IFA-3-SYS : Systèmes d'exploitation 15 mars 2023

NOM Prénom :

Consignes

- L'examen dure 1h30. Prenez le temps de lire le sujet en entier (16 questions sur 8 pages)
- Écrivez lisiblement et surtout sans ratures. Utilisez un brouillon (vraiment).
- Les réponses seront à inscrire sur le sujet. Commencez par écrire votre nom ci-dessus.
- Documents et appareils interdits, sauf une feuille A4 recto-verso manuscrite.
- Pour le binaire et l'hexadécimal, aidez-vous des tableaux page 8.
- Dans les questions vrai/faux, les erreurs sont décomptées : ne répondez pas au hasard.

1 Noyau et processus

Question 1 Pour chacune des affirmations ci-dessous, entourez V si elle est correcte, ou entourez F si elle est incorrecte ou absurde.

V F Les systèmes d'exploitation Android, Fedora, et Ubuntu sont tous deux basés sur le même noyau Linux, donc ils offrent les mêmes appels système.

V F Pour des raisons de performance, le shell est en général implémenté comme un composant du noyau.

V F Si tous les processus du système sont suspendus, alors il faut rebooter la machine car aucun processus ne pourra plus être «réveillé».

V F Un changement de contexte est forcément causé par une trappe ou une interruption matérielle.

Question 2 Un processus parent crée un enfant avec `fork()` puis appelle la fonction `execl()`. On s'intéresse à ce qui se passe ensuite. Pour chacune des affirmations suivantes, entourez V si elle vous parait correcte, ou entourez F si elle est fausse ou absurde.

V F L'enfant continue à exécuter le code abandonné par le parent.

V F L'appel à `execl()` échoue et renvoie une erreur car ce scénario est invalide.

V F L'exécution de l'enfant suspendue jusqu'à ce que le parent se termine.

V F L'exécution du parent est suspendue jusqu'à ce que l'enfant se termine.

Question 3 Complétez la fonction ci-dessous de telle sorte que l'exécution de `chaine(N)` crée N nouveaux processus, avec une structure de parenté en forme de chaîne. Autrement dit, on veut que chaque processus de la chaîne (sauf le dernier) ait exactement un enfant.

```
void chaine(unsigned int N)
{

}
```

2 Ordonnancement

Question 4 Pour chacune des affirmations ci-dessous, entourez V si elle est correcte ou entourez F si elle est incorrecte et/ou absurde.

V F Dans un ordonnanceur Round Robin, plus on réduit le quantum, plus on augmente la fréquence des commutations de contexte.

V F Lorsqu'un processus cause un défaut de page, l'ordonnanceur le passe dans l'état `BLOCKED`.

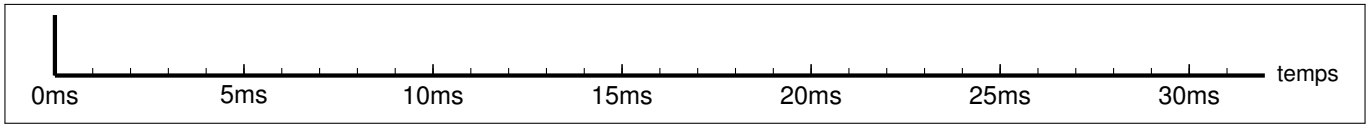
V F Par définition, un ordonnanceur préemptif est immunisé contre le risque de famine.

V F Pendant qu'un processus est à l'intérieur d'une section critique, il est interdit pour l'ordonnanceur de lui réquisitionner le CPU.

Question 5 Soient trois processus A, B et C avec les comportements suivants. L'exécution de A commence par une CPU-burst de durée 3ms, puis une IO-burst de durée 9ms, puis une nouvelle CPU-burst de durée 3ms, et ainsi de suite. De même, l'exécution de B alterne entre des CPU-burst de 7ms et des IO-burst de 3ms. L'exécution de C alterne entre des CPU-burst de durée 12ms, et des IO-burst de 1ms.

À l'instant initial, les trois processus A, B et C sont prêts (dans cet ordre) à exécuter leur première CPU-burst. On suppose que la machine peut faire plusieurs IO simultanément si besoin, mais qu'elle dispose d'un unique processeur, ordonnancé en Round Robin avec un quantum de 5ms.

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



Question 6 On se demande maintenant si les différents processus de la question précédente sont plutôt *CPU-bound* ou plutôt *IO-bound* ? Pour chaque processus, entourez la bonne réponse, ou entourez «NSP» si les informations données ne permettent pas de conclure.

- Processus A : CPU IO NSP
- Processus B : CPU IO NSP
- Processus C : CPU IO NSP

3 Mémoire virtuelle

Question 7 Dans cette question, le terme «mémoire physique» désigne la mémoire principale (typiquement, la DRAM). Pour chaque affirmation dans la liste ci-dessous, entourez V si elle est correcte, ou entourez F si elle est incorrecte ou absurde.

