Systèmes d'exploitation

Troisième année de la licence d'informatique
UFR Sciences, sites de Luminy et d'Aix-en-Provence,
Aix-Marseille Université
Lundi 12 janvier 2026
durée 1h30, calculatrices et documents autorisés

Responsables :
J.-L. Massat et B. Bergougnoux

1 Gestion des processus (9 points, 40 mn)

Considérons un système équipé d'un disque et de **trois processeurs** gérés par l'algorithme du **tourniquet**. Travaillons sur le processus ci-dessous :

```
begin b = bb(); x = aa(b); y = cc(b); dd(x, y); end
```

Les fonctions sont toutes indépendantes et ont le comportement ci-dessous :

`aa` et `cc` 3s de CPU suivies de 1s d'E/S `bb` 2s de CPU suivies de 2s d'E/S
`dd` 1s de CPU

A) Donnez le temps de réponse (TR) et le taux d'utilisation des processeurs (TX). **[2 points]**

```
     TR = 13    TX = 9/39 = 23,0%
---- :    1  2  3  4  5  6  7  8  9  10 11 12 13
CPU0 : |bb|bb|--|--|aa|aa|aa|--|cc|cc|cc|--|dd|
E/S  : |--|--|bb|bb|--|--|--|aa|--|--|--|cc|--|
```

B) Afin d'exploiter nos processeurs, nous préparons la version ci-dessous. Donnez TR et TX pour ce nouveau code. **[3 points]**

```
begin
  b = null;
  cobegin // exécuter en parallèle dans plusieurs threads
    b = bb();
    begin
      tant-que (b == null) consommer de la CPU;
      x = aa(b);
    end
    begin
      tant-que (b == null) consommer de la CPU;
      y = cc(b);
    end
  coend // fin des threads
  dd(x, y);
end
```

```
     TR = 10    TX = (9+8)/30 = 56,7%
---- :    1  2  3  4  5  6  7  8  9  10
CPU0 : |bb|bb|--|--|--|--|--|--|--|dd|
CPU1 : |tq|tq|tq|tq|aa|aa|aa|--|--|--|
CPU2 : |tq|tq|tq|tq|cc|cc|cc|--|--|--|
E/S  : |--|--|bb|bb|--|--|--|aa|cc|--|
```

C) Proposez une nouvelle version basée sur un sémaphore qui permet d'éviter la perte CPU de l'attente active. **[2 points]**

```
begin
  sb = sémaphore(0);
  cobegin // exécuter en parallèle dans plusieurs threads
    begin b = bb(); V(sb); V(sb); end
    begin P(sb); x = aa(b); end
    begin P(sb); y = cc(b); end
  coend // fin des threads
  dd(x, y);
end
```

D) Quels sont TR et TX pour la version avec semaphore ? **[1 point]**

```
     TR = 12    TX = 9/30 = 30,0%
---- :    1  2  3  4  5  6  7  8  9  10 11 12
CPU0 : |bb|bb|--|--|aa|aa|aa|--|--|dd|
CPU1 : |--|--|--|--|cc|cc|cc|--|--|--|
CPU2 : |--|--|--|--|--|--|--|--|--|--|
E/S  : |--|--|bb|bb|--|--|--|cc|aa|--|
```

E) Pouvez-vous donner une version aussi efficace mais sans utiliser de sémaphore ? **[1 point]**

```
begin
  b = bb();
  cobegin x = aa(b); y = cc(b); coend
  dd(x, y);
end
```

2 Gestion de fichiers (4 points, 20 mn)

On vous donne l'instance d'une FAT suivante (-1 signale le dernier bloc d'un fichier, et -2 un bloc libre).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FAT[i]	-1	-2	-2	-2	-2	15	-1	6	10	-2	13	12	-1	11	8	-1

A) Dans cette configuration combien avons-nous de fichiers et quels sont les blocs physiques de chaque fichier ? **[2 points]**

```
Il y a 4 fichiers, voilà les blocs de ces fichiers :
- 0.
- 7, 6.
- 5, 15.
- 14, 8, 10, 13, 11, 12.
```

B) Écrivez `void free_blocks(int start, int FAT[FAT_SIZE])` qui désalloue les blocs physiques d'un fichier dont le premier bloc physique est `start`. **[2 points]**

```
int i = start;
while(i != -1){
  int next = FAT[i];
  FAT[i] = -2;
  i = next;
}
```

3 Gestion de la mémoire (7 points, 30 mn)

Considérons un système qui organise la mémoire de chaque processus en deux segments (code et données). Nous avons dans le processeur les quatre registres associés suivants :

`CS` : code segment, `CZ` : code size, `DS` : data segment, `DZ` : data size.

A) Si nous avons `CS=125,CZ=2000,DS=3050,DZ=10240`, quelles sont les six adresses physiques générées par l'exécution du code ci-dessous. **[2 points]**

```
 8 : load R1, 550   | load : chargement d'une case mémoire
12 : store R1, 258  | store : sauvegarde dans une case mémoire
16 : jump 250       | jump : saut inconditionnel
```

```
- Code :  8+CS, 12+CS, 16+CS, 250+CS
- Data :  550+DS, 258+DS
```

B) Que se passe-t-il lors de l'exécution de l'instruction suivante. **[1 point]**

```
250 : jump 3025     | jump : saut inconditionnel
```

```
Le processeur génère une erreur d'adressage car 3025 > 2000.
```

À partir de maintenant, considérons un autre système qui utilise la pagination (sans segmentation) pour organiser la mémoire de chaque processus. Ce système utilise des pages de taille 256 octets et des adresses sur 16 bits, les numéros de page sont codés avec les 8 bits de poids forts, tandis que les déplacements sont codés avec les 8 bits de poids faible.

C) Avec ce nouveau système, quelles sont les six adresses physiques générées par l'exécution du code de la question **A)** dans le cas où le processus courant a la table des pages ci-dessous. **[2 points]**

Index de la table :	0	1	2	3	4
Contenu de la table :	10	20	100	42	78

```
8 est sur la page logique 0, le déplacement est de 8.
La page logique 0 est stockée sur la plage physique 10.
Donc 8 est associé à l'adresse physique (10*256)+8.
De la même manière, 12, 16 et 250 sont associées respectivement à :
(10*256)+12,  (10*256)+16 et (10*256)+250.

258 est sur la page logique 1 avec un déplacement de 2 = 258-(1*256).
Donc 258 est associé à (20*256)+2.

550 est sur la page logique 2 avec un déplacement de 38 = 550-(2*256).
Donc 550 est associé à (100*256)+38.
```

B) Expliquez brièvement pourquoi la pagination ralentit les accès à la mémoire centrale et comment on peut remédier à ce problème. **[2 points]**

Accéder à une adresse logique demande 2 accès à la mémoire centrale.
Car on doit accéder à une entrée dans la page des tables.

Pour remédier à ça, on peut utiliser la mémoire associative et
le principe de localité : un processus travaille sur peu de page
à la fois et son activité présente est une bonne estimation
de son activité future.

Système d'exploitation

Responsables :
J.-L. Massat et B. Bergougnoux

Troisième année de la licence d'informatique
UFR Sciences, sites de Luminy et d'Aix-en-Provence,
Aix-Marseille Université
jeudi 9 janvier 2025
durée 1h30, calculatrices et documents autorisés

1 Alice, Eve et Bob (6 points, 25 mn)

Trois processus : Alice, Eve et Bob partagent un buffer `buff`. Bob souhaite envoyer successivement des messages à Alice et Eve. Avant d'envoyer un nouveau message, Bob doit s'assurer que son message précédent a bien été lu par Alice et Eve.

