**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** :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stddef.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <sys/socket.h>
7  #include <sys/un.h>
8
9  #define MAXSIZE 255
10
11 int main(int argc, char** argv) {
12   struct sockaddr_un serv_addr;
13   char roomn[MAXSIZE+1], messg[MAXSIZE+1];
14   int  sockfd, rc;
15
16   if(argc < 2) { fprintf(stderr, "usage> %s roomname message\n", argv[0]); exit(1); }
17   strncpy(roomn, argv[1], MAXSIZE);
18   if(argc < 3) strncpy(messg, "", MAXSIZE);
19   else strncpy(messg, argv[2], MAXSIZE);
20
21   sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
22   if(sockfd < 0) { perror("socket() failed"); exit(1); }
23   memset(&serv_addr, 0, sizeof(serv_addr));
24   serv_addr.sun_family = AF_UNIX;
25   strncpy(serv_addr.sun_path, roomn, sizeof(serv_addr.sun_path));
26   rc = connect(sockfd, (struct sockaddr*)&serv_addr, SUN_LEN(&serv_addr));
27   if(rc < 0) { perror("connect() failed"); exit(1); }
28
29   write(sockfd, messg, strnlen(messg, MAXSIZE)+1);
30   close(sockfd);
31   return 0;
32 }
```

(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.

**Solution:** Le premier argument sur la ligne de commande correspond à un nom unique pour mettre en relation les programmes de l'application. Il est utilisé pour identifier et publier le socket Unix du serveur que les clients peuvent utiliser pour débuter la communication. Ce nom correspond à un fichier créé localement et vide.

(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.

**Solution:** La fonction `strcpy` fait une copie qui s'interrompt seulement lorsque le caractère `\0` signalant la fin de la chaîne de caractères est rencontré. Si le message sur la ligne de commande donné dans `argv[2]` excède la taille `MAXSIZE` imposé à la chaîne de caractères `messg`, une écriture aurait été faite dans une région de la mémoire qui n'est pas allouée à la chaîne de caractères, pouvant ainsi générer une corruption de la mémoire. Avec l'appel à `strncpy`, on s'assure ainsi que la copie de la chaîne de caractères s'interrompt à la fin de l'espace mémoire alloué à `messg`.

(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.

**Solution:** Le deuxième lancement du programme serveur avec le même nom donné comme argument sur la ligne de commande va échouer. Le programme va échouer à la ligne 29 de `chtrmserv.c`, où la valeur de retour de l'appel à la fonction `bind` va échouer, comme un socket associé au nom dans `roomn` est déjà en fonction – dans la configuration par défaut, un deuxième lancement de serveur avec un appel à la fonction `bind` ne permettra pas le lancement, retournant une valeur négative dans `rc` pour intercepter la situation problématique.

(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`).

**Solution:** Un appel à la fonction `read` ne garantit pas que le message sera reçu en entier. Pour de longs messages, il est possible que l'on doive faire plusieurs appels consécutifs à la fonction `read` pour obtenir le message en entier, soit jusqu'à ce que la fonction nous retourne 0 comme nombre d'octets lut.

(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.

**Solution:** Si le programme est interrompu, le fichier pour publier et accéder au serveur restera sur disque, rendant impossible le relancement du serveur avec le même nom tant que le fichier n'est pas effacé, comme la ligne 47 de `chtrmserv.c` ne sera pas exécutée avant la complétion du programme.
Voici le code révisé du serveur avec la gestion des signaux.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stddef.h>
4  #include <string.h>
5  #include <errno.h>
6  #include <unistd.h>
7  #include <limits.h>
8  #include <sys/socket.h>
9  #include <sys/un.h>
10 #include <signal.h>
11
12 #define MAXSIZE 255
13 #define MAXCLIENTS 4
14
15 char roomn[MAXSIZE+1];
16 int serv_sockfd;
17
```

```
18 static void sighdl(int sig) {
19   printf("Interrupted with signal %d\n", sig);
20   close(serv_sockfd);
21   unlink(roomn);
22   exit(0);
23 }
24
25 int main(int argc, char** argv) {
26   struct sigaction act;
27   struct sockaddr_un serv_addr, clnt_addr;
28   char messg[MAXSIZE+1];
29   int  clnt_sockfd, i, rc, rdb;
30
31   if(argc < 2) { fprintf(stderr, "usage> %s roomname\n", argv[0]); exit(1); }
32   strncpy(roomn, argv[1], MAXSIZE);
33
34   serv_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
35   if(serv_sockfd < 0) { perror("socket() failed"); exit(1); }
36
37   memset(&act, '\0', sizeof(act));
38   act.sa_handler = &sighdl;
39   rc = sigaction(SIGINT, &act, NULL);
40   if(rc < 0) { perror("sigaction() for SIGTINT failed"); exit(1); }
41   rc = sigaction(SIGTERM, &act, NULL);
42   if(rc < 0) { perror("sigaction() for SIGTERM failed"); exit(1); }
43
44   memset(&serv_addr, 0, sizeof(serv_addr));
45   serv_addr.sun_family = AF_UNIX;
46   strncpy(serv_addr.sun_path, roomn, sizeof(serv_addr.sun_path)-1);
47   rc = bind(serv_sockfd, (struct sockaddr*)&serv_addr, SUN_LEN(&serv_addr));
48   if(rc < 0) { perror("bind() failed"); exit(1); }
49
50   rc = listen(serv_sockfd, MAXCLIENTS);
51   if(rc < 0) { perror("listen() failed"); exit(1); }
52
53   for(i=0; i<INT_MAX; ++i) {
54     clnt_sockfd = accept(serv_sockfd, NULL, NULL);
55     if(clnt_sockfd < 0) { perror("accept() failed"); exit(1); }
56     rdb = 0;
57     for(rc=-1; (rc!=0) && (rdb<MAXSIZE); rdb+=rc) {
58       rc = read(clnt_sockfd, messg+rdb, MAXSIZE-rdb);
59       if(rc < 0) perror("read() from socket failed");
60     }
61     printf("%i: %s\n", i, messg);
62     close(clnt_sockfd);
63   }
64
65   close(serv_sockfd);
66   unlink(roomn);
67   return 0;
68 }
```