V F Grâce à la mémoire virtuelle, les accès à la mémoire physique paraissent plus rapides (du point de vue du processeur).

V F Le nombre de pages virtuelles disponibles pour un processus dépend de la quantité de mémoire physique installée sur la machine.

V F Sur une même machine, une page virtuelle et une page physique ont toujours exactement la même taille.

V F Quand on dit que «la machine swappe», c'est parce que le matériel est occupé à transférer des données entre le processeur et la mémoire physique.

Question 8 La signature de l'appel système `mmap()` est rappelée ci-dessous :

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Pour chacun des paramètres de `mmap()`, indiquez à quoi il sert, et donnez un exemple de valeur typique.

`addr` _____

_____

`length` _____

_____

`prot` _____

_____

`flags` _____

_____

`fd, offset` _____

_____

_____

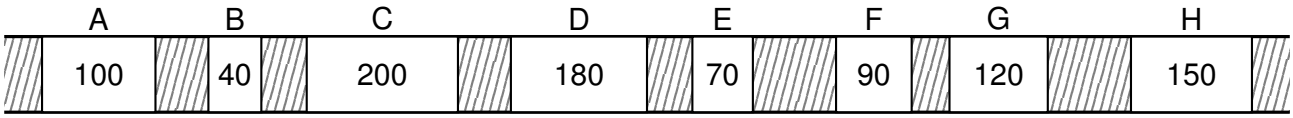
Question 9 On suppose dans cette question un système à mémoire virtuelle paginée avec les caractéristiques suivantes :

- les adresses virtuelles sont encodées sur 32 bits
- les adresses physiques sont encodées sur 48 bits
- la taille des pages est de 512kio
- la capacité de la mémoire physique (DRAM) est de 2Gio
- la capacité du stockage de masse (SSD) est de 32Gio

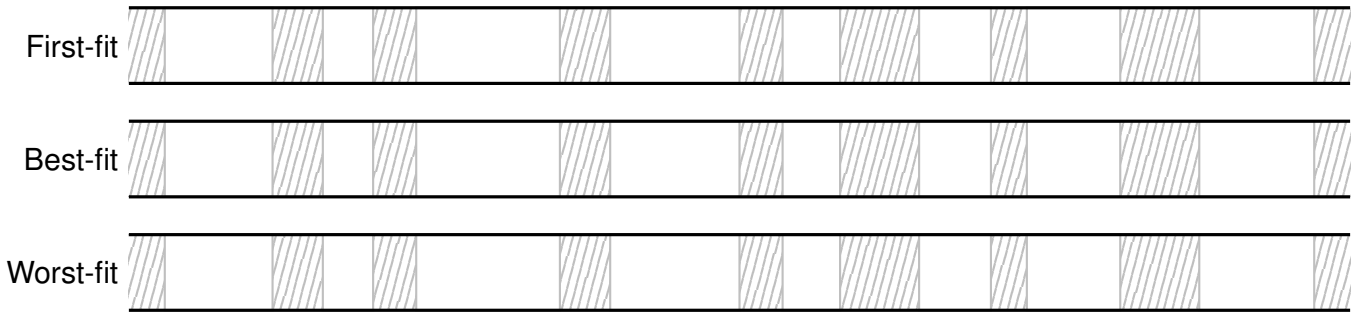
Quelle est la taille, en nombre de PTE, d'une table de pagination sur ce système ?
Si vous avez besoin d'aide avec les puissances de 2, reportez-vous au tableau page 8.

4 Allocation dynamique

Question 10 On s'intéresse dans cette question aux différentes stratégies d'allocation dynamique vues en cours. L'état initial du tas est illustré ci-dessous. La *freelist* comporte 8 blocs libres, chaînés dans cet ordre : bloc A (100 octets), bloc B (40 octets) etc, jusqu'au bloc H (150 octets).
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'espace occupé par les méta-données). Par ailleurs, les tailles des blocs (libres ou occupés) peuvent être des entiers quelconques : pas de restriction aux multiples de 8.



L'application demande successivement trois allocations, X, Y, et Z de respectivement 120, 100, et 90 octets. Pour chaque stratégie d'allocation, dessinez au brouillon l'état du tas après le traitement des trois requêtes. Indiquez en particulier l'emplacement de X, Y, et Z, ainsi que la taille des blocs encore libres. Si une stratégie ne parvient pas à satisfaire les trois requêtes, indiquez simplement «échec». Finalement, recopiez vos réponses au propre dans les cadres ci-dessous.