A) Le code suivant tente de résoudre cette problématique. Les variables B et L sont des sémaphores. B est initialisé à 2 et L à 0. Avec cette solution, il peut arriver qu'Eve loupe un message de Bob. Pourquoi ? Argumentez en décrivant une exécution erronée. **[2 points]**

Code de Bob :

```
while (1) {
   P(B) ; P(B) ;
   < écrire dans buff >;
   < écrire dans buff >;
   V(L) ; V(L) ;
}
```

Code d'Alice et de Eve :

```
while (1) {
   P(L) ;
   < lire depuis buff >;
   V(B) ;
}
```

- les messages dans le buffer sont indifférenciés
- Bob: ... V(L); V(L); ...; Eve: P(L); Lire; V(B); P(L); Lire; V(B)

B) Présentez une solution correcte (avec trois sémaphores à définir). **[2 points]**

B: sémaphore(1); A: sémaphore(0); E: sémaphore(0);
- Bob: while (1) { P(B); Ecrire; Ecrire; V(A); }
- Alice: while (1) { P(A); Lire; V(E); }
- Eve: while (1) { P(E); Lire; V(B); }

C) Finalement, Bob préfère utiliser deux buffers séparés (`buffAlice` et `buffEve`). Comment reformuler le code de ces trois acteurs (et les sémaphores à definir) ? **[2 points]**

```
bufferAlice: sémaphore(1), bufferEve: sémaphore(1);
messageAlice: sémaphore(0), messageEve: sémaphore(0);

Code Bob:
   while (1) {
      P(bufferAlice); Ecrire(buffAlice); V(messageAlice);
      P(bufferEve);   Ecrire(buffEve);   V(messageEve);
   }

Code d'Alice:
   while (1) { P(messageAlice); Lire(buffAlice); V(bufferAlice); }

Code d'Eve:
   while (1) { P(messageEve); Lire(buffEve); V(bufferEve); }
```

2 Gestion de la mémoire (6 points, 25 mn)

Considérons un système qui organise la mémoire de chaque processus en trois segments (code, données et pile). Nous avons dans le processeur six registres associés (`CS` : code segment, `CZ` : code size, `DS` : data segment, `DZ` : data size, `SS` : stack segment, `SZ` : stack size).

A) Pourquoi prévoir trois segments (donnez quatre arguments dont au moins un pour `SS/SZ`) ? **[2 points]**

```
- pour appliquer des protections différentes
- pour avoir des segments plus petits et plus facile à allouer
- pour pouvoir partager le segment de code
- pour gérer des mémoire plus grande
- pour séparer les piles des threads
```

B) Pouvez-vous donner la signification des registres `CS`, `DS` et `SS` ? **[2 points]**

```
- CS: adresse physique du début du code en mémoire centrale
- DS: adresse physique du début des données en mémoire centrale
- SS: adresse physique du début de la pile en mémoire centrale
```

C) Si nous avons `CS=2048,CZ=3096,DS=614400,DZ=10240,SS=8192,SZ=1024` et un pointeur de pile `SP=512`, quelles sont les six adresses physiques générées par l'exécution du code ci-dessous. **[2 points]**

```
2040 : load R1, 6100        | load : chargement d'une case mémoire
2044 : push 562             | push : empiler la valeur 562
2048 : jump 2108            | jump : saut inconditionnel
```

```
- Code:  2040+CS, 2044+CS, 2048+CS, 2108+CS
- Data:  6100+DS,
- Stack: SS+SP = 8192+512
```

3 Gestion des processus (8 points, 40 mn)

Considérons un système ou le **processeur unique** est géré par l'algorithme du **tourniquet** et travaillons sur le processus ci-dessous :

```
calculer_1s_puis_attendre_2s_puis_calculer_1s();  // Travail 1
calculer_2s_puis_attendre_3s_puis_calculer_1s();  // Travail 2
calculer_3s_puis_attendre_4s_puis_calculer_1s();  // Travail 3
calculer_1s();                                    // Travail final
```

A) Donnez le temps de réponse (TR) et le taux d'utilisation du processeur (TX). **[2 points]**

```
Ti = travail i              TR = 19
CF = code final             TX = 10/19
---- :    1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
EXEC : |T1|--|--|T1|T2|T2|--|--|--|T2|T3|T3|T3|--|--|--|--|T3|CF|
WAIT : |--|T1|T1|--|--|--|T2|T2|T2|--|--|--|--|T3|T3|T3|T3|--|--|
```

B) Avec le code ci-dessous, donnez TR et TX si nous considérons que les attentes peuvent se dérouler en parallèle. **[3 points]**

```
créer trois threads pour faire les travaux 1, 2 et 3   // Travaux 1,2,3
attendre la fin des trois threads
calculer_1s();                                         // Travail final
```

```
---- :    1  2  3  4  5  6  7  8  9  10 11 12 13
EXEC1: |T1|T1|T1|--|--|T1|T1|--|--|--|--|--|--|
WAIT1: |--|--|--|T1|T1|--|--|--|--|--|--|--|--|

EXEC2: |T2|T2|T2|T2|T2|--|--|--|T2|--|--|--|--|     TR = 13
WAIT2: |--|--|--|--|--|T2|T2|T2|--|--|--|--|--|     TX = 10/13

EXEC3: |T3|T3|T3|T3|T3|T3|T3|--|--|--|--|T3|--|
WAIT3: |--|--|--|--|--|--|--|T3|T3|T3|T3|--|--|

CF   : |--|--|--|--|--|--|--|--|--|--|--|--|CF|
```

C) Même question que précédemment si nous disposons de trois processeurs. **[2 points]**

```
---- :    1  2  3  4  5  6  7  8  9
T1   : |C1|W1|W1|C1|--|--|--|--|--|     TR = 9
T2   : |C2|C2|W2|W2|W2|C2|--|--|--|     TX = 10/27
T3   : |C3|C3|C3|W2|W2|W2|W3|C3|--|
CF   : |--|--|--|--|--|--|--|--|CF|
```

D) Quelle type d'opération peut se cacher derrière ces attentes (donnez au moins deux exemples et un contre exemple) ? **[1 point]**

```
Exemples : - E/S clavier et/ou souris, - E/S reseau, - sleep
Contre exemples : - E/S disque, - E/S imprimante (ressource critique)
```

Système d'exploitation

Responsable : Jean-Luc Massat

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
21 décembre 2023
durée 1h30, calculatrices et documents autorisés

1 Simuler l'ordonnancement de processus (12 points, 50 mn)

Nous souhaitons **simuler** différentes stratégies d'allocation du processeur afin d'en évaluer les performances. Pour ce faire, nous définissons les données ci-dessous pour représenter un ensemble de processus qui ne réalisent que du calcul :

```
typedef enum {FUTUR, PRET, TERMINE} ETAT;
typedef enum {FIFO, SJF} STRATEGIE;

typedef struct {
    ETAT etat;          // état du processus
    long total_cpu;     // besoin total en CPU
    long besoin_cpu;    // besoin restant en CPU (besoin <= total)
    long date_arrivee;  // date d'arrivée du processus
    long date_fin;      // date de fin du processus
    long date_requi;    // date de la dernière réquisition
} PROCESSUS;

int nb_processus = 0;
PROCESSUS proc[MAX_PROCESSUS];
```

A) Si vous considérez que ce tableau est rempli avec des processus (certains n'ont pas démarré, d'autres sont en cours, d'autres enfin sont terminés), proposez une implantation pour la fonction `int ordonnancer_FIFO()` qui renvoie le numéro du processus choisi (ou -1). **[2 points]**

```
int ordonnancer_FIFO() {
    int older = -1;
    for(int k=0; (k < nb_processus); k++) {
        if (proc[k].etat != PRET) continue;
        if (older < 0) {
            older = k;
        } else if (proc[k].date_requi < proc[older].date_requi) {
            older = k;
        }
    }
    return older;
}
```

B) Même contexte et même question pour la fonction `int ordonnancer_SJF()`. **[3 points]**

```
int ordonnancer_SJF() {
    int min = -1;
    for(int k=0; (k < nb_processus); k++) {
        if (proc[k].etat != PRET) continue;
        if (min < 0) {
            min = k;
        } else if (proc[k].besoin_cpu < proc[min].besoin_cpu) {
            min = k;
        } else if (proc[k].besoin_cpu == proc[min].besoin_cpu) {
            if (proc[k].date_requi < proc[min].date_requi) {
                min = k;
            }
        }
    }
    return min;
}
```

