IF-3-SYS : Systèmes d'exploitation 10 juin 2024

NOM Prénom :

Consignes

- L'examen dure 1h30. Prenez le temps de lire le sujet en entier (17 questions sur 9 pages)
- Les réponses seront à inscrire sur le sujet. Commencez par écrire votre nom ci-dessus.
- Écrivez lisiblement et surtout sans ratures. Utilisez un brouillon (vraiment).
- Documents et appareils interdits, sauf une feuille A4 recto-verso manuscrite.
- Pour le binaire et l'hexadécimal, aidez-vous des tableaux page 8.

1 Questions de cours

Sauf mention contraire, chaque question est indépendante. Dans les questions vrai/faux, les erreurs sont décomptées : ne répondez pas au hasard.

Noyau, processus, appels système

Question 1 On s'intéresse à un processus parent qui vient de créer un ou plusieurs enfants, et on suppose que, pour l'instant, ni parent ni enfant(s) n'ont invoqué d'autre appel système. Pour chaque proposition ci-dessous, entourez V si elle est correcte, ou entourez F si elle est fausse ou absurde.

V F Le parent exécute le même programme que ses enfants.
V F Le parent partage son espace d'adressage avec ses enfants.
V F Le parent connaît les numéros (pid) de ses enfants.
V F Si le parent meurt, ses enfants restent en vie.

Question 2 Sous Linux, comment s'appelle l'utilitaire qui permet d'afficher les appels système invoqués par un programme au fur et à mesure de son exécution ?

Question 3 Écrivez ci-dessous un programme qui crée deux processus enfants : l'un affiche les entiers de 1 à 50 (un nombre par ligne), l'autre de 51 à 100 (un par ligne aussi). Le parent attend la fin de ces affichages, puis il affiche *"au revoir"* avant de rendre la main au shell.
Vous trouverez en page 9 un aide-mémoire des différents appels système disponibles.

```
int main(void)
{
```

```
    printf("au revoir\n");
    return 0;
}
```

Ordonnancement

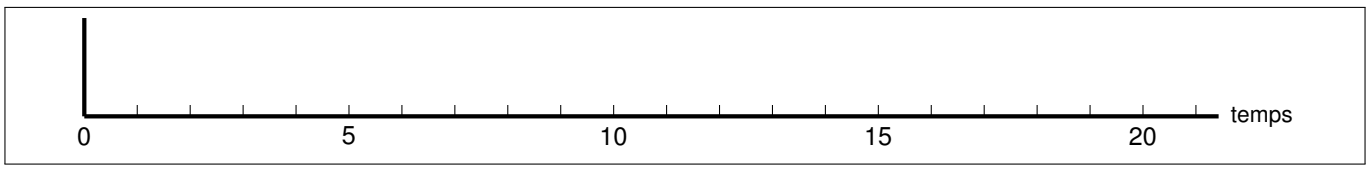
Question 4 Dans le cadre ci-dessous, dessinez le diagramme états/transitions d'un PCB dans l'ordonnanceur, en supposant une stratégie non préemptive.

Question 5 On suppose dans cette question un ordonnanceur avec trois niveaux de priorité statiques (i.e. la priorité des tâches est fixée lors de leur création) haute, moyenne, et basse. Entre des niveaux distincts, l'ordonnancement est strictement préemptif. Parmi les tâches d'un même niveau, l'ordonnancement est circulaire (Round Robin) avec un quantum de 2 unités de temps.

On s'intéresse aux tâches ci-dessous :

processus	A	B	C	D	E	F	G
durée d'exécution	7	4	4	1	2	2	1
instant d'arrivée	0	0	1	1	1	2	2
priorité	moy.	haute	basse	moy.	haute	basse	moy.

Au brouillon, dessinez un chronogramme indiquant la succession des tâches sur le processeur. Recopiez ensuite votre réponse au propre dans le cadre ci-dessous.



Mémoire virtuelle

Question 6 Pour chaque proposition ci-dessous, entourez V si elle est correcte, ou entourez F si elle est fausse ou absurde. Si une même adresse virtuelle est utilisée par deux processus distincts, alors :

V F c'est une erreur car les adresses doivent être uniques.
V F elle sera traduite selon la même table de pages.
V F elle ne sera pas forcément traduite en la même adresse physique.
V F il est possible qu'elle soit traduite en la même adresse physique.

