IF-3-SYS : Systèmes d'exploitation 8 juin 2023

NOM Prénom :

Consignes

- L'examen dure 1h30. Prenez le temps de lire le sujet en entier (15 questions sur 6 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.

1 Questions de cours

Dans les questions vrai/faux, les erreurs sont décomptées : ne répondez pas au hasard.

Noyau, processus, appels système

Question 1 Comment l'exécution d'un processus peut-elle se terminer ? Pour chaque proposition ci-dessous, entourez V s'il s'agit d'un scénario plausible, ou entourez F dans le cas contraire.

V F Le processus fait un appel système pour demander à se terminer.
V F Le processus a fini son programme : il n'a plus d'instructions à exécuter et donc il se termine.
V F Un autre processus vient de demander la fin de ce processus-ci.
V F Une erreur matérielle a provoqué le plantage du processus.

Question 2 Si on exécute le programme ci-dessous, combien de fois la lettre X sera-t-elle affichée ?

```
main()
{
   int r1 = fork();
   int r2 = fork();
   if(r1 * r2 == 0)
   {
      fork();
   }
   print("X");
}
```

La lettre X sera affichée [ ] fois en tout.

Question 3 Donnez deux exemples d'appels système «bloquants» (au sens de l'ordonnancement) et deux appels système «non bloquants».

Bloquants : [ ] et [ ]

Non-bloquants : [ ] et [ ]

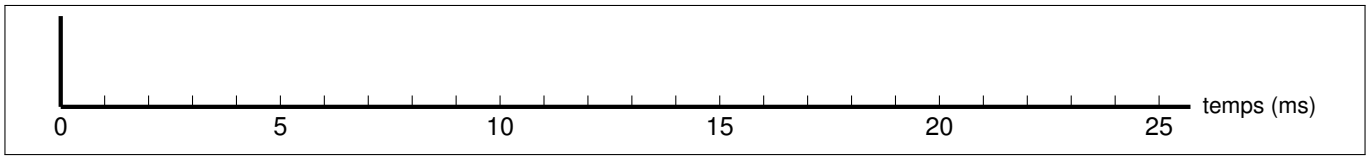
Ordonnancement

Question 4 Parmi les différentes stratégies d'ordonnancement vues en cours, laquelle est implémentée par le système d'exploitation de votre smartphone, et pourquoi ?

Question 5 On s'intéresse dans cette question à l'ordonnancement de deux processus qu'on nommera P1 et P2. L'exécution de P1 commence par une phase de calcul (CPU-burst) de 1ms, puis continue avec une phase d'entrées-sorties (IO-burst) pour une durée de 5ms, puis une seconde CPU-burst identique à la première, et ainsi de suite. L'exécution de P2 est similaire, sauf que chacune de ses IO-burst dure seulement 3 ms. Par ailleurs, P1 est prêt à l'instant initial, alors que P2 n'est lancé que 2 ms plus tard.

La machine dispose d'un unique processeur, ordonnancé selon la stratégie FCFS. Par ailleurs, elle ne peut faire qu'une seule opération d'entrées-sorties à la fois, et les requêtes IO sont également ordonnancées en FCFS.

Au brouillon, déroulez l'ordonnancement de P1 et P2 pendant 25 ms. Puis, dessinez ci-dessous un chronogramme illustrant la succession des tâches sur le processeur.



Gestion mémoire

Question 6 Complétez le paragraphe ci-dessous.

Grâce à la mémoire virtuelle, le système donne l'illusion aux [ ] de disposer d'une mémoire plus [ ] que celle réellement présente sur la machine. Cependant, cette mémoire leur apparaîtra aussi plus [ ] à cause des [ ] qui se produiront de temps en temps.

Question 7 Pour chaque proposition ci-dessous, entourez V si elle est correcte, ou entourez F si elle est fausse ou absurde.

V F Le temps d'accès à un «disque» SSD est de l'ordre de la nanoseconde.
V F Le temps d'accès à la mémoire principale (i.e. DRAM) est de l'ordre de la nanoseconde.
V F Le temps d'exécution d'une instruction est de l'ordre de la nanoseconde.
V F Le temps nécessaire pour traduire une adresse virtuelle vers une adresse physique est de l'ordre de la nanoseconde.