C) Pour simuler nous avons besoin d'un jeu d'essais. Nous vous demandons de coder une version de la fonction `void generer()` qui prépare 300 fois trois processus (A d'une durée de 2, B d'une durée de 4 et C de 10). B démarre 1 unité de temps après A et C démarre 2 unités après B. Ce triplet recommence toutes les 10 unités de temps. **[2 points]**

```
void generer() {
    long temps = 0;
    for(int k=0; k<300; k++) {
        proc[nb_processus].total_cpu = 2;
        proc[nb_processus].date_arrivee = temps;
        proc[nb_processus].besoin_cpu = 2;
        proc[nb_processus].etat =  FUTUR;
        nb_processus++;
        proc[nb_processus].total_cpu = 4;
        proc[nb_processus].date_arrivee = (temps + 1);
        proc[nb_processus].besoin_cpu = 4;
        proc[nb_processus].etat =  FUTUR;
        nb_processus++;
        proc[nb_processus].total_cpu = 10;
        proc[nb_processus].date_arrivee = (temps + 3);
        proc[nb_processus].besoin_cpu = 10;
        proc[nb_processus].etat =  FUTUR;
        nb_processus++;
        temps += 10;
    }
}
```

D) Nous pouvons maintenant simuler l'exécution d'un processus. Proposez une version de la fonction `long executer(int p, long maintenant, long duree_tranche)` qui simule l'exécution du processus `p` à la date `maintenant` avec une tranche de temps qui mesure `duree_tranche`. Cette fonction renvoie la durée **effectivement utilisée** par le processus (il n'a pas forcément besoin de toute la tranche). Cette exécution peut, éventuellement, terminer le processus. **[3 points]**

```
#define MIN(a,b)   ((a)<(b)?(a):(b))

long executer(int p, long maintenant, long duree_tranche) {
    assert(proc[p].etat == PRET);

    duree_tranche = MIN(duree_tranche, proc[p].besoin_cpu);
    proc[p].besoin_cpu -= duree_tranche;
    proc[p].date_requi = (maintenant + duree_tranche);

    if (proc[p].besoin_cpu <= 0) {   // fin du processus
        proc[p].etat = TERMINE;
        proc[p].date_fin =  (maintenant + duree_tranche);
    }
    return duree_tranche;
}
```

E) Nous pouvons maintenant simuler l'exécution de tous les processus. Proposez une version de la fonction `void executer_tout(long duree_tranche, STRATEGIE strategie)` qui simule l'exécution de tous les processus. Vous devrez utiliser la fonction ci-dessous : **[2 points]**

```
void demarrer_processus(long maintenant) {
    for(int k=0; (k < nb_processus); k++) {
        if (proc[k].etat != FUTUR) continue;
        if (proc[k].date_arrivee > maintenant) continue;
        // à reveiller
        proc[k].etat = PRET;
        proc[k].date_requi = proc[k].date_arrivee;
    }
}
```

```
void executer_tout(long duree_tranche, STRATEGIE st) {
   generer();
   long temps = 0;
   for (;;) {
      demarrer_processus(temps);
      int p = (st == FIFO) ? ordonnancer_FIFO() : ordonnancer_SJF();
      if (p < 0) break;
      long duree_utilisee = executer(p, temps, duree_tranche);
      temps += duree_utilisee;
   }
}
```

2 Gestion de la mémoire (4 points, 20 mn)

Considérons un système qui organise la mémoire de chaque processus en deux segments (code et données). Nous avons dans le processeur quatre registres associés (`CS` : code segment, `CZ` : code size, `DS` : data segment, `DZ` : data size).

A) Pourquoi prévoir deux segments (donnez trois arguments) ? **[1 point]**

```
- pour appliquer des protections différentes
- pour avoir des segments plus petits et plus facile à allouer
- pour pouvoir partager le segment de code
```

B) Pouvez-vous donner la signification de ces quatre registres ? **[1 point]**

```
- CS: adresse physique du début du code en mémoire centrale
- CZ: taille du segment de code en octets
```

C) Si nous avons `CS=2048,CZ=3096,DS=614400,DZ=10240`, quelles sont les six adresses physiques générées par l'exécution du code ci-dessous. **[2 points]**

```
1200 : load R1, 6100        | load : chargement d'une case mémoire
1204 : store R1, 7800       | store : modification d'une case mémoire
1208 : jump 2108            | jump : saut inconditionnel
```

```
- 1200+CS, 1204+CS, 1208+CS, 2108+CS
- 6100+DS, 7800+DS
```

3 Gestion des processus et synchronisation (4 points, 20 mn)

Considérons un système ou les **processeurs** sont gérés par l'algorithme du **tourniquet** et travaillons sur le processus ci-dessous :

```
faire quatre fois
   |  créer un thread afin d'exécuter un calcul de trois secondes
fin-faire
attendre la fin des quatre threads fils
calculer pendant deux secondes
```

A) Donnez le temps de réponse (TR) et le taux d'utilisation des processeurs (TX) si nous avons deux, trois et quatre processeurs. **[2 points]**

```
Avec TT = mélange des T1, T2, T3 et T4 sur les 2 ou 3 CPUs

---- :    1  2  3  4  5  6  7  8     TR=8, TX=14/16
CPU1 : |TT|TT|TT|TT|TT|TT|XX|XX|
CPU2 : |TT|TT|TT|TT|TT|TT|--|--|

---- :    1  2  3  4  5  6           TR=6, TX=14/18
CPU1 : |TT|TT|TT|TT|XX|XX|
CPU2 : |TT|TT|TT|TT|--|--|
CPU3 : |TT|TT|TT|TT|--|--|

---- :    1  2  3  4  5              TR=5, TX=14/20
CPU1 : |T1|T1|T1|XX|XX|
CPU2 : |T2|T2|T2|--|--|
CPU3 : |T3|T3|T3|--|--|
CPU4 : |T4|T4|T4|--|--|
```

B) Donnez TR et TX si nous avons deux processeurs, un sémaphore `mutex` initialisé à `1` et que le code de trois secondes devient (donnez la trace) ? **[2 points]**

```
calculer 1s ; P(mutex); calculer 1s ; V(mutex); calculer 1s
```

```
Avec Thread i : Ai 1s ; P(mutex); Bi 1s ; V(mutex); Ci 1s
     AA = mélange des A1, A2, A3 et A4 sur les deux CPU

---- :    1  2  3  4  5  6  7  8  9    TR=9, TX=14/18
CPU1 : |AA|AA|B1|B2|B3|B4|--|XX|XX|
CPU2 : |AA|AA|--|C1|C2|C3|C4|--|--|
```

Système d'exploitation

Responsable : Jean-Luc Massat

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
15 décembre 2022
durée 1h30, calculatrices et documents autorisés

1 Ordonnancement de processus (7 points, 35 mn)

Considérez un système d'ordonnancement qui utilise les structures de données ci-dessous.

```
enum {MAX_PROCESS=10, NO_PROCESS=-1};
typedef enum {EMPTY=0, READY=1} STATE;
typedef enum {HIGH=0, LOW=1} PRIORITY;

struct PCB {
    PSW cpu;              // Processor Status Word (non explicité)
    STATE state;          // EMPTY or READY
    PRIORITY priority;    // HIGH or LOW
    int time;             // Estimated execution time
} process[MAX_PROCESS];

int current_process = NO_PROCESS;
```

L'algorithme d'ordonnancement de ce système est basé sur le principe de **Files de priorité**. Les processus sont classés sur 2 niveaux de priorités distincts : `HIGH` et `LOW`. La file `HIGH` gére les processus de priorité **haute**. Cette file utilise l'algorithme *SJF - (Shortest Job First)* pour ordonnancement. La file `LOW` gère les processus de priorité **basse** et utilise l'algorithme *RR - (Round Robin ou Tourniquet)*. Les processus de priorité `HIGH` sont prioritaires par rapport aux processus de priorité `LOW`.

