Systèmes embarqués temps réel (GIF-3004)
Département de génie électrique et de génie informatique
Hiver 2020

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 et Linux

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 ftruncate(int fildes, off_t length)`	`unistd.h`	Changer taille d'un fichier.
`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 mlock(const void *addr, size_t len)`	`sys/mman.h`	Bloquer une plage en mémoire vive.
`int mlockall(int flags)`	`sys/mman.h`	Bloquer tout l'espace du processus en mémoire vive.
`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`	`sys/mman.h`	Créer une image mémoire.
`int munlock(const void *addr, size_t len)`	`sys/mman.h`	Débloquer une plage en mémoire vive.
`int munlockall(void)`	`sys/mman.h`	Débloquer tout l'espace du processus en mémoire vive.
`int munmap(void *addr, size_t length)`	`sys/mman.h`	Désallouer une image mémoire.
`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 shm_open(const char *name, int oflag, mode_t mode)`	`sys/mman.h`	Partage d'un emplacement mémoire.
`int shm_unlink(const char *name)`	`sys/mman.h`	Désallocation d'un emplacement mémoire.
`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 (47 points sur 100)

Supposons une application qui vise à faire l'affichage de messages provenant de différentes sources dans une console. L'application est organisée selon deux programmes, un programme serveur qui reçoit et affiche les messages dans la console et un programme client qui lit les messages sur la ligne de commande pour les transmettre au serveur, via un socket Unix (socket local). Les deux programmes suivants donnent l'implémentation du serveur (`chtrmserv.c`) et du client (`chtrmclnt.c.`) de l'application.

`chtrmserv.c` :

```c
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <limits.h>
#include <sys/socket.h>
#include <sys/un.h>

#define MAXSIZE 255
#define MAXCLIENTS 4

int main(int argc, char** argv) {
  struct sockaddr_un serv_addr, clnt_addr;
  char roomn[MAXSIZE+1], messg[MAXSIZE+1];
  int  serv_sockfd, clnt_sockfd, i, rc, rdb;

  if(argc < 2) { fprintf(stderr, "usage> %s roomname\n", argv[0]); exit(1); }
  strncpy(roomn, argv[1], MAXSIZE);

  serv_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
  if(serv_sockfd < 0) { perror("socket() failed"); exit(1); }

  memset(&serv_addr, 0, sizeof(serv_addr));
  serv_addr.sun_family = AF_UNIX;
  strncpy(serv_addr.sun_path, roomn, sizeof(serv_addr.sun_path)-1);
  rc = bind(serv_sockfd, (struct sockaddr*)&serv_addr, SUN_LEN(&serv_addr));
  if(rc < 0) { perror("bind() failed"); exit(1); }

  rc = listen(serv_sockfd, MAXCLIENTS);
  if(rc < 0) { perror("listen() failed"); exit(1); }

  for(i=0; i<INT_MAX; ++i) {
    clnt_sockfd = accept(serv_sockfd, NULL, NULL);
    if(clnt_sockfd < 0) { perror("accept() failed"); exit(1); }
    rdb = 0;
    for(rc=-1; (rc!=0) && (rdb<MAXSIZE); rdb+=rc) {
      rc = read(clnt_sockfd, messg+rdb, MAXSIZE-rdb);
      if(rc < 0) perror("read() from socket failed");
    }
    printf("%i: %s\n", i, messg);
    close(clnt_sockfd);
  }

  close(serv_sockfd);
  unlink(roomn);
  return 0;
}
```

`chtrmclnt.c` :

```c
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

#define MAXSIZE 255

int main(int argc, char** argv) {
  struct sockaddr_un serv_addr;
  char roomn[MAXSIZE+1], messg[MAXSIZE+1];
  int  sockfd, rc;

  if(argc < 2) { fprintf(stderr, "usage> %s roomname message\n", argv[0]); exit(1); }
  strncpy(roomn, argv[1], MAXSIZE);
  if(argc < 3) strncpy(messg, "", MAXSIZE);
  else strncpy(messg, argv[2], MAXSIZE);

  sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
  if(sockfd < 0) { perror("socket() failed"); exit(1); }
  memset(&serv_addr, 0, sizeof(serv_addr));
  serv_addr.sun_family = AF_UNIX;
  strncpy(serv_addr.sun_path, roomn, sizeof(serv_addr.sun_path));
  rc = connect(sockfd, (struct sockaddr*)&serv_addr, SUN_LEN(&serv_addr));
  if(rc < 0) { perror("connect() failed"); exit(1); }

  write(sockfd, messg, strnlen(messg, MAXSIZE)+1);
  close(sockfd);
  return 0;
}
```

(4) (a) Pour les deux programmes, un premier argument obligatoire sur la ligne de commande est requis. Expliquez précisément à quelle fin cet argument est utilisé par les deux programmes dans la mécanique de communication par sockets Unix.

