Systèmes embarqués temps réel (GIF-3004)
Département de génie électrique et de génie informatique
Hiver 2017

UNIVERSITÉ LAVAL

EXAMEN PARTIEL

Instructions : – Une feuille aide-mémoire recto verso manuscrite est permise ;
– Durée de l'examen : 1 h 50.

Pondération : Cet examen compte pour 25% de la note finale.

Aide-mémoire sur quelques fonctions POSIX

Fonction	Fichier d'en-tête	Description
`int accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen)`	`sys/socket.h`	Créer socket anonyme pour nouvelles connections.
`int bind(int sockfd, const struct sockaddr* addr, socklen_t len)`	`sys/socket.h`	Associer socket à une adresse.
`int connect(int sockfd, const struct sockaddr* addr, socklen_t len)`	`sys/socket.h`	Ouvrir connexion client à un socket.
`int execve(const char *path, char *const argv[], char *const envp[])`	`unistd.h`	Exécuter un nouveau programme dans le processus actuel.
`pid_t fork(void)`	`unistd.h`	Créer un processus.
`int kill(pid_t pid, int sig)`	`signal.h`	Appeler signal du processus `pid`.
`int listen(int sockfd, int backlog)`	`sys/socket.h`	Préparer socket à recevoir connections.
`int pclose(FILE *fp)`	`stdio.h`	Fermer fichier et attends fin du processus.
`int pipe(int fd[2])`	`unistd.h`	Créer un pipe.
`FILE* popen(const char* cmdstring, const char* type)`	`stdio.h`	Lancer programme et lire E/S standard correspondantes.
`int pthread_attr_init(pthread_attr_t *attr)`	`pthread.h`	Créer structure d'attribut de thread.
`int pthread_attr_destroy(pthread_attr_t *attr)`	`pthread.h`	Détruire structure d'attributs de thread.
`int pthread_cancel(pthread_t tid)`	`pthread.h`	Demander terminaison du thread `tid`.
`void pthread_cleanup_pop(int execute)`	`pthread.h`	Retire fonction de terminaison la plus haute de la pile.
`void pthread_cleanup_push(void (*routine)(void *), void *arg)`	`pthread.h`	Ajouter routine sur la pile des fonctions appelées lors de la terminaison du thread.
`int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *attr, void *(*start_rtn)(void*), void *arg)`	`pthread.h`	Créer un thread.
`int pthread_cond_broadcast(pthread_cond_t *cond)`	`pthread.h`	Réveiller tous les threads en attente sur la condition.
`int pthread_cond_destroy(pthread_cond_t *cond)`	`pthread.h`	Détruire condition.
`int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr)`	`pthread.h`	Créer condition.
`int pthread_cond_signal(pthread_cond_t *cond)`	`pthread.h`	Signaler un thread en attente sur la condition.
`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`	`pthread.h`	
`int pthread_detach(pthread_t thread)`	`pthread.h`	Détacher thread `thread` du thread courant.
`int pthread_equal(pthread_t tid1, pthread_t tid)`	`pthread.h`	Déterminer si deux identifiants de threads sont les mêmes.
`void pthread_exit(void *rval_ptr)`	`pthread.h`	Terminer exécution du thread.
`int pthread_join(pthread_t thread, void **rval_ptr)`	`pthread.h`	Attendre fin de l'exécution du thread `thread`.
`int pthread_mutex_destroy(pthread_mutex_t *mutex)`	`pthread.h`	Détruire mutex.
`int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t attr)`	`pthread.h`	Créer mutex.
`int pthread_mutex_lock(pthread_mutex_t *mutex)`	`pthread.h`	Acquérir mutex (bloquant).
`int pthread_mutex_trylock(pthread_mutex_t *mutex)`	`pthread.h`	Tenter acquisition du mutex (non-bloquant).
`int pthread_mutex_unlock(pthread_mutex_t *mutex)`	`pthread.h`	Libérer mutex.
`pthread_t pthread_self(void)`	`pthread.h`	Obtenir identifiant unique du thread courant.
`ssize_t read(int fd, void *buf, size_t count)`	`unistd.h`	Lire données de fichiers.
`ssize_t recv(int sockfd, void *buf, size_t len, int flags)`	`sys/socket.h`	Recevoir données sur socket.
`ssize_t send(int sockfd, const void *buf, size_t len, int flags)`	`sys/socket.h`	Envoyer données sur socket.
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`	`signal.h`	Modifier routine de gestion de signal.
`int socket(int domain, int type, int protocol)`	`sys/socket.h`	Créer sockets (fonction générale).
`int socketpair(int domain, int type, int protocol, int sockfd[2])`	`sys/socket.h`	Créer sockets Unix pour communication inter processus.
`pid_t wait(int *stat_loc)`	`sys/wait.h`	Attendre fin d'un des processus enfants.
`pid_t waitpid(pid_t pid, int *stat_loc, int options)`	`sys/wait.h`	Attendre fin du processus enfant `pid`.
`ssize_t write(int fd, const void *buf, size_t count)`	`unistd.h`	Écrire données dans fichier.