A) Écrivez la fonction d'ordonnancement `int scheduler_HIGH()` des processus de priorité *haute*. La fonction renvoie l'indice du processus choisi ou `NO_PROCESS`. **[2 points]**

```
int scheduler_HIGH() {
    int best = NO_PROCESS;
    for(int i=0; (i<MAX_PROCESS); i++) {
        if (process[i].state != READY) continue;
        if (process[i].priority != HIGH) continue;
        if (best < 0 || process[i].time < process[best].time) {
            best = i;
        }
    }
    return best;
}
```

B) Écrivez la fonction d'ordonnancement `int scheduler_LOW()` des processus de priorité *basse*. La fonction renvoie l'indice du processus choisi ou `NO_PROCESS`. **[2 points]**

```
int scheduler_LOW() {
    static int fifo = -1;
    for(int i=0; (i<MAX_PROCESS); i++) {
        fifo = (fifo + 1) % MAX_PROCESS;
        if (process[fifo].state != READY) continue;
        if (process[fifo].priority != LOW) continue;
        return fifo;
    }
    return NO_PROCESS;
}
```

C) Écrivez la fonction générale d'ordonnancement `PSW scheduler(PSW cpu)` basée sur la structure et le principe des **Files de priorité** décrits ci-dessus. Cette fonction doit sauvegarder le processus courant, restaurer et retourner le processus choisi. Cette fonction utilise les deux fonctions précédentes. La fonction fixe `current_process` à `NO_PROCESS` s'il n'y a plus aucun processus à ordonnancer. **[3 points]**

```
PSW scheduler(PSW cpu) {
    if (current_process != NO_PROCESS) {
        process[current_process].cpu = cpu;
    }
    int current_process = scheduler_HIGH();
    if (current_process == NO_PROCESS) {
        current_process = scheduler_LOW();
    }
    if (current_process == NO_PROCESS) {
        return cpu;
    }
    return process[current_process].cpu;
}
```

2 Gestion de fichiers à trous (7 points, 30 mn)

Considérons un disque géré à l'aide d'une FAT (*File Allocation Table*). Le numéro permet de coder des fichiers à trous (tous les numéros logiques ne correspondent pas à des blocs physiques). Si `fat[`P`].num =` L, alors le bloc logique L est stocké dans le bloc physique P. **Attention** : le chaînage de la FAT ne suit pas forcément l'ordre logique des blocs.

```
struct {
    int next;          /* adr. phy. du bloc suivant (-1 si dernier) */
    int num;           /* numéro du bloc logique                    */
} fat[ NB_BLOCS ];     /* la FAT chargée en mémoire                 */
```

Dans l'exemple ci-dessous vous pouvez retrouver un fichier qui commence à **l'adresse 4** et qui est constitué de **trois** blocs physiques. Les blocs libres sont signalés par le champ `next` à -2.

	0	1	2	3	4	5	6	7	8	9	10	11	12
next	-2	-1	-2	11	8	-2	-1	-2	1	-2	-2	12	6
num	-	0	-	3	9	-	10	-	7	-	-	4	5

A) Retrouvez dans l'exemple l'autre fichier, son nombre de blocs logiques et physiques. **[2 points]**

```
blocs physiques/logiques : 3/3, 11/4, 12/5, 6/10,
blocs logiques : 11 max, 4 réels, 7 trous
```

B) Écrivez `int nb_holes(int first)` qui calcule le nombre de trous d'un fichier (des blocs logiques qui ne correspondent à aucun bloc physique). **[2,5 points]**

```
int nb_holes(int first) {
    int max = -1;
    int counter = 0;
    while (first >= 0) {
        counter++;
        if (fat[first].num > max) {
            max = fat[first].num;
        }
        first = fat[first].next;
    }
    return (max + 1 - counter);
}
```

C) Écrivez `int physical_address(int first, int num)` qui renvoie l'adresse physique d'un bloc logique `num`. Cette fonction renvoie -1 si le bloc en question n'existe pas (tentative de lecture dans un trou). **[2,5 points]**

```
int physical_address(int first, int num) {
    while (first >= 0) {
        if (fat[first].num == num) return first;
        first = fat[first].next;
    }
    return -1;
}
```

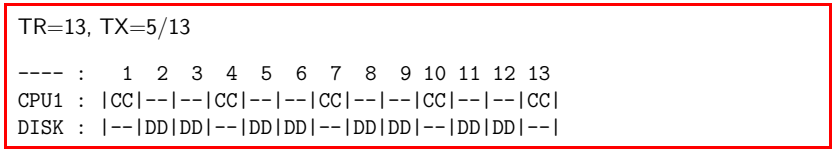
3 Gestion des processus et synchronisation (6 points, 25 mn)

Considérons un système équipé d'un **seul disque** et d'un seul **processeur** géré en temps partagé par l'algorithme du **tourniquet**.

A) Donnez le temps de réponse du processus ci-dessous et le taux d'utilisation du processeur. Indiquez clairement le déroulement des opérations sur le processeur et le disque. **[2 points]**

```
faire quatre fois
  |  calculer pendant une seconde
  |  écrire sur un fichier pendant 2 secondes
fin-faire
calculer pendant une seconde
```

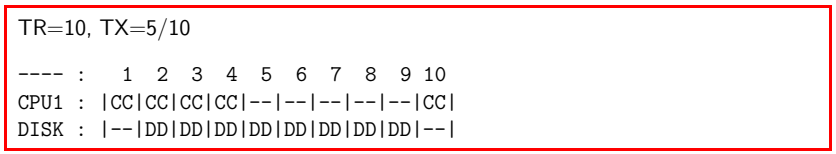
TR=13, TX=5/13



B) Même question avec les mêmes détails si nous utilisons une écriture asynchrone gérée par un thread séparé. **[2 points]**

```
faire quatre fois
  |  calculer pendant une seconde
  |  créer un thread pour faire
  |     |  écrire sur un fichier pendant 2 secondes
  |  fin du thread
fin-faire
attendre la fin des threads
calculer pendant une seconde
```

TR=10, TX=5/10



C) Quels sont les résultats si nous ajoutons un deuxième disque, que nous montons nos deux disques en RAID 0 (stripping) et que nous continuons à utiliser un thread pour gérer les écritures **[2 points]**.

TR=6, TX=5/6

Système d'exploitation

Responsable : Jean-Luc Massat

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
16 décembre 2021,
durée 1h30, calculatrices et documents autorisés

1 Améliorer un système (6 points, 25 mn)

Dans les systèmes Unix/Linux la création de processus et l'attente de la fin de ses fils passent par les deux fonctions `fork` et `wait` :

```
int main (void) {
    if (fork() == 0) {              // créer un processus fils
        printf("Fils\n");           // je suis le fils
        return EXIT_FAILURE;        // la fin du fils
    }
    printf("Père\n");               // je suis le père
    wait(NULL);                     // attendre la mort de son fils
    printf("Fin du père\n");        // le père a le dernier mot
    return EXIT_SUCCESS;
}
```

Nous souhaitons faire la même chose dans **notre mini-système développé en TP**.

