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).

Solution:

Exécution incorrecte Le programme comporte une course (*race condition*) entre les threads producteur et consommateur qui peut mener à une exécution incorrecte aux lignes 23 à 25, car le thread consommateur peut être préempté entre les lignes 23 et 24, avec le thread producteur qui continue de produire des données, dont celle à `tampon[c]`, écrasant ainsi la valeur du tampon avant que celle-ci soit lue et affichée à la ligne 24 et modifiée à la ligne 25.

Inefficacité de l'implémentation Les threads font une attente active sur l'état des compteurs `p` et `c` (lignes 13 et 22), par une pause de 1 ms, ce qui peut ralentir significativement l'exécution du programme tout en générant de nombreux changements de contextes inutiles.

(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.

Solution:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  #define MAX 1000000
6  #define N 100
7  pthread_mutex_t mutex;                          /* Mutex */
8  pthread_cond_t cond;                            /* Condition */
9  int tampon[N];                                  /* Tampon partagé */
10 int p, c;                                       /* Compteurs de position */
11
12 void* producteur(void *ptr) {
13     for(int i=0; i<MAX; ++i) {
14         pthread_mutex_lock(&mutex);             /* Acquérir mutex */
15         if(((p+1) % N) == c)                    /* Si tampon plein */
16             pthread_cond_wait(&cond, &mutex);   /* Alors attendre sur condition */
17         p = (p+1) % N;                          /* Incrémenter compteur */
18         tampon[p] = i;                          /* Produire donnée dans tampon */
19         if(p == ((c+1) % N))                    /* Si tampon n'est plus vide*/
20             pthread_cond_signal(&cond);         /* Signaler consommateur */
21         pthread_mutex_unlock(&mutex);           /* Libérer mutex */
22     }
23     pthread_exit(0);
24 }
25
26 void* consommateur(void *ptr) {
27     for(int i=0; i<MAX; ++i) {
28         pthread_mutex_lock(&mutex);             /* Acquérir mutex */
29         if(c == p)                              /* Si tampon vide */
30             pthread_cond_wait(&cond, &mutex);   /* Alors attendre sur condition */
31         printf("%d ", tampon[c]);               /* Afficher donnée */
32         tampon[c] = -1;                         /* Consommer donnée dans tampon */
33         c = (c+1) % N;                          /* Incrémenter compteur */
34         if(c == ((p+2) % N))                    /* Si tampon n'est plus plein */
35             pthread_cond_signal(&cond);         /* Alors signaler producteur */
36         pthread_mutex_unlock(&mutex);           /* Libérer mutex */
37     }
38     pthread_exit(0);
39 }
40
41 int main(int argc, char **argv) {
42     pthread_t pt_p, pt_c;                       /* Threads */
43     p = c = 0;                                  /* Initialiser compteur */
44     pthread_mutex_init(&mutex, NULL);           /* Initialiser mutex */
45     pthread_cond_init(&cond, NULL);             /* Initialiser condition */
46     pthread_create(&pt_c, NULL, cons, NULL);    /* Lancer thread consommateur */
47     pthread_create(&pt_p, NULL, prod, NULL);    /* Lancer thread producteur */
48     pthread_join(pt_p, NULL);                   /* Attendre thread producteur */
49     pthread_join(pt_c, NULL);                   /* Attendre thread consommateur */
50     pthread_cond_destroy(&cond);                /* Détruire condition */
51     pthread_mutex_destroy(&mutex);              /* Détruire mutex */
52     return 0;
53 }
```

(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.

Solution:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  #define MAX 1000000
6
7  int main(int argc, char **argv) {
8      int fd[2];
9      pid_t pid;
10     if(pipe(fd) < 0) {          /* Créer pipe */
11         perror("Erreur avec pipe()");
12         return -1;
13     }
14     pid = fork();
15     if(pid < 0) {
16         perror("Erreur avec fork()");
17         return -1;
18     }
19     else if(pid > 0) {          /* Processus parent est le producteur */
20         close(fd[0]);           /* Fermer pipe côté réception */
21         for(int i=0; i<MAX; ++i) {
22             write(fd[1], &i, sizeof(i));  /* Produire et envoyer */
23         }
24         waitpid(pid, NULL, 0);  /* Attendre fin processus enfant */
25         close(fd[1]);           /* Fermer pipe */
26     }
27     else {                      /* Processus enfant est le consommateur */
28         close(fd[1]);           /* Fermer pipe côté expédition */
29         for(int i=0; i<MAX; ++i) {
30             int d = 0;          /* Variable réception de donnee */
31             read(fd[0], &d, sizeof(d));   /* Consommer donnée */
32             printf("%d ", d);             /* Afficher donnée */
33         }
34         close(fd[0]);           /* Fermer pipe */
35     }
36     return 0;
37 }
```

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.