Question 1 (27 points sur 100)

Soit le code suivant, implémentant un cas de producteur-consommateur avec deux threads POSIX et un tampon circulaire partagé.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  #define MAX 1000000
6  #define N 100
7
8  int tampon[N];                              /* Tampon circulaire partagé */
9  int p, c;                                   /* Compteurs de position */
10
11 void* prod(void *ptr) {
12     for(int i=0; i<MAX; ++i) {
13         while(((p+1)%N) == c) usleep(1000);  /* Tant que tampon plein, attendre 1 ms */
14         p = (p+1) % N;                       /* Incrémenter compteur */
15         tampon[p] = i;                       /* Produire donnée dans tampon */
16     }
17     pthread_exit(0);
18 }
19
20 void* cons(void *ptr) {
21     for(int i=0; i<MAX; ++i) {
22         while(c == p) usleep(1000);          /* Tant que tampon vide, attendre 1 ms */
23         c = (c+1) % N;                       /* Incrémenter compteur */
24         printf("%d ", tampon[c]);            /* Afficher donnée */
25         tampon[c] = -1;                      /* Consommer donnée dans tampon */
26     }
27     pthread_exit(0);
28 }
29
30 int main(int argc, char **argv) {
31     pthread_t pt_p, pt_c;                    /* Threads producteur et consommateurs */
32     p = c = 0;                               /* Initialiser compteurs de position */
33     pthread_create(&pt_c, NULL, cons, NULL); /* Lancer thread consommateur */
34     pthread_create(&pt_p, NULL, prod, NULL); /* Lancer thread producteur */
35     pthread_join(pt_p, NULL);                /* Attendre fin thread producteur */
36     pthread_join(pt_c, NULL);                /* Attendre fin thread consommateur */
37     return 0;
38 }
```

(5) (a) Ce code comporte deux problèmes principaux : 1) exécution potentiellement incorrecte du programme ; et 2) implémentation inefficace du programme. Expliquez précisément en quoi consistent ces problèmes relativement au code présenté, en indiquant les éléments spécifiquement problématiques (avec indications basées sur les lignes de code du programme).

(10) (b) Donnez une nouvelle version du code, en utilisant mutex et conditions POSIX, pour régler les deux problèmes mentionnés à l'énoncé de la sous-question précédente.

(12) (c) Produisez le code d'une nouvelle version de ce programme, cette fois-ci en suivant une approche multi processus (un processus producteur et un processus consommateur). Remplacez le tampon circulaire par des pipes Unix pour échanger les données entre ces deux processus.

Question 2 (25 points sur 100)

Soit les tâches temps réel données dans la table suivante, avec les temps de lancement, durées d'exécution, et échéances associés.

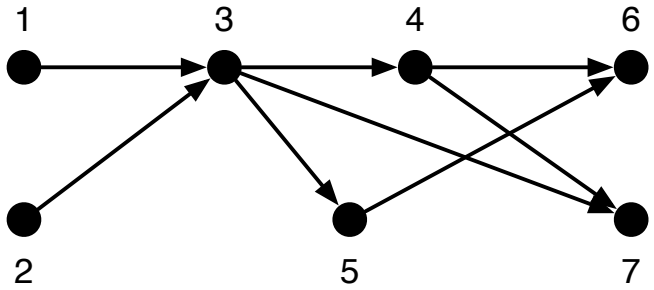
Tâche	1	2	3	4	5	6	7
Temps de lancement (r_i)	2	0	1	4	0	5	6
Durée d'exécution (e_i)	3	4	2	1	3	7	5
Échéance (d_i)	10	7	13	12	15	20	25

(5) (a) Effectuer un ordonnancement de type *round robin* sur un processeur, en démarrant l'exécution avec la tâche 2. Indiquez également si les contraintes d'exécution temps réel des tâches sont respectées.

(5) (b) Donnez l'ordonnancement par l'échéance la plus hâtive (EDF, *earliest deadline first*) sur un processeur, avec préemption. Indiquez si les contraintes d'exécution temps réel des tâches sont respectées.