A) Commençons par ajouter un appel système permettant à un processus d'attendre la mort d'un de ses fils (comme le `wait` de l'exemple). Donnez les deux étapes principales de la réalisation de cette attente ainsi que les données supplémentaires dont vous avez besoin ? [4 points].

```
Données:
- ajouter un état WAIT
- ajouter, pour chaque processus, le numéro de son père
WAIT:
- si ce processus a un fils, le mettre dans l'état WAIT
- choisir un autre processus
```

B) Comment modifier l'appel système `EXIT` afin de reveiller l'éventuel père en attente ? Donnez les trois étapes principales [2 points].

```
EXIT:
- chercher son père
- si il est WAIT, le passer à READY
- si le processus moribond a des fils, supprimer la liaison
```

2 Gestion de fichiers (6 points, 25 mn)

Dans un système très simplifié, la FAT d'un disque unique pourrait être représentée par les structures de données ci-dessous :

```
enum { FAT_EOF = -1, FAT_FREE = -2, FAT_RESERVED = -3 };

struct {
    int first;          // adresse du premier bloc (FAT_EOF si vide)
    int size;           // taille en octets (-1 si inutilisé)
} inodes[ NB_FILES ];   // les descripteurs

int fat[ NB_BLOCKS ];   // la FAT du disque unique
```

A) Écrivez la fonction `int count_free_blocks(void)` qui va calculer et renvoyer le nombre de blocs libres. [2 points]

```
int count_free_blocks() {
    int nb = 0;
    for(int i=0; i<NB_BLOCKS; i++) {
        if (fat[i] == FAT_FREE) nb++;
    }
    return nb;
}
```

B) Écrivez la fonction `int count_data_blocks(void)` qui va calculer et renvoyer le nombre de blocs **effectivement** occupés par tous les fichiers. [2 point]

```
int count_data_blocks() {
    int nb = 0;
    for(int i=0; i<NB_FILES; i++) {
        if (inodes[i].size < 0) continue;
        int first = inodes[i].first;
        while (first >= 0) {
            nb++;
            first = fat[first];
        }
    }
    return nb;
}
```

C) Finalement, écrivez une fonction qui calcule le nombre de blocs perdus (marqués comme utilisés mais qui n'apparaissent dans aucun fichier). [2 point]

```
int count_lost_blocks() {
    int nb = 0;
    for(int i=0; i<NB_BLOCKS; i++) {
        if (fat[i] >= 0) nb++;
        else if (fat[i] == FAT_EOF) nb++;
    }
    return nb - count_data_blocks();
}
```

3 Gestion des processus et synchronisation (8 points, 40 mn)

Considérons un système équipé d'un **seul disque** et d'un seul **processeur** géré en temps partagé par l'algorithme du **tourniquet**.

A) Donnez le temps de réponse du processus ci-dessous et le taux d'utilisation du processeur. Indiquez clairement le déroulement des opérations sur le processeur et le disque. [2 points]

```
faire quatre fois
  |  calculer pendant une seconde
  |  écrire sur un fichier pendant 3 secondes
fin-faire
calculer pendant une seconde
```

TR=17, TX=5/17

```
---- :   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
CPU1 : |CC|--|--|--|CC|--|--|--|CC|--|--|--|CC|--|--|--|CC|
DISK : |--|DD|DD|DD|--|DD|DD|DD|--|DD|DD|DD|--|DD|DD|DD|--|
```

B) Même question avec les mêmes détails si nous utilisons une écriture asynchrone gérée par un thread séparé. [2 points]

```
faire quatre fois
  |  calculer pendant une seconde
  |  créer un thread pour faire
  |     |  écrire sur un fichier pendant 3 secondes
  |  fin du thread
fin-faire
attendre la fin des threads
calculer pendant une seconde
```

TR=14, TX=5/14

```
---- :   1  2  3  4  5  6  7  8  9 10 11 12 13 14
CPU1 : |CC|CC|CC|CC|--|--|--|--|--|--|--|--|--|CC|
DISK : |--|DD|DD|DD|DD|DD|DD|DD|DD|DD|DD|DD|DD|--|
```

C) Reprenez le code de la question **B** et remplacez l'instruction d'attente des threads par une version basée sur un sémaphore (à créer, initialiser et utiliser). [2 points]

```
finThreads := semaphore( 0 );
faire quatre fois
  |  calculer pendant une seconde
  |  créer un thread pour faire
  |     |  écrire sur un fichier pendant 3s
  |     |  V(finThreads)
  |  fin du thread
fin-faire
faire 4 fois P(finThreads); fin-faire
calculer pendant une seconde
```

D) Normalement, nous ne devrions pouvoir faire qu'une seule écriture à la fois. Comment le garantir avec un sémaphore ? [1 point] Est-ce le travail du programmeur d'applications que de faire cette vérification ? [1 point]

```
mutex := semaphore( 1 );
...
  |     |  P(mutex)
  |     |  écrire sur un fichier pendant 3s
  |     |  V(mutex)
 ...

Non, ce n'est pas le travail du programmeur car d'autres
processus peuvent demander des E/S vers le disque. La synchro
est donc à la charge du système.
```

Système d'exploitation

Responsable : Jean-Luc Massat

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
8 janvier 2021, première session
durée 1 heure, calculatrices et documents autorisés

1 Gestion de fichiers (6 points, 15 mn)

On vous donne une instance d'une FAT (-1 signale le dernier bloc d'un fichier, -2 un bloc réservé et -3 un bloc libre) :

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
FAT[i]	-2	-2	-2	-1	10	-3	-1	-3	13	11	-1	7	4	-1	8	5

A) Dans cette configuration combien avons-nous de fichiers et quels sont les blocs physiques de chaque fichier ? [3 points].

```
Fichier 1: 9, 13, 4
Fichier 2: 12, 7
Fichier 3: 16, 5, 10, 11
Fichier 4: 14
```

B) Le codage de cette FAT est-il erroné (indiquez clairement les erreurs et comment pouvons-nous les trouver) ? [1 point].

Le bloc 15 pointe vers le 8 qui est libre. C'est une erreur.

C) Que réalisent les opérations ci-dessous ? [1 point].

```
fat[6] = -1;
fat[14] = 6;
```

Ajout du bloc 6 à un fichier qui se terminait avant en 14.

D) Que contiennent les blocs réservés au début du disque ? [1 point]

Des descripteurs et la FAT.

2 Gestion des disques (4 points, 10 mn)

Considérons une machine dotée de 6 disques d'un téra-octet et des circuits d'E/S permettant à ces disques de réaliser des opérations en parallèle.

A) Quelle est la capacité de stockage pour une configuration RAID5 construite sur les six disques (que nous noterons par la suite RAID5(D1,D2,D3,D4,D5,D6)) ? [0,5 point]

capacité $= (5 \times \text{To})$

B) Même question pour un *striping* (RAID0) des disques : RAID0(D1,D2,D3,D4,D5,D6) ? [0,5 point]

capacité $= (6 \times \text{To})$

C) Quelle est la capacité de stockage et le nombre maximum de disques défaillants (sans altérer le bon fonctionnement du système) pour les trois configurations suivantes. Indiquez par un exemple les disques susceptibles d'être défaillants. [3 points]

C1 RAID0(RAID5(D1,D2,D3), RAID5(D4,D5,D6))

C2 RAID5(RAID0(D1,D2), RAID0(D3,D4), RAID0(D5,D6))

C3 RAID5(RAID1(D1,D2), RAID1(D3,D4), RAID1(D5,D6))

```
RAID1 = mirroring, capacité de RAID1(A,B) = 1 To

     | Capacité  | Fautes       | Exemple
-------------------------------------------------
C1 : | 4 To,     | 2 disques,   | D1 et D4
C2 : | 4 To,     | 2 disques,   | D5 et D6
C3 : | 2 To,     | 4 disques,   | D1, D3, D5 et D6
```

3 Gestion des processus et synchronisation (10 points, 35 mn)

Considérons un système équipé d'un **seul disque** et dans lequel les **deux processeurs** sont gérés en temps partagé par l'algorithme du **tourniquet**.

