TD Rappel Systèmes d'Exploitation

Exercice 1 : Multi-programmation

On s'intéresse à l'exécution de trois tâches T_1, T_2 et T_3 :
- La tâche T_1 dure $200ms$ (hors E/S) et réalise une unique E/S au bout de $110ms$;
- La tâche T_2 dure $50ms$ et réalise une unique E/S au bout de $5ms$;
- La tâche T_3 dure $120ms$ sans faire d'E/S.

Les tâches T_1 et T_2 sont créées à l'instant 0, la tâche T_3 est créée à l'instant 140. Chaque E/S dure $10ms$. La valeur du quantum est de $100ms$.

Question 1.1 : Représentez sur deux diagrammes des temps l'exécution des 3 tâches en batch et en temps partagé. Calculez dans les deux cas le temps total d'exécution T_{ex} et le taux d'occupation T_{oc} du processeur.

Exercice 2 : Interruption horloge

On considère un système en temps partagé et on suppose pour simplifier qu'à chaque passage sur le processeur une tâche utilise la totalité de son quantum de temps. On suppose par contre qu'il y a un overhead de s milliseconde à chaque commutation.

Question 2.1 : Quel est le pourcentage de temps perdu à cause des commutations ?

Question 2.2 : On considère un quantum de $100ms$ et un overhead de $1ms$. Quelle serait la durée totale (temps CPU consommé, en incluant les commutations) d'une tâche demandant $20ms$ de calcul ? D'une tâche demandant $500ms$ de calcul ?

Exercice 3 : Processus

L'appel système $fork()$ crée un processus fils qui diffère de son père uniquement par ses numéros de *pid* et de *ppid*. L'appel $fork()$ renvoi 0 pour le fils et le *pid* du fils créé pour le père. Juste après le $fork()$, les deux processus disposent des mêmes valeurs de données et de pile mais il n'y a pas de partage : chaque processus a sa propre copie.

Soit le programme *C* suivant :

```
main( ) {
int pid ;
printf("Debut \n") ;
pid=fork() ;
if (pid == 0) then
   printf("execution 1 \n") ;
else
   printf("execution 2 \n") ;
end if
printf("Fin \n") ;
return (EXIT_SUCCESS) ;
}
```

Question 3.1 : Donnez les affichages effectués par le processus père et par le processus fils.

On considère maintenant le programme suivant :

```
int main(int argc, char * argv[]) {
int a, e;
a = 10;
if (fork() == 0) then
   a=a× 2;
   if (fork() == 0) then
      a=a+ 1;
      exit(2);
   end if
   printf("% d \n", a);
   exit(1);
end if
wait(& e);
printf("a :% d, e :% d \n", a, WEXITSTATUS(e));
return (0);
}
```

Question 3.2 : Donnez le nombre de processus créés, ainsi que les affichages effectués par chaque processus.

Question 3.3 : On supprime l'instruction $exit(2)$, reprenez la question précédente en conséquence.

Question 3.4 : Modifiez le programme initial pour créer un processus zombie pendant $30 secondes$.

Exercice 4 : Ordonnancement

On considère une variante de l'algorithme d'ordonnancement circulaire ou tourniquet. Les tâches sont rangées dans une file unique. Le processeur est donné à la première tâche prête de la file. La tâche perd le processeur en cas d'E/S ou quand elle a épuisé le quantum de temps. Elle est alors mise en fin de la file d'attente des tâches. Si une tâche arrive au début de la file dans l'état *bloqué* (attente d'une E/S), elle reste en début de file et on parcourt la file pour trouver une tâche prête. Toute nouvelle tâche est mise à la fin de la file.

On considère les tâches suivantes :

Tâche	Temps CPU	E/S	Durée E/S
T_1	$300ms$	aucune	
T_2	$30ms$	toutes les $10ms$	$250ms$
T_3	$200ms$	aucune	
T_4	$40ms$	toutes les $20ms$	$180ms$

On considérera un quantum de $100ms$, et on supposera que les tâches sont initialement toutes prêtes et rangées dans l'ordre de leur numéro et que les E/S des tâches T_2 et T_4 se font sur des disques différents.

Question 4.1 : Décrire précisément l'évolution du système : *instant, nature de l'événement (commutation, demande d'E/S, fin E/S, ...), état de la file et la tâche élue*. Donner pour chacune des tâches l'instant où elle termine son exécution.

Exercice 5 : Ressource critique

Un sémaphore est un objet permettant de contrôler l'accès à une ressource en évitant l'attente active. Il est constitué d'un compteur et d'une file d'attente. La valeur du compteur dénombre, lorsqu'elle est positive, le nombre de ressources disponibles, lorsqu'elle est négative le nombre de processus en attente de ressource. La politique de gestion de la file d'attente est définie par le concepteur du système. A la création d'un sémaphore, la file est vide. Un sémaphore est manipulé à l'aide des opérations indivisibles suivantes :

- $SEM * CS(cpt)$: création d'un sémaphore dont le compteur est initialisé à cpt.
- $DS(sem)$: destruction d'un sémaphore. Si la file n'est pas vide, un traitement d'erreur doit être effectué.
- $P(sem)$: demande d'acquisition d'une ressource. Si aucune ressource n'est disponible, le processus est bloqué.
- $V(sem)$: libération d'une ressource. Si la file d'attente n'est pas vide, un processus est débloqué.

Question 5.1 : Donnez le code de primitives P et V en utilisant les opérations suivantes :

- $TACHE * courant()$: désigne la tâche en cours d'exécution (la tâche élue),
- $void\ insérer(TACHE\ tache, FILE * file)$: insère une tâche dans la file,
- $TACHE * extraire(FILE * file)$: extrait une tâche de la file.

Question 5.2 : Expliquez pourquoi ces primitives doivent être rendues indivisibles et comment on peut réaliser cette indivisibilité.

Question 5.3 : Pourquoi n'utilise-t-on pas le masquage/démasquage des interruptions pour réaliser une section critiques en mode utilisateur.

On considère maintenant les trois processus suivants qui s'exécutent de manière concurrente, et le sémaphore Mutex initialisé à 1.

Processus A	***Processus B***	***Processus C***
(a) P(Mutex) ;	(d) P(Mutex) ;	(g) P(Mutex) ;
(b) x=x+1 ;	(e) x=x+1 ;	(h) x=x+1 ;
(c) V(Mutex) ;	(f) V(Mutex) ;	(i) V(Mutex) ;

On considère le scénario suivant : a d b g c e f h i

Question 5.4 : Donnez après chaque opération sur le sémaphore Mutex : la valeur du compteur, le contenu de la file du sémaphore, et l'état de chacun des processus (élu, prêt, bloqué).