

# INF463 — Systèmes d'exploitation

## Devoir Surveillé

Durée : 1h30 — Sans document

### 1 Ordonnancement de processus (questions d'échauffement)

**Question 1** Dans un système d'exploitation interactif tel qu'Unix, rappelez précisément pourquoi un processus exécutant une boucle infinie ne monopolise pas pour autant le processeur s'il existe d'autres processus prêts dans le système.

**Question 2** Dans le cas où les autres processus ont une priorité de base très basse (fixée avec `nice`), est-ce encore vrai ?

**Question 3** Dans le simulateur Nachos, l'option `-rs` permet de simuler le comportement d'un système interactif. Expliquez le mécanisme utilisé par Nachos dans ce cas. Quelles en sont les limites, par rapport à un véritable système d'exploitation ?

### 2 Outils de synchronisation

Voici un exemple d'implémentation de *verrous* au sein d'un système d'exploitation :

```
typedef struct {
    unsigned value;
} mutex_t;

void mutex_lock(mutex_t *m)
{
    while(test_and_set(&m->value) == 1) /* rien */ ;
}

void mutex_unlock(mutex_t *m)
{
    m->value = 0;
}
```

**Question 1** Que fait l'appel à `test_and_set` ? Expliquez précisément le statut de cette instruction (est-ce une fonction ? un appel système ? autre chose ?)

Quelle est la propriété fondamentale de `test_and_set` ?

**Question 2** Voici une proposition d'alternative pour la fonction `mutex_lock` :

```
void mutex_lock(mutex_t *m)
{
    while(test_and_set(&m->value) == 1) {
        while(m->value == 1) /* rien */ ;
    }
}
```

Expliquez pourquoi cette version fonctionne également correctement. Selon vous, quel est son intérêt ?

**Question 3** Lorsqu'un processus se trouve en section critique (après un appel réussi à `mutex_lock`), il est possible qu'il soit interrompu et retiré du processeur par l'ordonnanceur au profit d'un autre processus. Quel problème va-t-on observer si ce dernier tente à son tour d'entrer dans la même section critique ? (On ne demande pas de proposer une solution dans cet exercice.)

**Question 4** Dans le simulateur Nachos, la plupart des synchronisations visant à entrer en section critique utilisent simplement la technique de *masquage des interruptions* : les interruptions sont d'abord masquées (i.e. leur réception est différée) avant d'entrer en section critique, puis rétablies<sup>1</sup> en sortie de section critique.

Expliquez pourquoi cette technique suffit à assurer qu'un seul thread noyau peut exécuter une section critique à la fois. Serait-ce imaginable d'utiliser uniquement cette technique pour implanter des sections critiques dans un vrai noyau (e.g. Linux) ? Pourquoi ?

**Question 5** Plus précisément, Nachos exécute l'appel `interrupt->SetLevel(IntOff)` pour masquer les interruptions et `interrupt->SetLevel(IntOn)` pour les rétablir. Voici pour illustration le squelette de la fonction `Semaphore::P` :

```
void Semaphore::P ()
{
    IntStatus oldLevel = interrupt->SetLevel (IntOff);
    // disable interrupts
    ...
    (void) interrupt->SetLevel (oldLevel); // re-enable interrupts
}
```

Expliquez pourquoi, à la fin de la fonction, c'est en réalité la valeur `oldLevel` qui est utilisée au lieu de `IntOn` ?

**Question 6 (difficile)** Indiquez précisément comment on pourrait utiliser la technique du masquage d'interruptions *en complément* de `test_and_set` pour corriger le problème évoqué à la question 3.

### 3 Barrières de synchronisation en deux temps

On dispose des primitives suivantes pour manipuler des moniteurs de Hoare :

```
typedef ... mutex_t ;
typedef ... cond_t ;
void mutex_lock(mutex_t *m);
void mutex_unlock(mutex_t *m);
void cond_wait(cond_t *c, mutex_t *m);
void cond_signal(cond_t *c);
void cond_bcast(cond_t *c);
```

**Question** En utilisant ces primitives, et en supposant l'existence d'une constante `MAX`, donnez le code des fonction `signaler()` et `attendre()` correspondant aux deux étapes de synchronisation d'une barrière en deux temps (telle que vue en cours) entre `MAX` processus.

On se limitera à une solution simple qui fonctionne dans le cas où les processus rejoignent une seule fois la barrière durant leur exécution.

---

<sup>1</sup>ce qui a notamment pour effet de déclencher immédiatement les interruptions en attente