A) Donnez le temps de réponse du processus ci-dessous et le taux d'utilisation des processeurs sur la période considérée. Indiquez clairement le déroulement des opérations sur les processeurs et le disque. [2 points]

```
faire quatre fois
  |  calculer pendant une seconde
  |  écrire sur un fichier pendant 2 secondes
fin-faire
calculer pendant une seconde
```

TR=13, TX=5/26

```
---- :   1  2  3  4  5  6  7  8  9 10 11 12 13
CPU1 : |CC|--|--|CC|--|--|CC|--|--|CC|--|--|CC|
CPU2 : |--|--|--|--|--|--|--|--|--|--|--|--|--|
DISK : |--|DD|DD|--|DD|DD|--|DD|DD|--|DD|DD|--|
```

B) Même question avec les mêmes détails si nous utilisons des threads (version ci-dessous). [3 points]

```
faire quatre fois
  |  créer un thread pour faire
  |    |  calculer pendant une seconde
  |    |  écrire sur un fichier pendant 2 secondes
  |  fin du thread
fin-faire
attendre la fin des threads fils
calculer pendant une seconde
```

TR=11, TX=5/22

```
---- :   1  2  3  4  5  6  7  8  9 10 11
CPU1 : |CC|CC|--|--|--|--|--|--|--|--|CC|
CPU2 : |CC|CC|--|--|--|--|--|--|--|--|--|
DISK : |--|--|DD|DD|DD|DD|DD|DD|DD|DD|--|
```

C) Même question avec les mêmes détails si nous synchronisons les threads (version ci-dessous). Expliquez notamment la valeur initiale du compteur du sémaphore [3 points]

```
s := nouveau_sémaphore( 2 );
faire quatre fois
  |  créer un thread pour faire
  |    |  P(s)
  |    |  calculer pendant une seconde
  |    |  V(s)
  |    |  écrire sur un fichier pendant 2 secondes
  |  fin du thread
fin-faire
attendre la fin des threads fils
calculer pendant une seconde
```

TR=10, TX=5/20

```
---- :   1  2  3  4  5  6  7  8  9 10
CPU1 : |CC|CC|--|--|--|--|--|--|--|CC|
CPU2 : |CC|CC|--|--|--|--|--|--|--|--|
DISK : |--|DD|DD|DD|DD|DD|DD|DD|--|
```

D) Comment reprendre le code de la question **B** et remplacer l'instruction d'attente par une version basée sur un sémaphore (à créer, initialiser et utiliser) [2 points].

```
attente := semaphore( 0 );
faire quatre fois
  |  créer un thread pour faire
  |    |  calculer pendant une seconde
  |    |  écrire sur un fichier pendant 2s
  |    |  V(attente)
  |  fin du thread
fin-faire
faire quatre fois P(attente); fin-faire
calculer pendant une seconde
```

Système d'exploitation

Responsable : Jean-Luc Massat

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
7 janvier 2020, première session
durée 2 heures, calculatrices et documents autorisés

1 Choix d'une page victime (6 points, 35 mn)

Considérons un système de mémoire paginée. Nous disposons de 4 pages physiques qui sont toutes occupéees, le tableau donnant ci-dessous, pour chacune d'elles, la date du chargement de la page, la date du dernier accès à cette page et l'état des indicateurs de la page physique (accédée et modifiée).

Page physique	Chargement	Accès	Accédée	Modifiée
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

A) En justifiant votre réponse, indiquez quelle sera la page remplacée (la victime) pour chacun des 3 algorithmes de remplacement suivants : LRU, FIFO et seconde chance (FINUFO). [3 points]

```
LRU    : page 1 (dernier accès le plus vieux 260)
FIFO   : page 2 (chargement le plus vieux 120)
FINUFO : page 0 (bit d'accès à zéro)
```

B) Quelle information vous manque-t-il pour pouvoir appliquer l'algorithme optimal ? [1 point]

```
La date du prochain accès dans le futur.
```

C) L'algorithme NRU (*Not Recently Used*) consiste à choisir la page victime en utilisant les deux bits d'accès et de modification. Comment, d'après-vous, l'algorithme classe-t-il les quatre configurations possibles (justifiez votre choix) ? [2 points]

```
A M  | Priorité pour choisir la victime
-----+--------------
1 1  | utilisée et modifiée -> à garder (priorité 0)
1 0  | utilisée mais non modifiée -> à garder si possible (priorité 2)
0 1  | non utilisée mais modifiée -> à garder si possible (priorité 1)
0 0  | non utilisée et non modifiée -> à supprimer (priorité 3)
```

2 Système de gestion de fichiers (5 points, 30 mn)

Considérez un système de gestion de fichiers où le disque est géré à l'aide d'une FAT (File Allocation Table). La FAT est un tableau d'entier qui possède autant de lignes que des blocs sur le disque. Un bloc i est libre si `fat[i] = FAT_FREE`. Un bloc défectueux est marqué `FAT_BAD`. Si le bloc i est le dernier bloc d'un fichier, on a `fat[i] = FAT_EOF`. Toute autre valeur positive sur `fat[i]` indique le bloc suivant.

A) Écrivez la fonction d'allocation qui recherche un espace libre de `n` blocs consécutifs et renvoie l'adresse du premier bloc (ou -1 en cas d'échec). Cet espace libre peut contenir des blocs défectueux (sauf le premier). La FAT doit être parcourue une seule fois. [3 points]

```
int alloc_blocks(int fat[], int taille_fat, int n)
```

```
alloc_blocks(int fat[], int taille_fat, int n)
   resultat = 0
   taille_libre = 0
   pour i variant de 0 a taille_fat-1
      si (fat[i] == FAT_FREE) alors
         taille_libre++;
         si (taille_libre == n) renvoyer resultat;
      sinon-si (fat[i] == FAT_BAD) alors
         si (taille_libre == 0) resultat = (i + 1);
      sinon
         resultat = i+1;
         taille_libre = 0;
      finsi
   fin-pour
   renvoyer -1
```

B) Écrivez la fonction de chaînage de la FAT lorsque les blocs sont alloués par la fonction d'allocation `alloc_blocks`. Le paramètre `debut` représente le premier bloc alloué et le paramètre `n` le nombre de blocs alloués. **Rappel** : le dernier bloc d'un fichier a la valeur `FAT_EOF`. [2 points]

```
void chainages(int fat[], int debut, int n)
```

```
void chainages(int fat[], int debut, int n)
   precedent = debut
   n--
   tant que (n > 0)
      debut++
      si (fat[debut] == FAT_FREE) alors
         fat[precedent] = debut
         precedent = debut
         n--
      finsi
   fin-tant que
   fat[precedent] = FAT_EOF
```

Tournez la page SVP...

3 Gestion des processus et synchronisation (9 points, 55 mn)

Considérons un système équipé d'un **seul disque** et dans lequel le processeur **unique** est géré en temps partagé par l'algorithme du **tourniquet**.

A) Donnez le temps de réponse du processus ci-dessous et le taux d'utilisation du processeur sur la période considérée. [1 point]

```
int main (void) {
    DATA a, b;
    a = produireA();  /* 4 secondes de CPU  */
    ecrire(a);        /* E/S de 6 secondes  */
    b = produireB();  /* 2 secondes de CPU  */
    ecrire(b);        /* E/S de 4 secondes  */
    finir();          /* 2 secondes de CPU  */
    return 0;
}
```

```
Exécution : ++++------++----++  (+ pour CPU et - pour E/S)
Temps de réponse = 18s
Taux d'utilisation de la CPU = 8S / 18s = 44%
```

B) Nous disposons maintenant d'une version asynchrone de la fonction d'écriture (voir ci-dessous). Proposez une version de `main` qui utilise cette fonction pour réduire le temps de réponse. Quel est le nouveau taux d'utilisation du processeur ? **Remarque** : utilisez le principe de l'attente active et limitez la consommation de CPU. [2 points]

```
void ecrire_asynchrone(DATA d, int* finie);
```

```
int main (void) {
    int finie = 0;
    DATA a = produireA();          /* 4 secondes de CPU            */
    ecrire_asynchrone(a, &finie);  /* E/S asynchrone de 6 secondes */
    DATA b = produireB();          /* 2 secondes de CPU            */
    while (finie == 0) {           /* attendre la fin de l'E/S     */
        usleep(1000);              /* ne pas consommer trop de CPU */
    }
    ecrire_asynchrone(b, &finie);  /* E/S asynchrone de 4 secondes */
    finir();                       /* 2 secondes de CPU            */
    while (finie == 0) {           /* attendre la fin de l'E/S     */
        usleep(1000);              /* ne pas consommer trop de CPU */
    }
    return 0;
}

trace CPU : ++++++....++..  (. pour l'attente)
trace E/S : ....----------  (. pas d'E/S à faire)
Temps de réponse = 14s
Taux d'utilisation de la CPU = 8S / 14s = 57%
```