Question 11 On souhaite écrire une fonction pour compter les bits non-nuls dans la représentation binaire d'un entier `N`, qu'on supposera strictement positif. Par exemple, si `N` vaut 42 on doit retourner 3, car en binaire 42 s'écrit 101010. Complétez le code ci-dessous. Pour les calculs en binaire, vous pouvez vous aider des tableaux page 8.

```
unsigned int hamming_weight(unsigned int N)
{
   assert(N > 0);









}
```

5 Programmation concurrente : le dîner des philosophes

Initialement formulé en 1965 par Dijkstra, le problème dit du «dîner des philosopes» est un grand classique de la programmation à threads. On s'intéresse à cinq personnages qui passent leur temps à *penser* et à *manger*. Ils sont assis autour d'une table ronde où sont disposées cinq assiettes de spaghetti et cinq fourchettes (cf illustration ci-dessous). Chacun se comporte de la façon suivante : après avoir *pensé* à des questions existentielles pendant un temps indéterminé, notre philosophe se rend compte qu'il est affamé. Il *se saisit* alors des deux fourchettes entourant son assiette, et se met à *manger* pendant un temps indéterminé. Une fois repus, il pourra *reposer* ses deux fourchettes et se remettre à penser.

L'intérêt du problème réside dans le fait que nos philosophes ne savent pas manger proprement des spaghetti sans utiliser *deux* fourchettes à la fois. Par politesse, ils s'autorisent seulement à utiliser les deux fourchettes les plus proches. C'est pourquoi un philosophe affamé devra parfois attendre que l'un de ses voisins ait reposé ses fourchettes.

Code de chaque thread

```
philosophe(int i) // i = 0,1,...,4
{
   while(true)
   {
      penser(i); // jusqu'à avoir faim
      saisir_fourchettes(i);
      manger(i); // jusqu'à satiété
      poser_fourchettes(i);
   }
}
```

On suppose que les philosophes savent `penser()` et `manger()` par eux-mêmes, et on va s'intéresser uniquement à leur synchronisation, c.à.d. à l'implémentation des fonctions `saisir_fourchettes()` et `poser_fourchettes()`. Les fourchettes représentent des ressources partagées que les threads veulent utiliser en exclusion mutuelle. On les représentera donc comme autant de sémaphore binaires. À l'instant initial, toutes les fourchettes sont disponibles :

Variables partagées (état initial)

```
Semaphore fourchette[5] = {1,1,1,1,1};
```

Chaque philosophe étant numéroté de 0 à 4, les deux fonctions ci-dessous lui permettent de désigner facilement ses deux fourchettes :

```
int gauche(int i)
{
    return i;
}
```

```
int droite(int i)
{
    return ( i + 1 ) % 5;
}
```

Question 12 Le code ci-dessous est une première tentative naïve de synchronisation :

```
saisir_fourchettes(int i)
{
    P( fourchette[gauche(i)] );
    P( fourchette[droite(i)] );
}
```

```
poser_fourchettes(int i)
{
    V( fourchette[gauche(i)] );
    V( fourchette[droite(i)] );
}
```

Hélas, cette implémentation comporte un problème majeur qui expose nos philosophes à un péril mortel ! Quel est nom de ce risque, et comment peut-il se produire ?

Question 13 Les philosophes réalisent qu'ils ne pourront jamais manger tous en même temps. Pour se prémunir contre les risques identifiés à la question 12, ils décident d'adopter une nouvelle convention : (1) chaque philosophe reste debout pendant tout le temps où il *pense*, et (2) il ne s'assied que quand il devient *affamé*. Toutefois, (3) il est interdit de s'asseoir si tous les autres philosophes sont déjà assis.

Expliquez en quoi ces règles permettent d'éviter le problème.

Question 14 Dans les cadres ci-dessous, implémentez ce nouveau schéma de synchronisation. Pour chaque sémaphore que vous utilisez, pensez à préciser sa valeur initiale.

Variables partagées (état initial)

```
Semaphore fourchette[5] = {1,1,1,1,1};
```

```
saisir_fourchettes(int i)
{

}
```

```
poser_fourchettes(int i)
{

}
```

Question 15 Ce n'est pas très confortable de devoir rester debout pendant tout le banquet. Heureusement, les philosophes réalisent bientôt que ce n'était pas nécessaire. En effet, le danger vient uniquement des convives *affamés* :

— Un philosophe occupé à *penser* est inoffensif, car il n'a aucun besoin de fourchette. Lui permettre de s'asseoir ne posera donc pas de problème.

— De même, un philosophe en train de *manger* n'est pas en compétition avec ses voisins.

Les philosophes se donnent donc une nouvelle convention. Dorénavant, tout le monde reste assis par défaut, on se lève uniquement pour prendre des fourchettes, puis on se rassied. De plus, il est interdit de se lever si quelqu'un d'autre est déjà debout.

Expliquez en quoi ces règles permettent d'éviter le problème de la question 12.

Question 16 Dans les cadres ci-dessous, implémentez ce nouveau schéma de synchronisation. Pour chaque sémaphore que vous utilisez, pensez à préciser sa valeur initiale.

Variables partagées (état initial)

```
Semaphore fourchette[5] = {1,1,1,1,1};
```

```
saisir_fourchettes(int i)
{







}
```

```
poser_fourchettes(int i)
{







}
```

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.