Solution:

Temps	Tâche exécutée	File d'attente
1	2	5,3
2	5	3,2,1
3	3	2,1,5
4	2	1,5,3,4
5	1	5,3,4,2,6
6	5	3,4,2,6,1,7
7	**3**	4,2,6,1,7,5
8	**4**	2,6,1,7,5
9	2	6,1,7,5
10	6	1,7,5,2
11	1	7,5,2,6
12	7	5,2,6,1
13	**5**	2,6,1,7
14	**2**	6,1,7
15	6	1,7
16	**1**	7,6
17	7	6
18	6	7
19	7	6
20	6	7
21	7	6
22	6	7
23	**7**	6
24	6	
25	**6**	

Les contraintes temps réel ne sont pas respectées pour les tâches 1, 2 et 6, comme leurs temps de terminaison ne respectent pas leurs échéances respectives.

(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.

Solution:

Temps	1	2	3	4	5	6	7	8	9	10
Tâche	2	2	2	2	1	1	1	4	3	3
Temps	11	12	13	14	15	16	17	18	19	20
Tâche	5	5	5	6	6	6	6	6	6	6
Temps	21	22	23	24	25					
Tâche	7	7	7	7	7					

Oui, toutes les contraintes d'exécution temps réel des tâches sont respectées par cet ordonnancement.

(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.

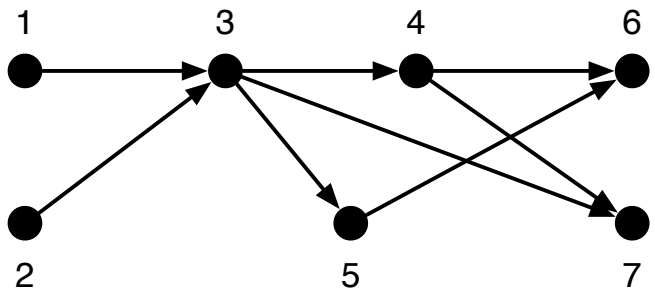
Solution:

Temps	1	2	3	4	5	6	7	8	9	10
Tâche proc 1	2	2	2	2	4	6	6	6	6	6
Tâche proc 2	5	3	1	1	1	3	5	5	7	7

Temps	11	12	13	14	15	16	17	18	19	20
Tâche proc 1	6	6								
Tâche proc 2	7	7	7							

Avec une telle cédule, il n'est pas possible d'être plus efficace, car la parallélisation maximale est atteinte. En effet, comme la somme des durées d'exécution des tâches est de 25 pas de temps, la parallélisation maximale sur deux processeurs devrait permettre une exécution au mieux de $\lceil 25/2 \rceil = 13$ pas de temps, ce qui est le résultat que nous obtenons.

(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.

Solution:

Tâche	1	2	3	4	5	6	7
Temps de lancement effectif (r'_i)	2	0	**5**	**7**	**7**	**10**	**8**
Durée d'exécution (e_i)	3	4	2	1	3	7	5
Échéance effective (d'_i)	**8**	7	**10**	12	**13**	20	25

(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.

Solution: On sait que l'ordonnancement par algorithme EDF permet de trouver la cédule optimale sur un processeur, si celle-ci existe. Nous allons donc appliquer EDF de nouveau pour voir si on peut obtenir cette cédule.

Temps	1	2	3	4	5	6	7	8	9	10
Tâche	2	2	2	2	1	1	1	3	3	4
Temps	11	12	13	14	15	16	17	18	19	20
Tâche	5	5	5	6	6	6	6	6	6	6
Temps	21	22	23	24	25					
Tâche	7	7	7	7	7					

Cette cédule est valide. Donc, oui, il est possible d'ordonnancer ces tâches en tenant compte des dépendances tout en respectant les contraintes d'exécution temps réel, un exemple de cédule valide étant présenté ci-haut.

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.

Solution: Dans les systèmes temps réels, en particulier les systèmes temps réel dur, il est important d'avoir des délais de traitement aussi stables que possibles, avec peu de variations dans leur durée d'exécution. Si la variabilité est importante, une exécution avec un niveau de confiance raisonnable à l'intérieur d'un délai raisonnable devient difficile.

