

Examen Intra

IFT 2245

22 février 2023

Directives

- Vous avez droit à une page de notes (recto verso) écrite à la main.
- Calculatrice est autorisée.
- Répondez dans le cahier fourni à l'exception de Question 1 que vous pouvez répondre sur cette feuille.
- Le pointage pour chaque question est entre parenthèses (total = 20).
- Les traductions en anglais sont en *italics*.
- Vous pouvez répondre aux questions en anglais ou en français.
- Notez clairement toutes les suppositions que vous faites.

Question 0. *Nom et prénom (1 point de bonus)*

Écrivez votre nom et prénom et votre matricule en haut et sur la page couverture du cahier réponse.

Question 1. *Questions à choix multiple (3 points)*

(a) (0.5 points) Pourquoi le temps nécessaire pour créer un nouveau thread dans un processus est-il inférieur au temps requis pour créer un nouveau processus ? (cochez tout ce qui s'applique)

- En raison de la parallélisation
- Parce que nous n'avons pas besoin de créer une nouvelle pile
- Parce que nous n'avons pas besoin de créer un nouvel espace d'adressage
- Parce que nous n'avons pas besoin de créer un nouvel PCB

(b) (0.5 points) De quelles manières un processus peut-il arriver dans la "queue prête" (*ready queue*) ? (cochez tout ce qui s'applique) ?

- Être créé par un "fork()"
- Être interrompu
- Être bloqué par un E/S (*I/O*)
- Terminer
- Réalisation d'un E/S (*I/O*)
- Être choisi par l'ordonnanceur court-terme

(c) (0.5 points) L'ordonnanceur à long terme doit s'exécuter plus fréquemment que l'ordonnanceur à court terme :

- A. Vrai
- B. Faux

(d) (0.5 points) Il est possible d'avoir de la concurrence sans parallélisme :

- A. Vrai
- B. Faux

(e) (0.5 points) Étant donné qu'un changement de contexte prend temps = T, quelle est la durée maximale pendant laquelle un processus peut être bloqué pour qu'il soit plus efficace d'utiliser un spinlock au lieu d'un mutex ?

- A. Il est toujours plus efficace d'utiliser un spinlock
- B. $0.5 * T$
- C. $2 * T$
- D. T
- E. Il n'est jamais plus efficace d'utiliser un spinlock

(f) (0.5 points) L'ordonnancement par SJF (*shortest job first*) garanti un temps d'attente optimal.

- A. Vrai
- B. Faux

Question 2. Synchronization (5 points)

Voici une solution au problème des *dining philosophers* avec un moniteur :

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[N];
    condition self[N];
    condition eating;
    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i+(N-1))%N);
        test((i+1)%N);
    }
}
```

```
void eat (int i) {
    if (state[i] == EATING) {
        eating.wait();
        // eat
        eating.signal();
    }
}
```

```

void test(int i) {
    if ((state[(i+(N-1))%N] != EATING) && (state[i] == HUNGRY) && (state[(i+1)%N] != EATING) ) {
        state[i] = EATING;
        self[i].signal();
    }
}

initialization_code() {
    for (int i = 0; i < N; i++)
        state[i] = THINKING;
}
}

```

Où maintenant pour manger, une philosophe doit exécuter :

```

DiningPhilosophers.pickup(i);
...
// eat → DiningPhilosophers.eat(i)
...
DiningPhilosophers.putdown(i);

```

(a) (1 point) D'après cette solution, combien de philosophes peuvent manger en même temps (en fonction de N)?

$\lfloor N/2 \rfloor$

(b) (4 points) Supposons maintenant que nous voulions modifier ce qui précède afin qu'un seul philosophe puisse manger à la fois. Quelles modifications apporteriez-vous à la solution ci-dessus pour appliquer cela? (indice : maintenant la fonction *eat* devrait être à l'intérieur du moniteur)

Question 3. Opérations atomiques (1 point)

Considérant les fonctions atomiques *test_and_set()* et *compare_and_swap()* :

```

boolean test_and_set(boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}

int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

```

Comment pourrions-nous implémenter la fonction *test_and_set()*, en utilisant la fonction de *compare_and_swap()*?