Question 7 On s'intéresse dans cette question à un système 32 bits (c.à.d. que les adresses virtuelles et physiques sont codées sur 32 bits) avec mémoire virtuelle paginée. La machine dispose de 2Gio de mémoire vive, et la taille des pages est de 1Mio (cf annexe page 8). Quelle est la taille maximale, en nombre de pages, de l'espace d'adressage d'un processus ?
Vous pouvez exprimer votre réponse en décimal, en puissances de 2, ou en utilisant les préfixes binaires Ki, Mi etc.

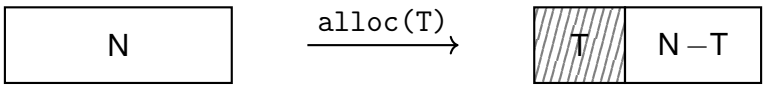
Question 8 On s'intéresse dans cette question à un processus qui vient de rencontrer un défaut de page. Dans quel état (au sens de la question 4) est ce processus vis-à-vis de l'ordonnanceur ?

Allocation dynamique

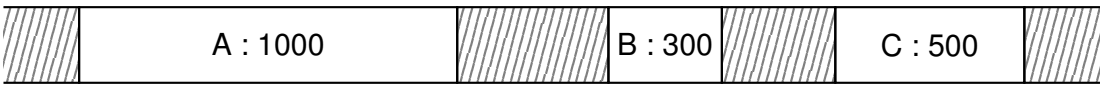
Question 9 On souhaite allouer dynamiquement un tableau de N entiers, tous initialisés à zéro. Complétez le code ci-dessous. Vous n'avez pas le droit d'utiliser l'opérateur d'indexation « `[]` ».

```
int* new_array(int N)
{
    int *tab = malloc(









   return tab;
}
```

Question 10 On s'intéresse dans cette question aux stratégies d'allocation dynamique best-fit et first-fit. Pour simplifier l'exercice, on suppose ici qu'un bloc de taille N peut servir à allouer une zone de taille $T \leqslant N$, laissant le cas échéant un bloc libre de taille $N - T$. Autrement dit on néglige l'alignement, les en-têtes, etc :

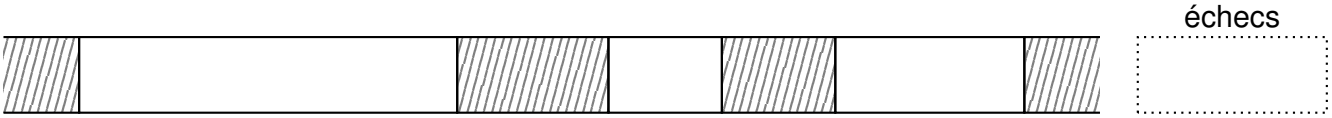


L'état initial du tas est illustré ci-dessous : la *freelist* contient uniquement trois blocs libres A (1000 octets), B (300 octets) et C (500 octets), chaînés dans cet ordre.



L'application demande successivement quatre allocations X, Y, Z, et T de respectivement 100, 200, 400 et 800 octets.

Dessinez ci-dessous l'état du tas après traitement de ces quatre requêtes selon la stratégie best-fit. Si une ou des allocations échoue(nt), indiquez les lettres correspondantes dans le rectangle à droite.



En repartant du même état initial, dessinez maintenant l'état final du tas en supposant plutôt une stratégie first-fit. Si une ou des allocations échoue(nt), indiquez les lettres correspondantes dans le rectangle à droite.

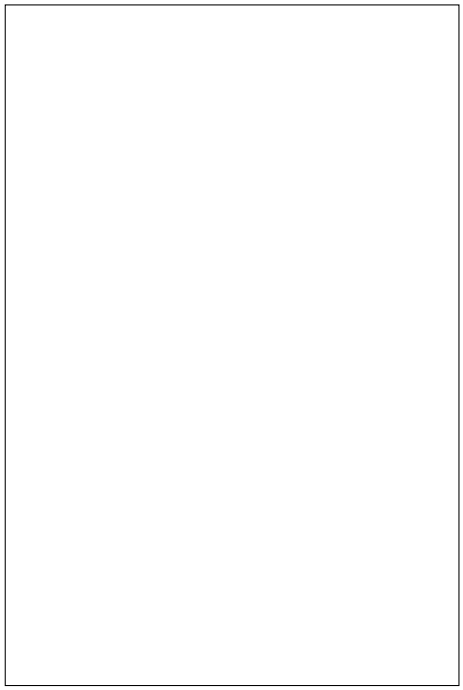
Concurrence et synchronisation

Question 11 Quel est le nombre minimal de threads que peut contenir un processus ? Si besoin, vous pouvez utiliser dans votre réponse les constantes suivantes :

- P : le nombre actuel de processus sur la machine
- T : le nombre actuel de threads sur la machine
- C : le nombre de processeurs physiques sur la machine