Question 8 L'outil `objdump` sert à examiner le contenu d'un fichier exécutable. En particulier, il permet de consulter le détail de :

V F la section `.text`
V F la section `.data`
V F la section `.stack`
V F la section `.heap`

Question 9 On a utilisé en TP différentes fonctions pour allouer de la mémoire, en particulier `malloc()` et `mmap()`. Mais pourquoi existe-t-il plusieurs fonctions similaires ? Plus précisément : dans quelles situations est-il plus avantageux pour le programmeur d'utiliser l'une ou l'autre ?
Donnez deux avantages de `malloc()` sur `mmap()` :

Donnez deux avantages de `mmap()` sur `malloc()` :

Concurrence et synchronisation

Question 10 Pour chaque élément de la liste ci-dessous, indiquez s'il est propre à un seul thread (entourez 1), ou s'il est partagé par l'ensemble des threads d'un processus (entourez N).

1 N L'espace d'adressage
1 N L'état d'ordonnancement
1 N La pile d'exécution
1 N Les registres CPU

Question 11 Soit le programme ci-dessous, composé de deux threads concurrents A et B. Complétez ce programme avec des synchronisations (sémaphores, structures de contrôle, etc) pour que l'affichage alterne toujours de la façon suivante : A-B-B-A-B-B-A-B-B-A-B-B...
N'oubliez pas d'indiquer les valeurs initiales de vos variables (entiers et sémaphores).

Conditions initiales

Thread A

```
while(true) {


  print("A");


}
```

Thread B

```
while(true) {


  print("B");


}
```

2 Exercice : Systèmes de gestion de fichiers

On s'intéresse dans cette partie à un système de gestion de fichiers basé sur une *File Allocation Table*. La FAT est un tableau d'entiers avec une case pour chaque bloc du disque :

- Si le bloc n° `i` est libre, alors `fat[i] = FAT_FREE = -2`
- Si le bloc n° `i` est le dernier bloc d'un fichier, alors `fat[i] = FAT_EOF = -1`
- Sinon, `fat[i]` est positif ou nul, et indique le numéro du bloc suivant dans le même fichier.

Dans toutes les questions suivantes, vous pourrez compter sur les déclarations ci-dessous :

```
enum { FAT_EOF = -1, FAT_FREE = -2 };
#define NB_BLOCKS ...
int fat[ NB_BLOCKS ] = { ... };
```

Question 12 Voici un exemple de FAT avec un disque formaté en 16 blocs.

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

Dans cet exemple, combien y a-t-il de fichiers, et quels sont les blocs qui composent chaque fichier ?

Question 13 Écrivez une fonction `int count_free_blocks_at(int pos)` qui parcourt la FAT et renvoie le nombre de blocs libres consécutifs à partir du bloc n° `pos` (inclus). Par exemple, avec les valeurs de la question précédente, on aura `count_free_blocks_at(1)` = 3, `count_free_blocks_at(7)` = 1, et `count_free_blocks_at(9)` = `count_free_blocks_at(10)` = 0.

Remarque : Utilisez les constantes `FAT_EOF`, `FAT_FREE`, `NB_BLOCKS` plutôt que leurs valeurs numériques.

```
int count_free_blocks_at(int pos)
{









}
```

Question 14 En principe, les différents blocs qui composent un fichier peuvent être situés n'importe où sur le disque, puisqu'ils sont chaînés par la FAT. En pratique cependant, placer ces blocs à la suite les uns des autres sur le disque donnera de meilleures performances. Écrivez une fonction `int find_hole_ff(int len)` qui parcourt la FAT, cherche un «trou» d'au moins `len` blocs libres consécutifs, et renvoie le premier numéro de bloc du trou choisi, ou -1 en cas d'échec. Vous appliquerez une stratégie de recherche *first-fit*, s'arrêtant au premier trou suffisamment grand.

```
int find_hole_ff(int len)
{









}
```

Question 15 Écrivez une fonction `int find_hole_bf(int len)` qui fait la même recherche mais en suivant une stratégie *best-fit*.

```
int find_hole_bf(int len)
{









}
```

Annexe : aide pour les calculs en binaire

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

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, etc. avec dans l'ordre : Pio, Eio, Zio, Yio. En cas de doute sur ces unités, n'hésitez pas à demander des précisions.