Examen Intra

IFT-2245

February 20, 2017

Directives

- Documentation autorisée: *une* page *manuscrite* recto.
- Pas de calculatrice, téléphone, ou autre appareil électronique autorisé.
- Répondre sur le questionnaire dans l'espace libre qui suit chaque question. Utiliser le verso des pages si nécessaire.
- Chaque question vaut 5 points pour un total maximum de 25 points.
- Les questions ne sont pas placées par ordre de difficulté.

0 Nom et prénom (1 point de bonus)

Écrire son nom et prénom et son code permanent en haut de chaque page.

1 Primitives de synchronisation

Les variables "full/empty" sont des outils de synchronisation inspirés des principes du *dataflow*. Une telle "variable" est un objet avec deux états:

- elle peut être *vide*
- ou *pleine*, auquel cas elle contient une valeur (par exemple de type `void*`).

Elle a deux opérations, une qui met une valeur et doit attendre que la variable soit *vide* avant de pouvoir le faire, et une autre qui prend la valeur et doit donc attendre que la variable soit *pleine* avant de pouvoir le faire.

Utiliser ces "variables" (de type `struct fe_var`) pour implémenter les primitives habituelles d'un verrou: `acquire` et `release`.

```
extern void fe_put (struct fe_var *v, void *data);
extern void *fe_take (struct fe_var *v);

struct verrou {


};

void acquire (struct verrou *l) {









}

void release (struct verrou *l) {









}
```

2 Ordonnancement

Soit un système d'exploitation de style Unix utilisé pour un ordinateur de bureau. L'ordonnanceur utilise un système de priorités dynamiques, où chaque niveau de priorité a sa propre queue. Chaque niveau de priorité utilise un quantum de temps qui est le quadruple de celui de la priorité antérieure et le noyau change la priorité des processus en observant leur comportement.

1. Donner les 2 plus importants critères de performance que l'ordonnanceur aimerait optimiser.

2. Quand l'ordonnanceur diminue-t-il la priorité d'un processus?
   Quand l'ordonnanceur augmente-t-il la priorité d'un processus?

3. Si tout se passe bien, quel genre de processus devrait obtenir une priorité élevée, et quel genre obtiendra une priorité basse?

4. Soit un processus de simulation de qualité de l'air qui prend beaucoup de temps et de mémoire, et que l'utilisateur laisse tourner pendant qu'il développe frénétiquement la version suivante dans son éditeur préféré (Emacs, bien sûr).Quel niveau de priorité le noyau va-t-il donner à ce processus de simulation et pourquoi? Quel niveau de priorité le noyau va-t-il avoir tendance à donner à Emacs et pourquoi?

3 Conditions de course

Soit une fonction qui renvoie des "numéros de ticket", censés être uniques.

```
static int counter = 0;
int get_unique_ticket_number (void) {
  return counter++;
}
```

1. Lors d'une exécution, l'utilisateur constate que la fonction renvoie les nombres suivants: `..., 41, 42, 13, ...`
   Donner un scénario qui explique comment cela a pu se passer.

2. Après ces 3 nombres, quel pourrait-être le nombre suivant?
   Expliquer brièvement pourquoi.

3. Montrer comment éviter ces problèmes avec l'aide de l'instruction *compare&swap*.

4 Context switch

Placer les opérations suivantes dans l'ordre:

1. Le thread fait un appel système (*trap*) `read` pour lire du disque.

◯ L'ordonnanceur à court terme choisi un autre thread.

◯ Le CPU poursuit l'exécution du code du thread.

◯ Le CPU passe en mode *noyau*.

◯ Le CPU passe en mode *utilisateur*.

◯ Le SE charge le contenu des registres à partir du PCB.

◯ Le SE sauve le contenu des registres dans le PCB.

◯ Le SE envoie la commande au disque

5 QCM

	oui	*non*
L'ordonnancement par SJB (shortest job first) garanti un temps d'attente optimal	☐	☐
Lorsque le parent de P meurt avant P, le processus P est un zombie	☐	☐
Pour éviter les conditions de course, il "suffit" de toujours prendre les verrous dans le même ordre	☐	☐
Un petit *quantum* maximise le débit (*throughput*) de Round Robin	☐	☐
L'usage systématique de *moniteur* évite les interblocages	☐	☐
Un cache donne l'illusion d'une mémoire à la fois grande et rapide	☐	☐
Il est possible d'avoir de la concurrence sans parallélisme	☐	☐
La synchronisation optimiste ne souffre jamais d'inversion de priorité	☐	☐

Un thread parent et son thread enfant partagent:

- ☐ Les registres
- ☐ La pile
- ☐ Le tas
- ☐ Les variables globales