C) Le système d'exploitation nous offre deux fonctions pour gérer les threads (voir ci-dessous). Comment écrire la fonction `ecrire_asynchrone` en utilisant la fonction `ecrire` et la création de thread ? [2 points]

```
int  new_thread();   /* 0 chez le fils et > 0 chez le père */
void exit_thread();  /* fin du thread courant              */
```

```
void ecrire_asynchrone(DATA d, int* finie) {
    *finie = 0;
    if (new_thread() == 0) {
        ecrire(d);             /* le thread fils réalise l'E/S */
        *finie = 1;
        exit_thread();
    }
}
```

D) Notre système dispose également de trois fonctions afin de manipuler des sémaphores (voir ci-dessous). Proposez une nouvelle version de l'écriture asynchrone basée sur un sémaphore : `SEMAPHORE ecrire_asynchrone_sem(DATA d)` . [2 points]

```
SEMAPHORE sem_create(int counter);
void sem_P(SEMAPHORE sem);
void sem_V(SEMAPHORE sem);
```

```
SEMAPHORE ecrire_asynchrone(DATA d) {
    SEMAPHORE s = sem_create(0);  /* Sémaphore de droit de passage */
    if (new_thread() == 0) {
        ecrire(d);                /* le thread fils réalise l'E/S  */
        sem_V(s);                 /* donner le droit de passage    */
        exit_thread();
    }
    return s;
}
```

E) Proposez une nouvelle version de `main` basée sur `ecrire_asynchrone_sem`. [1 point]

```
int main (void) {
    SEMAPHORE s;
    DATA a = produireA();           /* 4 secondes de CPU            */
    s = ecrire_asynchrone_sem(a);   /* E/S asynchrone de 6 secondes */
    DATA b = produireB();           /* 2 secondes de CPU            */
    sem_P(s);                       /* attendre la fin de l'E/S     */
    s = ecrire_asynchrone(b);       /* E/S asynchrone de 4 secondes */
    finir();                        /* 2 secondes de CPU            */
    sem_P(s);                       /* attendre la fin de l'E/S     */
    return 0;
}
```

F) Quel est le temps de réponse et le taux d'utilisation du processeur si les deux écritures (de A et de B) sont réalisées sur deux disques indépendants. [1 point]

```
trace CPU    : ++++++++..  (. pour l'attente)
trace disk 1 : ....------  (. pas d'E/S à faire)
trace disk 2 : ......----  (. pas d'E/S à faire)
Temps de réponse = 10s
Taux d'utilisation de la CPU = 8S / 10s = 80%
```

Les sujets plus anciens sont basés sur une version du cours assez différente (6 crédits, 60 heures et des chapitres en plus). De ce fait, les corrections ne sont pas disponibles.

Système d'exploitation

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
11 janvier 2019, première session
durée 2 heures, calculatrices et documents autorisés

Responsable : Jean-Luc Massat

1 Gestion des processus et synchronisation *(8 points, 50 minutes)*

Considérons un système équipé d'un **seul disque** et dans lequel le processeur **unique** est géré en temps partagé par l'algorithme du **tourniquet**.

A) Donnez le temps de réponse du processus ci-dessous et le taux d'utilisation du processeur sur la période considérée. [1 point]

```
int main (void) {
    calculer();  /* 3 secondes          */
    ecrire();    /* E/S de 4 secondes   */
    finir();     /* calcul de 2 secondes */
    return 0;
}
```

B) Nous disposons maintenant d'une version asynchrone de la fonction d'écriture (voir ci-dessous). Proposez une version de `main` qui utilise cette fonction pour réduire le temps de réponse. Quel est le nouveau taux d'utilisation du processeur ? **Remarque** : utilisez le principe de l'attente active et limitez la consommation de CPU. [2 points]

```
void ecrire_asynchrone(int* finie);
```

C) Le système d'exploitation nous offre deux fonctions pour gérer les threads (voir ci-dessous). Comment écrire la fonction `ecrire_asynchrone` en utilisant la fonction `ecrire` et la création de thread ? [1,5 point]

```
int  new_thread();   /* 0 chez le fils et > 0 chez le père */
void exit_thread();  /* fin du thread courant              */
```

D) Notre système dispose également de trois fonctions afin de manipuler des sémaphores :

```
int  sem_create(int counter);
void sem_P(int sem_id);
void sem_V(int sem_id);
```

Proposez une nouvelle version de l'écriture asynchrone `int ecrire_asynchrone_sem(void)` basée sur un sémaphore (la fonction renvoie l'identifiant du sémaphore). [2 points]

E) Proposez une nouvelle version de `main` basée sur `ecrire_asynchrone_sem`. Quel est le temps de réponse et le taux d'utilisation du processeur de cette dernière version si l'écriture dure 6 secondes ? [1,5 point]

Tournez la page SVP...

2 Les régions mémoire de Linux *(7 points, 35 minutes)*

la carte mémoire des régions d'un processus Linux basée sur une mémoire paginée est détaillée ci-dessous (la taille est exprimée en pages) :

n°	adresse	taille	droits
0	0x10000000	256	r-x
1	0x12000000	512	rw-
2	0xEF800000	2048	rw-

Questions :

A) Donnez un sens à ces régions (plusieurs solutions sont possibles). [2 points]

B) La colonne **adresse** contient-elle des adresses logiques ou des adresses physiques ? [1 point]

C) Donnez un adresse qui ne correspond à aucun région. [1 point]

D) Quelle est la taille de la mémoire logique de ce processus ? [1 point]

E) Si nous décidons d'interdire l'écriture sur la quatrième page (de numéro 3) de la région 1, quel est le nouveau contenu de la table des régions ? **Remarque** : notez qu'une page mesure 2^{12} octets, soit 4096 octets en décimal et 0x1000 octets en hexadécimal. [2 points]

3 Gestion d'une mémoire virtuelle *(5 points, 35 minutes)*

Considérons une mémoire composée de trois pages physiques initialement vides. Lors de son exécution, un processus unique accède dans l'ordre aux pages virtuelles suivantes :

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.

Voila une trace de l'évolution de la mémoire sur les trois premiers accès :

	init	7	0	1	...
page physique 0	-	7	7	7	...
page physique 1	-	-	0	0	...
page physique 2	-	-	-	1	...
défaut de page		oui	oui	oui	...

Questions :

A) Devons-nous considérer que ce processus a effectué seulement 22 accès à la mémoire ? [1 point]

B) Pour chacun des algorithmes FIFO et LRU, donnez le contenu des pages physiques (c-à-d un numéro de page virtuelle) après chaque accès. Précisez également le nombre total de défauts de page provoqués par ces accès. [3 points]

C) Combien faut-il de pages physiques pour minimiser le nombre de défauts de page ? [1 point]

Système d'exploitation

Responsable : Jean-Luc Massat

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
16 mai 2018, première session
durée 2 heures, calculatrices et documents autorisés

1 Questions... réponses (8 points)

Les questions ci-dessous doivent être traitées dans le cadre d'un système d'exploitation moderne, multi-programmé et doté d'une gestion mémoire performante (n'oubliez pas de justifier vos réponses). **Questions :** (un point par question sauf exceptions)

a) Sur une zone de 1024 ko gérée par l'algorithme du *buddy system*, quel est l'état des blocs libres et occupés après les allocations de 65 ko, 130 ko, 220 ko, 50 ko et 91 ko dans cet ordre ?

b) Si le programme $\ll P(s); P(t); \ldots; V(s); \ldots; P(t); \ldots; V(t); V(t); \ldots; P(s); V(s); \gg$ est exécuté en plusieurs exemplaires, existe-t-il un risque d'interblocage (les compteurs sont initialisés à deux) ?

c) Soient les cinq processus suivants qui démarrent en même temps. Combien faut-il de processeurs pour les exécuter sans contrainte (nous considérerons que les E/S se déroulent en parrallèle) ? Quel est le taux d'utilisation du processeur ? [2 points]

- calculer 3s, E/S sur 4s, calculer 3s
- calculer 2s, E/S sur 3s, calculer 1s, E/S de 3s, calculer 1s
- calculer 2s, E/S de 2s, calculer 2s, E/S de 2s, calculer 2s
- dormir sur 5s, calculer 2s, E/S de 3s
- E/S de 3s, calculer 3s, E/S de 4s

d) Donnez la matrice `max` de l'algorithme des banquiers pour ces trois processus :

- P_1 : allouer R_1 et R_2, ..., libérer R_1, ..., allouer R_3, R_2 et R_1, ..., libérer toutes les resources
- P_2 : allouer R_3 et R_1, ..., libérer R_3, ..., allouer R_2 et R_1, libérer toutes les ressources
- P_2 : allouer R_2 et R_3, ..., allouer R_2, ..., allouer R_3, ..., libérer toutes les ressources

e) Si le système reçoit les requêtes ci-dessous, donnez l'ordre de traitement en se basant sur l'algorithme SSTF (*Shortest Seek Time First*). La tête de lecture se trouve sur la piste 500.