(5) (c) Donnez maintenant l'ordonnancement de ces tâches pour exploiter **deux** processeurs, toujours en utilisant l'algorithme EDF, avec préemption et en supposant que les tâches ne peuvent pas migrer entre les processeurs une fois qu'elles sont démarrées (système statique). À la lumière de cet ordonnancement, indiquez s'il aurait été possible de faire un meilleur ordonnancement de ces tâches sur deux processeurs, pour réduire les temps de traitement, en justifiant votre réponse.

(5) (d) Soit le graphe de dépendances suivant entre les tâches.



Calculez les nouveaux temps de lancement effectifs et les nouvelles échéances effectives des tâches en tenant compte de ces dépendances.

(5) (e) En tenant compte des dépendances présentées à la sous-question précédente, déterminez s'il est possible d'ordonnancer ces tâches pour respecter les contraintes d'exécution temps réel des tâches, sur un seul processeur. Si oui, donnez un exemple de cédule permettant de le faire. Si non, indiquez avec justifications les contraintes qui ne peuvent pas être respectées dans ce contexte.

Question 3 (48 points sur 100)

Répondez aussi brièvement et clairement que possible aux questions suivantes.

(4) (a) Dans le cours, il a été expliqué que la variabilité des délais de traitement (*jitter*) est un problème important avec les systèmes temps réels. Expliquez en quoi précisément ceci est problématique dans ce contexte.

(4) (b) Il a été dit dans le cours qu'un contrôle activé par le temps est plus fréquent dans les systèmes temps réel durs, alors qu'un contrôle activé par les évènements est plus fréquent dans les systèmes temps réel doux. Expliquez pourquoi il en est ainsi.

(4) (c) Lorsqu'on fait la commande `man ls` dans un terminal sous un système Unix, indiquez combien de processus sont typiquement exécutés.

(4) (d) Indiquez pourquoi il est important de faire des appels aux fonctions `wait` ou `waitpid` dans les processus parents.

(4) (e) Expliquez pourquoi, dans Unix, les routines de gestion des signaux doivent être réentrantes.

(4) (f) Présentez les avantages et les désavantages d'un ordonnancement hors ligne dans des systèmes temps réel.

(4) (g) Dans Linux, de quelle façon tiens-t'on compte des priorités des différents processus dans l'assignation des ressources avec CFS (*completely fair scheduler*).

(4) (h) Les fautes de page ont été mentionnées comme étant un problème important dans les systèmes temps réel utilisant des systèmes d'exploitation. Indiquez en quoi ceci est un problème et de quelle façon peut-on l'éviter dans Linux.

(4) (i) Expliquez en quoi le mécanisme de copie sur écriture (*copy-on-write*) utilisé dans Linux pour l'allocation de mémoire aux processus est particulièrement efficace lors de création de processus enfants avec la fonction `fork`.

(4) (j) Expliquez ce en quoi consiste le problème d'inversion de priorités dans une application multi processus ou multithreadée.

(4) (k) Dans les 17 règles de la philosophie Unix d'Eric S. Raymond, présentées en classe, expliquez précisément la règle de *clarté*.

(4) (l) Expliquez quelles sont les différences entre les processus et les threads dans Linux.