(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.

Solution: Avec un contrôle activé par le temps, les tâches a effectuées sont connues à l'avance et l'ordonnancement correspondant à celle-ci peut se faire hors ligne. De cette façon, les délais sont prévisibles et les perturbations provenant de l'environnement externe au système ont peu ou pas d'impacts sur la réalisation des tâches. De telles caractéristiques sont désirables dans un système temps réel dur, où l'on veut limiter la variabilité dans l'exécution d'une tâche et où l'interaction avec les évènements provenant de l'environnement sont peu fréquents.

Avec un contrôle activé par les évènements, le système est plus à la merci de son environnement. Certaine séquence d'évènements peut augmenter la charge de traitement et faire que les contraintes d'exécution des tâches temps réel ne soient pas toutes respectées. Ceci est caractéristique d'un système temps réel doux, où le manquement des contraintes peut être toléré.

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

Solution: Un seul processus sera exécuté, soit celui de la commande `man`, l'argument `ls` étant le nom de la commande pour laquelle nous voulons le manuel, cette commande en soit ne sera pas exécutée.

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

Solution: Les fonctions `wait` ou `waitpid` permettent de libérer la table des processus du systèmes d'exploitation, lorsque les processus enfants du processus parent actuel ont terminé leur exécution. Autrement, les processus enfants ayant terminé, mais dont le statut n'a pas été collecté par le processus parent restera actif comme processus zombie, consommant inutilement des ressources.

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

Solution: Les routines de gestion des signaux peuvent être appelées à tout moment dans l'exécution d'un programme, possiblement même alors qu'une autre routine de gestion des signaux est en cours d'exécution. Le code doit donc être fait pour être réentrant, pour assurer qu'elles peuvent être appelées à tout moment sans risque de corrompre des structures de données partagées.

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

Solution: Un ordonnancement hors ligne a l'avantage de pouvoir générer des cédules optimales pour des ensembles de tâches complexes à ordonnancer. En effet, dans ce contexte, le temps de traitement pour déterminer la cédule à utiliser n'est pas très important et peut se faire sur des ordinateurs puissants. Le désavantage principal de l'approche est que la cédule doit être calculée au préalable et peut difficilement être modifiée en cours d'exécution, par exemple lorsque des évènements imprévus provoque des changement relativement aux tâches à exécuter.

(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*).

Solution: Les priorités des processus sont utilisées pour déterminer la juste part du temps de processeur de chacun des processus actifs. L'algorithme CFS vise à donner aux processus actifs au moins l'équivalent de cette juste part durant leur exécution.

(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.

Solution: Les fautes de page impliquent de devoir échanger une quantité important de données entre la mémoire vive et le disque dur, ce qui peut introduire des délais d'exécution très importants à des endroits en apparence bénigne (ex. écriture en mémoire), sans être facilement prévisible.

Dans Linux, on peut éviter ce problème en bloquant les pages de mémoire utilisées par les processus correspondant à des tâches temps réel en utilisant les fonctions `mlock` ou `mlockall`.

(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`.

Solution: Lors de l'appel de la fonction `fork`, une copie exacte du processus parent est faite pour le processus enfant. Cependant, cette copie n'est pas faite en pratique, les processus parent et enfant possèdent la même copie, en lecture seulement. C'est seulement lors d'une écriture qu'une copie est faite. De plus, si le processus enfant consiste en l'exécution d'un programme différent que le processus parent, aucune copie inutile du programme du processus parent n ?est effectuée lors de l'appel de la fonction `fork`.

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

Solution: Le problème d'inversion de priorité consiste en deux processus (ou threads) de priorités différentes, voulant accéder en même temps à une ressource partagée. Le processus de priorité plus basse obtient la ressource avant le processus de priorité plus élevée, mais est préempté par d'autres processus externes de priorités intermédiaires. Dans ce cas, le processus de basse priorité sera bloqué tant que les processus intermédiaires monopoliseront les processeurs, faisant en sorte que le processus de priorité plus élevée soit également bloqué, comme s'il avait la même priorité du processus de priorité faible.

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

Solution: La règle de clarté consiste à préférer du code lisible par des humains à du code ingénieux, mais illisible.

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

Solution: La seule différence dans Linux est au niveau du partage de la mémoire, tous les threads d'un programme partagent la mémoire du processus associés. Les processus quant à eux ne partagent aucun espace mémoire entres eux, à moins de configuration particulière. Les threads sont autrement traités comme des processus distincts par l'ordonnanceur.