Stockage et systèmes de fichiers

Question 12 Donnez une liste de lignes de commandes shell permettant de créer l'arborescence illustrée ci-dessous. À l'instant initial, le dossier courant est «`racine`», supposé vide. Dans votre réponse, aucune ligne de commandes ne peut comporter un slash « `/` ».



```
racine
├── dossier1
│   ├── fichierA
│   └── dossier2
│       └── fichierB
└── dossier3
    └── dossier4
        └── fichierC
```

Question 13 On s'intéresse à une carte mémoire de 16 Gio, formattée en un volume de type FAT avec des blocs de 4 kio (cf annexe page 8). Les numéros de blocs sont codés sur 32 bits. Quelle est la taille, en octets, occupée par la table d'allocation des fichiers ?
Vous pouvez exprimer votre réponse en décimal, en puissances de 2, ou en utilisant les préfixes binaires Kio, Mio etc.

2 Problème : la barrière de rendez-vous synchrone

Dans cette seconde partie du sujet, on va étudier plusieurs variantes d'une primitive déjà abordée en TD, la barrière de synchronisation. Rappel : lorsqu'un thread atteint une barrière, il doit attendre que tous les autres threads du programme aient également atteint leur barrière. Une fois que tout le monde est arrivé au point de rendez-vous, les threads peuvent «franchir la barrière» et poursuivre leur exécution.

Question 14 On s'intéresse dans un premier temps à un programme avec seulement deux threads :

```
threadA()
{
   C1();
   barrièreA();
   C2();
}
```

```
threadB()
{
   C3();
   barrièreB();
   C4();
}
```

Les fonctions C1..C4 sont supposées purement calculatoires, c'est à dire qu'elles ne comportent que des instructions ordinaires. Votre travail consiste à implémenter les fonctions `barrièreA` et `barrièreB` de façon à garantir que C1 et C3 seront terminées avant que C2 et C4 ne soient exécutées.
Vous pouvez utiliser un ou plusieurs sémaphores, et/ou des instructions et variables ordinaires. Dans tous les cas, n'oubliez pas d'en indiquer les valeurs initiales.
Pour la syntaxe, vous pouvez utiliser au choix la notation P/V vue en TD, ou des appels systèmes Unix comme en TP (cf page 9).

Variables partagées

barrièreA()

barrièreB()

Question 15 On cherche maintenant une solution générique pour un programme à N threads. Chaque thread est identifié par son numéro `i` (avec $1 \leqslant$ `i` $\leqslant$ N) et exécute le code suivant :

```
thread(int i)
{
   C(2*i-1);
   barrière(i);
   C(2*i);
}
```

Votre travail consiste à implémenter la fonction `barrière(int i)`. N'oubliez pas de déclarer explicitement tous vos sémaphores et variables partagés ainsi que leur valeur initiale.

Variables partagées

`barrière(int i)`

Question 16 La barrière que vous venez d'implémenter est-elle *réutilisable* ? Autrement dit, on se demande si elle fonctionnerait correctement pour des threads de la forme ci-dessous :

```
thread(int i)
{
   while(true)
   {
      C(i);
      barrière(i);
   }
}
```

Si vous pensez que votre barrière est réutilisable, donnez un argument pour justifier votre affirmation. Au contraire, si vous répondez qu'elle ne l'est pas, donnez un exemple de scénario causant un problème.

Question 17 Proposez une nouvelle version de `barrière(i)` satisfaisant ces deux propriétés :

1. Le nombre de sémaphores est indépendant du nombre de threads.
2. La barrière est réutilisable, au sens de la question précédente.

Variables partagées

`barrière(int i)`

Annexe : aide pour les calculs en binaire

Les premiers nombres entiers, notés en décimal, en hexadécimal, et en binaire :

Dec	Hex	Bin
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100

Dec	Hex	Bin
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001

Dec	Hex	Bin
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110

Dec	Hex	Bin
15	F	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011

Les premières puissances de 2 :