```

boolean test_and_set(boolean *target)
{
    // utilise le compare_and_swap()
}

```

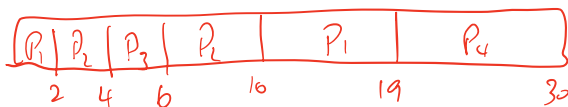
return (boolean *) compare_and_swap((int *) target, 0, 1)

Question 4. L'ordonnancement (4 points)

Pour un ensemble de processus $P_{1..4}$, nous anticipons la situation suivante :

(b) $\frac{1}{4}(19 + 8 + 2 + 24)$
 $\frac{53}{4} = 13.25$

Processus	Arrivée* (ms)	CPU burst time(ms)
P_1	0	11
P_2	2	6
P_3	4	2
P_4	6	11



* indique le moment où le processus est d'abord prêt à être exécuté.

- (a) (1 point) Dessinez le diagramme de Gantt pour l'algorithme "shortest remaining time first" (préemptif)
- (b) (1 point) Quel est le délai d'exécution moyen pour les 4 processus ?
- (c) (2 points) Considérons maintenant que tous les processus arrivent en même temps (mais sont toujours traités dans l'ordre P_1 à P_4)

Processus	Arrivée* (ms)	CPU burst time(ms)
P_1	0	11
P_2	0	6
P_3	0	2
P_4	0	11

si $q \geq 2, q \leq 6$
 T_{response} = $\frac{1}{4}(0 + Q + 2Q + 2Q + 2)$

Dans ce cas, nous planifions avec l'algorithme tourniquet (round robin). Quel est le quantum qui minimise le nombre de changements de contexte, mais assure également un temps de réponse moyen inférieur à 8 ms? = $5Q + 2$
 Quel est le nombre de changements de contexte résultant? (Vous pouvez supposer que le quantum est un nombre entier)

$\frac{1}{4}(5Q + 2) < 8$
 $Q < 6$

Question 5. Threads (3 points)

Considérez l'extrait de code suivant (supposons que toutes les structures de données sont correctement initialisées) :

```

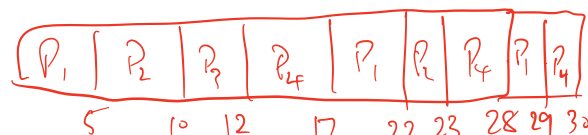
int value = 5;
void *runner(void *param);
pthread_mutex_t lock;

int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;

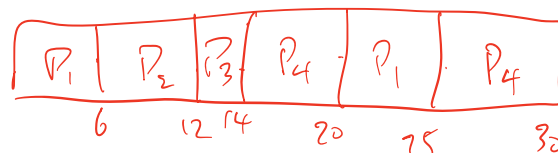
    pid = fork();
    if (pid == 0){

```

$Q = 5 \quad \#CC = 8$



$Q = 6 \quad \#CC = 5$



```

pid_t pid2;
pid2 = fork();

if (pid2 == 0){
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, NULL);
    pthread_join(tid, NULL);
}
else{
    waitpid(pid2, NULL, 0);
}

printf("LINE A, value=%d\n", value);
}
else{
    waitpid(pid, NULL, 0);
}

printf("LINE B, value=%d\n", value);
}

void *runner(void *param) {
    value += 5;
    pthread_exit(0);
}

```

LINE A, value = 10
 LINE B, value = 10
 LINE A, value = 5
 LINE B, value = 5
 LINE B, value = 5

qu'est-ce qui est imprimé ?

Question 6. Interblocage (4 points)

Considérons un système avec trois types de ressources, chacune avec un nombre différent d'unités totales :

A - 5, B - 7, C - 9

À un instant donné, il existe trois processus dont les allocations de ressources actuelles et maximales sont indiquées dans le tableau ci-dessous :

Processus	Alloué A, B, C	Max A, B, C	NEEDED			WORK		
			A	B	C	A	B	C
P_0	1, 0, 2	2, 5, 2	1	5	0	3	5	3
P_1	0, 1, 3	3, 4, 4	3	3	1	$P_1 \rightarrow$ 3	6	6
P_2	1, 1, 1	2, 5, 7	1	4	6	$P_0 \rightarrow$ 4	6	8

P_2 5-7-9.

Supposons que chacune des questions ci-dessous est indépendante (c'est-à-dire que l'état revient à l'état ci-dessus pour chaque question)

(a) (1 point) Quel est l'ordre d'exécution des processus qui démontrent que nous sommes dans un état sûr ?

$P_1 - P_0 - P_2$ ou $P_1 - P_2 - P_0$ ou $P_0 - P_1 - P_2$.

(b) (1 point) P_2 demande 1 ressource de type A et 1 ressource de type C, devraient-ils être accordés ? Pourquoi ou pourquoi pas ?

oui

(c) (2 points) Quel est le nombre maximum de chaque type de ressource pouvant être alloué à P_1 ?

- (a) Request $\leq (3, 5, 3)$
- (b) Request $\leq (3, 3, 1)$
- (c) et cetera sur $\leq (3, 3, 1)$

Available.
Needed.