ASE - Contrôle de TD (durée 1h30 – documents autorisés)

Exercice 1 : Contextes et applications

Question 1: Contextes

1.1 : Expliquez le rôle des registres esp,ebp et eip sur une architecture $\geq$ i386, et l'intérêt de leur sauvegarde lors d'un changement de contexte.

1.2 : Expliquez ensuite comment ces trois registres sont sauvegardés/restaurés lors d'un changement de contexte dans le TP. Vous pouvez vous concentrer au cas d'un multitâche *coopératif* (appel explicite à yield ()).

Question 2: Rendez-vous

2.1 : Proposez un mécanisme de rendez-vous entre deux processus à l'aide de deux sémaphores initialisés à 0. Combien faut-il de sémaphores et d'instructions sem_down(sem) / sem_up(sem) pour généraliser ce mécanisme à 3, puis N processus ? Expliquez.

2.2 : Implantez un mécanisme de rendez-vous (méthode rendezvous()) entre N processus, à l'aide de sémaphores. De manière à éviter d'utiliser autant de sémaphores qu'il y a de processus, vous pourrez, par exemple, utiliser un mutex, un sémaphore barrière, et un compteur entier, en prenant garde à ne modifier le compteur qu'en exclusion mutuelle. On supposera que la méthode n'est utilisable que pour un et un seul rendez-vous.

2.3 : Proposez enfin une implantation directe de la méthode rendez-vous (sans utiliser de primitives associées aux sémaphores) dans votre TP.

Question 3: Mutex

d'après Philippe, sur une idée de Gilles

On souhaite un mécanisme d'exclusion mutuelle sous la forme de verrou. Un verrou peut être, soit libre, soit verrouillé par un processus : ce processus est alors dit *propriétaire* du verrou. Comparé aux sémaphores, l'utilisation des verrous est plus contraignante : seul le processus propriétaire du verrou peut le libérer et ainsi débloquer un autre processus en attente du verrou.

On souhaite réaliser un mécanisme de verrou *ré-entrant* : le processus propriétaire du verrou peut appeler autant de fois la fonction *lock* sur ce verrou, sans que cela ne le bloque. Bien entendu, une fois le verrou libéré par *unlock*, le processus en perd la propriété, qui est alors ré-attribuée à autre un processus bloqué (s'il en existe).

L'interface de manipulation des verrous est la suivante :

```
void mtx_init(struct mtx_s * mutex);
void mtx_lock(struct mtx_s * mutex);
void mtx_unlock(struct mtx_s * mutex);
```

3.1 : En vous inspirant de ce que nous avons réalisé pour les sémaphores en TP, donnez le code de la structure mtx_s et de la fonction void mtx_init(struct mtx_s * mutex) en expliquant vos choix.

3.2 : Proposez ensuite une implantation des deux fonctions void mtx_lock(struct mtx_s * mutex) et void mtx_unlock(struct mtx_s * mutex) en vous inspirant de ce que nous avons réalisé pour les sémaphores en TP.

Nous sommes dans la situation suivante :
- le processus A est propriétaire du mutex M_1 et M_6 et en attente du mutex M_5.
- le processus B est en attente du mutex M_2.
- le processus C est propriétaire du mutex M_2 et en attente du mutex M_3.
- le processus D est propriétaire du mutex M_3 et en attente du mutex M_4.
- le processus E est propriétaire du mutex M_4 et en attente du mutex M_2.
- le processus F est propriétaire du mutex M_5.

3.3 : Sommes nous en situation d'inter-blocage ? Quels sont les processus bloqués par ce *deadlock* ?

3.4 : Donner un algorithme qui puisse détecter de tels inter-blocages dans le système.

Exercice 2 : Fonctions longjmp/setjmp

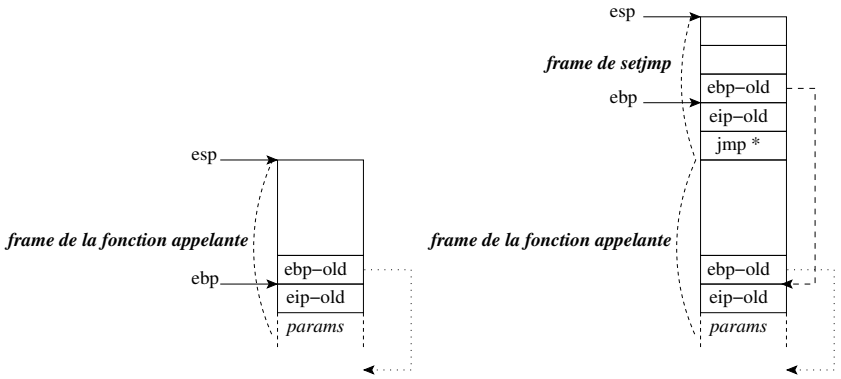
On souhaite, dans notre TP ordonnanceur, coder les clones des fonctions (utilisées lors du TP) `int setjmp(struct jmp_s * env)` et `void longjmp(struct jmp_s * env, int retvalue)` .

La fonction `setjmp` sauvegarde la position de *frame* de la fonction **appelante** (et non de sa propre *frame*) ainsi que le point de retour dans le code (`old-eip`) de la fonction appelante. La fonction `longjmp` est capable de restaurer la frame sauvegardée, et retourner dans le code de la fonction appelante au point sauvegardé par `setjmp`.

Question 1:

1.1 : Expliquez pourquoi `setjmp` ne sauvegarde pas sa *propre frame*. Expliquez en quoi cela diffère des fonctions `try` ou `switch_to_ctx` utilisées en TP.

1.2 : Donnez la structure `jmp_s` nécessaire pour sauvegarder un contexte *complet*.



Question 2:

En vous aidant des deux schémas (pile *avant* et *pendant* l'appel à `setjmp`),

2.1 : Expliquez comment obtenir, lors du déroulement de `setjmp`, la valeur des registres `esp`,`ebp` **avant** l'appel à `setjmp`, ainsi que la valeur `eip-old` ?

2.2 : Donnez le code de la fonction `setjmp`.

Avant de réaliser le code le la fonction `longjmp`, vous noterez que le résultat d'une fonction est toujours passé dans le registre `%%eax` : la fonction `longjmp`, qui est pourtant de type retour `void` peut ainsi retourner un résultat lorsqu'elle atteint le point de retour sauvegardé par `setjmp`.

Vous noterez enfin, qu'une fois les registres `ebp` et `esp` restaurés sur la frame de la fonction appelante, il ne sert à rien de continuer d'exécuter `longjmp` : en effet, toutes les manipulations qui y sont faites lors du dépilement de variables locales, paramètres locaux, ou pour restaurer `esp` et `ebp` sur la frame de la fonction qui a appelé `longjmp` sont trop dangereuses, car elles manipulent nos deux pointeurs de pile. Faites bref, et l'instruction assembleur `jmp <<adresse>>` peut dans ce cas vous être utile.

Question 3: Donnez le code de la fonction `longjmp`.