$2^0 = 1$	$2^{16} = 65\,536$	$2^{32} = 4\,294\,967\,296$	$2^{48} = 281\,474\,976\,710\,656$
$2^1 = 2$	$2^{17} = 131\,072$	$2^{33} = 8\,589\,934\,592$	$2^{49} = 562\,949\,953\,421\,312$
$2^2 = 4$	$2^{18} = 262\,144$	$2^{34} = 17\,179\,869\,184$	$2^{50} = 1\,125\,899\,906\,842\,624$
$2^3 = 8$	$2^{19} = 524\,288$	$2^{35} = 34\,359\,738\,368$	$2^{51} = 2\,251\,799\,813\,685\,248$
$2^4 = 16$	$2^{20} = 1\,048\,576$	$2^{36} = 68\,719\,476\,736$	$2^{52} = 4\,503\,599\,627\,370\,496$
$2^5 = 32$	$2^{21} = 2\,097\,152$	$2^{37} = 137\,438\,953\,472$	$2^{53} = 9\,007\,199\,254\,740\,992$
$2^6 = 64$	$2^{22} = 4\,194\,304$	$2^{38} = 274\,877\,906\,944$	$2^{54} = 18\,014\,398\,509\,481\,984$
$2^7 = 128$	$2^{23} = 8\,388\,608$	$2^{39} = 549\,755\,813\,888$	$2^{55} = 36\,028\,797\,018\,963\,968$
$2^8 = 256$	$2^{24} = 16\,777\,216$	$2^{40} = 1\,099\,511\,627\,776$	$2^{56} = 72\,057\,594\,037\,927\,936$
$2^9 = 512$	$2^{25} = 33\,554\,432$	$2^{41} = 2\,199\,023\,255\,552$	$2^{57} = 144\,115\,188\,075\,855\,488$
$2^{10} = 1024$	$2^{26} = 67\,108\,864$	$2^{42} = 4\,398\,046\,511\,104$	$2^{58} = 288\,230\,376\,151\,711\,744$
$2^{11} = 2048$	$2^{27} = 134\,217\,728$	$2^{43} = 8\,796\,093\,022\,208$	$2^{59} = 576\,460\,752\,303\,423\,488$
$2^{12} = 4\,096$	$2^{28} = 268\,435\,456$	$2^{44} = 17\,592\,186\,044\,416$	$2^{60} = 1\,152\,921\,504\,606\,846\,976$
$2^{13} = 8\,192$	$2^{29} = 536\,870\,912$	$2^{45} = 35\,184\,372\,088\,832$	$2^{61} = 2\,305\,843\,009\,213\,693\,952$
$2^{14} = 16\,384$	$2^{30} = 1\,073\,741\,824$	$2^{46} = 70\,368\,744\,177\,664$	$2^{62} = 4\,611\,686\,018\,427\,387\,904$
$2^{15} = 32\,768$	$2^{31} = 2\,147\,483\,648$	$2^{47} = 140\,737\,488\,355\,328$	$2^{63} = 9\,223\,372\,036\,854\,775\,808$
			$2^{64} = 18\,446\,744\,073\,709\,551\,616$

On rappelle également que :

— 1 kio = 1024 octets,
— 1 Mio = 1024 kio,
— 1 Gio = 1024 Mio,
— 1 Tio = 1024 Gio,
— et ainsi de suite, avec dans l'ordre : Pio, Eio, Zio, Yio

En cas de doute, n'hésitez pas à demander des précisions.

Annexe : aide pour les appels système Unix

La liste ci-dessous présente, dans une forme légèrement simplifiée, les signatures des différents appels systèmes utilisés au cours de l'année.
Si vous hésitez sur les détails, n'hésitez pas les écrire en pseudo-code : vous passez un examen de système, pas de syntaxe C. Mais n'écrivez pas non plus du code «magique».

```
int  close(int fd); // returns 0 on success, -1 on error
int  execl(char *path, char *arg0, char *arg1, ..., NULL); // returns -1 on error
void exit(int status); // never returns
int  fork(void); // return 0 or pid
int  getpid(); // returns pid
int  getppid(); // returns pid
int  kill(int pid, int sig); // returns 0 on success, -1 on error
void *mmap(NULL, int len, int prot, int flags,
           int fd, int offset); // returns address of region
int  open(char *path, int oflag); // returns file descriptor
int  pthread_create(pthread_t *thread, NULL, void * (*function) (void *),
                    void *arg); // returns 0 on success, non-zero on error
void pthread_exit(void *retval); // never returns
int  pthread_join(pthread_t thread,
                  void **retvalp); // returns 0 on success, non-zero on error
int  read(int fd, void *buf, size_t nbyte); // returns nb of bytes read
int  sem_init(sem_t *sem, 0, unsigned int value); // returns 0 on success, -1 on error
int  sem_post(sem_t *sem); // returns 0 on success, -1 on error
int  sem_wait(sem_t *sem); // returns 0 on success, -1 on error
int  sleep(int seconds); // returns 0 on success, remaining nb of seconds otherwise
int  wait(NULL); // returns pid of child
int  waitpid(int pid, int *statusp); // returns pid of child
int  write(int fd, void *buf, size_t nbyte); // returns nb of bytes written
```