université de BORDEAUX

Collège Sciences et technologies

| Année | 2024-2025 | Type | Devoir Surveillé |
|---|---|---|---|
| Master | Informatique | | |
| Code UE | 4I1N705U | Épreuve | Systèmes d'Exploitation |
| Date | 4/11/2021 | Documents | Non autorisés |
| Début | 16h15 | Durée | 1h30 |

**Plusieurs questions peuvent être traitées même si vous n'avez pas répondu aux précédentes.** *À la fin du sujet, vous trouverez un memento vous rappelant la syntaxe de quelques appels système utiles.*

# 1 Questions de cours (échauffement)

**Question 1** Rappelez le principe de fonctionnement et l'utilité du circuit appelé TLB au sein d'un processeur. Pourquoi n'a-t-on pas besoin de doter le TLB de beaucoup de mémoire ?

**Question 2** Expliquez en quoi consiste une opération de « changement de contexte » entre threads au sein d'un système d'exploitation. Y a-t-il une différence de traitement suivant que les threads appartiennent (ou non) au même processus ?

**Question 3** L'algorithme d'ordonnancement implanté dans les noyaux Linux 2.4.$x$ utilise un système de crédits que les processus sont autorisés à utiliser pendant la durée d'une « *époque* ». Tracez un petit chronogramme sur une durée de deux époques illustrant à quels moments surviendront les changements de contexte pour la configuration suivante de deux processus (`emacs` et `firefox`) :

- la priorité statique d'`emacs` lui donne droit à 2 crédits initiaux, alors que `firefox` a droit à 3 crédits au départ ;
- on suppose qu'`emacs` se bloque au milieu de sa première tranche de temps, et qu'il redevient prêt durant la quatrième tranche de temps de `firefox` sur le CPU.

Précisez bien le nombre de crédits que possède chaque processus à tout moment. Un processus peut-il encore avoir des crédits à la fin d'une époque ? Pourquoi ?

# 2 Atelier de réparation

On souhaite modéliser le fonctionnement d'un atelier de réparation de vélos. Les clients sont modélisés au moyen de *threads* dont le nombre et la date d'apparition sont aléatoires. Chaque thread exécute le code ci-dessous puis disparaît :

```
1 void client (void)
2 {
3   job_t j = deposit_for_repair ();
4
5   sleep (PROFITER_DU_TEMPS_LIBRE);
6
7   wait_repair_complete (j);
8 }
```

Lorsqu'un client dépose un deux-roues à l'atelier (ligne 3), il obtient une référence (de type `job_t`) qu'il pourra utiliser, plus tard, lorsqu'il ira patienter au comptoir que la réparation soit terminée (ligne 7).

Les deux-roues à réparer sont placés dans une file de taille bornée (`MAX_FIFO`). L'atelier dispose d'un certain nombre de techniciens (peu importe combien) dont le travail consiste à prendre un deux-roue dans la file (lorsqu'elle n'est pas vide), le réparer, puis recommencer...

Une implémentation préliminaire vous est fournie ci-dessous. Elle s'appuie sur une file bornée (`MAX_FIFO` articles) accessible au travers des primitives `__add`, `__remove` et `__size`. Aucune précaution particulière n'a été prise pour que ces primitives fonctionnent lorsqu'elles sont appelées de manière concurrente. Si la file est pleine, `__add` échoue et renvoie $-1$. De même, si la file est vide, `__remove` échoue et renvoie $-1$.

```c
typedef struct job_struct {
  unsigned repaired;
} *job_t;

// Implementation details of the FIFO are private
#define MAX_FIFO ...
int       __add    (job_t j);
int       __remove  (job_t *j);
unsigned __size    (void);

int deposit (job_t j)
{
  return __add (t);
}

int retrieve (job_t *j)
{
  return __remove (t);
}
```

```c
void worker (void)
{
  for (;;) { // each worker executes an infinite loop
    job_t j;

    if (retrieve (&j) == 0) {
      sleep (REPAIR_DURATION); // repair the bike!
      j->repaired = 1; // mark the job as done
    }
  }
}

job_t deposit_for_repair (void)
{
  job_t j = malloc (sizeof (struct job_struct));
  j->repaired          = 0;

  while (deposit (j) == -1) /* nothing */ ;

  return j;
}
```

Pour l'instant, on ne s'intéresse pas à l'implémentation de la fonction `wait_repair_complete`.

**Question 1** En rappelant l'absence de garantie sur l'implémentation des fonctions `__add` et `__remove`, pensez-vous que le code fonctionne tout de même correctement en présence d'un seul client et d'un seul technicien ? Expliquez. Par ailleurs, le code risque-t-il de solliciter maladroitement les processeurs de la machine ? Pourquoi ?

**Question 2** En utilisant les outils associés aux moniteurs de Hoare (cf memento en fin de sujet), ajoutez la synchronisation nécessaire aux fonctions `deposit` et `retrieve` de sorte que

— le thread technicien (`worker`) se bloque lorsque la file est vide ;

— le thread client (`deposit_for_repair`) se bloque lorsque la file est pleine.

Indiquez quelles sont les variables globales ajoutées.

**Question 3** On s'intéresse maintenant à l'implémentation de la fonction `wait_repair_complete` qui permet à un client d'attendre la fin de la réparation de son deux-roues. L'implémentation ci-dessous fonctionne-t-elle ? Quel est le problème ?

```c
void wait_repair_complete (job_t j)
{
  while (j->repaired == 0) /* nothing */ ;

  free (j);
}
```

Donnez-en une version corrigée en ajoutant au besoin des champs dans la structure `job_struct` et des variables globales. Indiquez les modifications à apporter à la fonction `worker`.

**Question 4** On suppose maintenant que la réparation de certains deux-roues nécessitent l'intervention simultanée de plusieurs techniciens, et qu'un nouveau champ `required_manpower` est automatiquement rempli dans `deposit_for_repair` (avec la garantie que la valeur est comprise entre 1 et le nombre total de techniciens). On dispose également d'une nouvelle version de la primitive `__remove` qui possède un second paramètre `unsigned inspect_only` : en passant 0 on extrait réellement l'élément de la file, en passant 1 on obtient un pointeur sur la structure sans retirer l'élément de la file.

Donnez la nouvelle version de la fonction `retrieve` dans laquelle les workers doivent attendre qu'ils soient le nombre requis avant de pouvoir réparer un deux-roues.

**Memento**

```c
typedef ... mutex_t;
void mutex_lock(mutex_t *m);
void mutex_unlock(mutex_t *m);
```

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