(4) (b) Expliquez pourquoi un appel à la fonction `strncpy` avec comme argument `MAXSIZE` est effectué à la ligne 19 de `chtrmclnt.c` à la place d'un appel fait à `strcpy`. Expliquez un cas général d'utilisation où la version avec `strcpy` serait problématique.

(4) (c) Si on veut lancer une deuxième fois le programme serveur avec le même argument sur la ligne de commande sans que le premier lancement soit complété ou interrompu, est-ce que vous croyez que le programme serveur pourra être exécuté ? Si non, indiquez à quelle ligne du code de `chtrmserv.c` le deuxième lancement du programme échouera, en justifiant brièvement votre réponse.

(4) (d) Expliquez précisement pourquoi la lecture du message reçu par le programme serveur doit se faire dans une boucle for (lignes 37 à 41 de `chtrmserv.c`).

(11) (e) Si le programme `chtrmserv.c` est interrompu (ex. par `CTRL-C` dans le terminal), est-ce que vous croyez que la terminaison se fera correctement ? Justifiez votre réponse.

Proposez également une modification au code `chtrmserv.c` du serveur afin que celui-ci gère correctement les signaux `SIGINT` et `SIGTERM` afin de faire une terminaison convenable du programme.

(20) (f) Faites maintenant une version multithreadée du code du serveur (`chtrmserv.c`). Le serveur multithreadé devrait fonctionner de la façon suivante :
- Au démarrage, le programme doit lancer `MAXCLIENTS` threads de traitement, en plus du thread principal (fonction `main`). Ces threads sont en attente d'une tâche.
- Le thread principal gère les requêtes entrantes et distribue les tâches aux threads de traitement en les affectant au socket du client reçu via la fonction `accept` (ligne 35).
- Pour chaque connexion d'un client acceptée, un des threads de traitement fait les opérations décrites dans les lignes 37 à 43 de `chtrmserv.c`. Plus précisément, pour chaque connexion client, un thread de traitement est réveillé et reçoit le socket client (retourné comme variable `clnt_sockfd` à la ligne 35) qu'il doit traiter. Une fois le traitement complété, le thread de traitement se replace en attente d'une nouvelle connexion client à traiter.
- L'affichage dans la console (ligne 42 de `chtrmserv.c`) doit être bien encadré par une primitive de synchronisation pour éviter des appels simultanés à la fonction `printf` par plusieurs threads distincts.

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.

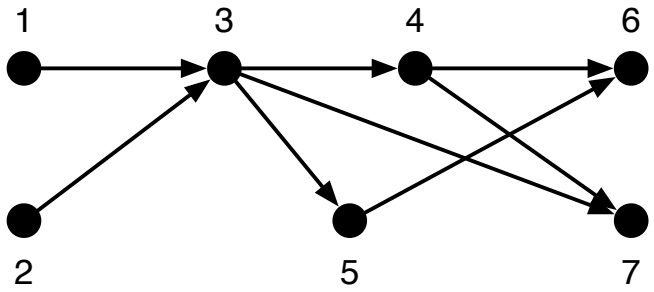
Tâche	1	2	3	4	5	6	7
Temps de lancement (r_i)	2	0	1	4	0	5	6
Durée d'exécution (e_i)	3	4	2	1	3	7	5
Échéance (d_i)	10	7	13	12	15	20	25

(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 (28 points sur 100)

Répondez aussi brièvement et clairement que possible aux questions suivantes.

(4) (a) Soit la ligne de commande suivante dans un terminal (shell) Unix.

```
ls -l /etc/ | grep conf | less
```

Expliquez ce que cette commande effectue précisément comme opérations. Indiquez tous les processus (programmes et arguments) qui seront lancés avec cette ligne commande.

(4) (b) Expliquez l'intérêt d'utiliser des *reader-writer locks* comparativement à des mutex standards.

(4) (c) Expliquez pourquoi la fonction `sched_yield` est importante lorsque l'on implémente des programmes temps réel dans Linux avec l'ordonnanceur `SCHED_DEADLINE`.

(4) (d) Indiquez les deux principaux changements que la patch PREEMPT_RT effectue au noyau Linux.

(4) (e) La gestion de la mémoire dans Linux se fait selon une approche *copy-on-write*. Ceci est efficace en général, mais peut être problématique dans un contexte temps réel si aucune disposition particulière n'est prise. Expliquez le problème précisément.

(4) (f) Expliquez pourquoi la variabilité dans la durée des temps de traitements (*jitter*) est à éviter dans la mesure du possible, dans les systèmes temps réels.

(4) (g) Dans Linux, de quelle façon tient-on compte des priorités des différents processus dans l'assignation des ressources avec CFS (*completely fair scheduler*).