200, 150, 320, 700, 210, 100, 600, 705, 490, 150, 310, 610, 150

f) Dans la FAT ci-dessous combien avons-nous de fichiers et quels sont les blocs physiques de chaque fichier ? [2 points]

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
FAT[i]	22	22	13	19	52	-1	20	17	52	15	6	3	-1	-1	4	52	18	-1	-1	8

2 Gestion des processus et synchronisation (8 points)

Considérons un système équipé d'un **seul disque** et dans lequel le processeur **unique** est géré en temps partagé par l'algorithme du **tourniquet**.

a) Donnez le temps de réponse du processus ci-dessous et le taux d'utilisation du processeur sur la période considérée. [2 points]

```
pour i ∈ {1, 2} faire
  | calculer pendant i seconde(s)
  | écrire sur un fichier pendant 4 secondes
fin-pour
calculer pendant 1 seconde
```

b) Même question que précédemment mais nous modifions le processus comme suit (indiquez clairement sur un schéma le déroulement de l'exécution). [2 points]

```
pour i ∈ {1, 2} faire
  |   créer un thread pour faire
  |     | calculer pendant i seconde(s)
  |     | écrire sur un fichier pendant 4 secondes
  |   fin du thread
fin-pour
attendre la fin des threads fils
calculer pendant 1 seconde
```

c) Si nous considérons que les deux E/S se déroulent sur deux disques différents, quel est l'impact sur le temps de réponse et le taux d'utilisation du processeur ? [2 points]

d) En gardant l'organisation en threads, comment modifier le code (avec l'utilisation d'un sémaphore) pour s'assurer que le calcul de 1 seconde s'exécute en premier et occupe toute la CPU ? [2 points]

3 Gestion des disques (4 points)

Considérons une machine dotée de 6 disques d'un téra-octet et des circuits d'E/S permettant à ces disques de réaliser des opérations en parallèle.

a) Quelle est la capacité de stockage pour une configuration RAID5 construite sur les six disques (que nous noterons par la suite RAID5(D1,D2,D3,D4,D5,D6)) ? [1 point]

b) Même question pour un *striping* (RAID0) des diques : RAID0(D1,D2,D3,D4,D5,D6) ? [1 point]

c) Quelle est la capacité de stockage et le nombre maximum de disques défaillants (sans altérer le bon fonctionnement du système) pour la configuration RAID0(RAID5(D1,D2,D3), RAID5(D4,D5,D6)) Indiquez par un exemple les disques susceptibles d'être défaillants. [2 points]

Système d'exploitation

Responsable : Jean-Luc Massat

Troisième année de la licence d'informatique
UFR Sciences, site de Luminy, Aix-Marseille Université
16 mai 2017, première session,
durée 2 heures, calculatrices et documents autorisés.

1 Exécution d'un processus — 9 points

Considérons le programme ci-dessous :

```
// Exécution : faire a fois ( 1 minute de CPU puis 2 minutes d'E/S )
int fa(int a) { ... }

// Exécution : faire b fois ( 1 minute de CPU puis 3 minutes d'E/S )
int fb(int b) { ... }

// Exécution : 1 minute de CPU
int fc(int s) { ... }

int main() { int a = fa(2); int b = fb(3); return fc(a+b); }
```

Questions (donnez à chaque fois la trace de l'exécution) :

A) Si nous avons un seul processeur, quel est le temps de réponse de ce programme et le taux d'utilisation de la CPU ? [1 point]

B) Même question si nous disposons de deux processeurs. [1 point]

C) Revenons au processeur unique et considérons que les E/S sont des écritures. Nous disposons d'un tampon de sortie de 4 minutes. L'opération de vidage de ce tampon est assurée de manière asynchrone par le système d'exploitation. La fin du processus implique l'attente du **vidage du tampon**. Dans ces conditions quel est le temps de réponse et le taux d'utilisation du processeur ? [2 points]

D) Nous avons maintenant **deux disques** (et donc deux tampons de 4 minutes). La fonction `fa()` utilise un disque et `fb()` l'autre. Calculez le temps de réponse et le taux d'utilisation du processeur. [2 points]

E) En plus des deux disques (et donc des deux tampons), nous disposons maintenant de **trois processeurs** et nous ajoutons à notre système des fonctions de gestion des threads :

```
int  th_create();    // 0: fils; numéro:père
void th_exit();      // fin du thread courant
void th_join(int n); // attente de la fin d'un thread
```

Modifiez la fonction `main()` pour exécuter les deux fonctions dans deux threads et calculez le temps de réponse et le taux d'utilisation du processeur. [3 points]

Tournez la page SVP...

2 Les philosophes et les banquiers — 5 points

Un beau jour de mai, 5 philosophes décident d'aller manger ensemble des spaghettis. Le restaurateur dresse une table ronde, mais place seulement 5 fourchettes (une entre chaque assiette). Pour manger, un philosophe a besoin de deux fourchettes : celle placée à sa droite et celle placée à sa gauche.

Questions :

A) Montrez qu'il existe un risque d'interblocage et proposez une solution simple à ce problème. [1 point]

B) Pour éviter les interblocages, les philosophes décident d'appliquer l'algorithme des banquiers. Donnez la matrice `Max` et le vecteur `Dispo`. [1 point]

C) Pour **ce problème bien particulier** donnez l'algorithme simplifié permettant de déterminer si oui ou non, le système est dans un état sain. [2 points]

D) Le restaurateur place maintenant les cinq fourchettes au **centre de la table** (chaque philosophe a toujours besoin de deux fourchettes). Donnez la nouvelle version de la matrice `Max` et du vecteur `Dispo`. [1 point]

3 Système de gestion de fichiers — 6 points

Considérez un système de gestion de fichiers où le disque est géré à l'aide d'une FAT (File Allocation Table). La FAT est un tableau d'entier qui possède autant de lignes que des blocs sur le disque. Un bloc i est libre si `fat[i] = FAT_FREE`. Si le bloc i est le dernier bloc d'un fichier, on a `fat[i] = FAT_EOF`. Toute autre valeur positive sur `fat[i]` indique le bloc suivant.

A) Écrivez la fonction d'allocation qui recherche un espace libre de `n` blocs consécutifs et renvoie l'adresse du premier bloc (ou -1 en cas d'échec) en se basant sur le principe de meilleur ajustement (best-fit). La FAT doit être parcourue une seule fois. [2 points]

```
int alloc_bloc_best_fit (int fat[], int taille_fat, int n)
```

B) Écrivez la fonction de chaînage de la FAT lorsque les blocs sont alloués par la fonction d'allocation `alloc_bloc_best_fit`. Le paramètre `debut` représente le premier bloc alloué et le paramètre `n` le nombre de blocs alloués. **Rappel** : le dernier bloc d'un fichier a la valeur `FAT_EOF`. [2 points]

```
void chainages(int fat[], int debut, int n)
```

C) Écrivez la fonction d'allocation `alloc_bloc_frac` qui alloue `n` blocs non-consécutifs et retourne l'adresse du premier (ou -1 s'il n'existe pas assez de blocs libres). **Rappel** : n'oubliez pas d'enchaîner correctement les blocs sur la `fat`. [2 points]

```
int alloc_bloc_frac (int fat[], int taille_fat, int n)
```