP_FAILED) { perror("mmap"); exit(1); }
16     close(fd);
17
18     while(1) {
19         scanf("%d", &i);
20         if(i == 99) break;
21         for(int j= 0; j< 10; j++) printf("\t%d\n", fileMap[j]);
22     }
23
24     munmap((void*) fileMap, FILESIZE*sizeof(int));
25     return 0;
26 }
27
28 #include <stdio.h>
29 #include <fcntl.h>
30 #include <sys/types.h>
31 #include <sys/shm.h>
32 #include <sys/wait.h>
33 #include <unistd.h>
34
35 #define READSIZE 1024
36
37 int copyFile(int f1, int f2) {
38     int totalByteRead= 0, lastByteRead= READSIZE;
39     char buffer[READSIZE];
40
41     while(lastByteRead == READSIZE) {
42         lastByteRead = read(f1, buffer, READSIZE);
43         write(f2, buffer, lastByteRead);
44         totalByteRead += lastByteRead;
45     }
46
47     return totalByteRead;
48 }
49
50 int main(int argc, char* argv[]) {
51     int file;
52     id_t id = shmemget(IPC_PRIVATE, sizeof(int), 0666);
53     int* shmArea = (int*) shmat(id, NULL, 0);
54
55     pid_t chld = fork();
56
57     if(chld == 0)    file = open((argc > 2) ? argv[2] : "file2", O_RDONLY);
58     else            file = open((argc > 1) ? argv[1] : "file1", O_RDONLY);
59
60     int byteCopied = copyFile(file, fileno(stdout));
61     close(file);
62
63     if(chld == 0) *shmArea = byteCopied;
64     else {
65         wait(NULL);
66         printf("Bytes copied: %d & %d\n", byteCopied, *shmArea);
67
68         shmdt((void*) shmArea);
69         shmctl(id, IPC_RMID, NULL);
70     }
71
72     return 0;
73 }
```

# 1 TD1

## 1.1 Ex1

P1 à t = 0	P2 à t = 1
Calcul pendant 3 unités	
Impression sur imprimante pendant 2 unités	Calcul pendant 2 unités
Calcul pendant 2 unités	Affichage à l'écran pendant 2 unités
Affichage à l'écran pendant 1 unité	Calcul pendant 1 unité
Calcul pendant 3 unités	Impression sur imprimante pendant 2 unités
FIN.	FIN.

### 1) Mono-programmé

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

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					$p_1$	$p_1$								
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							$p_2$	$p_2$							
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 
$$\begin{array}{c|c|c|c|c|c|c} \text{EX} & | & \text{A} & | & \text{B} & | & \text{C} & | & \text{D} & | & \text{H} \\ \hline \text{x} & | & \text{x} \end{array}$$

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 argc, 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 argc, 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

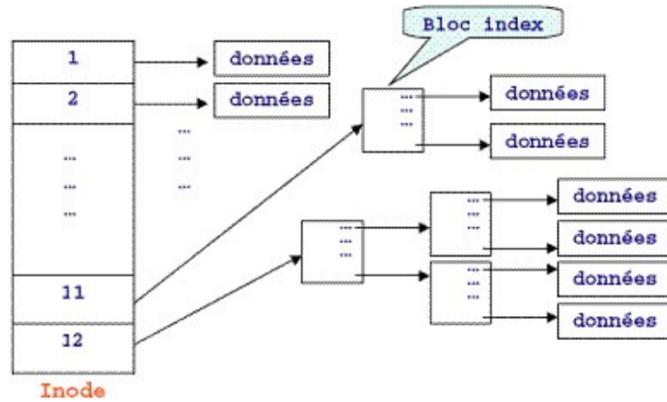
- Accès au répertoire, écriture sur le disque : 2
- Accès au répertoire : 1

### 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;
}
```