(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.

**Solution:**

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stddef.h>
4  #include <string.h>
5  #include <errno.h>
6  #include <unistd.h>
7  #include <limits.h>
8  #include <sys/socket.h>
9  #include <sys/un.h>
10 #include <pthread.h>
11
12 #define MAXSIZE 255
13 #define MAXCLIENTS 4
14
15 pthread_mutex_t mutx, mutxp;
16 pthread_cond_t  condc, condp;
17 int sockfdc, i;
18
19 void* threadc(void *ptr) {
20   char messg[MAXSIZE+1];
21   int clnt_sockfd, rdb, rc;
22   while(1) {
23     pthread_mutex_lock(&mutx);
24     while(sockfdc == 0) pthread_cond_wait(&condc, &mutx);
25     clnt_sockfd = sockfdc;
26     sockfdc = 0;
27     pthread_cond_signal(&condp);
28     pthread_mutex_unlock(&mutx);
29     rdb = 0;
30     for(rc=-1; (rc!=0) && (rdb<MAXSIZE); rdb+=rc) {
31       rc = read(clnt_sockfd, messg+rdb, MAXSIZE-rdb);
32       if(rc < 0) perror("read() from socket failed");
33     }
34     pthread_mutex_lock(&mutxp);
35     printf("%i: %s\n", i++, messg);
36     pthread_mutex_unlock(&mutxp);
37     close(clnt_sockfd);
38   }
39 }
40
41 int main(int argc, char** argv) {
42   pthread_t workers[MAXCLIENTS];
43   struct sockaddr_un serv_addr, clnt_addr;
44   char roomn[MAXSIZE+1];
45   int  serv_sockfd, clnt_sockfd, rc, j;
46
47   if(argc < 2) { fprintf(stderr, "usage> %s roomname\n", argv[0]); exit(1); }
48   strncpy(roomn, argv[1], MAXSIZE);
49
50   serv_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
51   if(serv_sockfd < 0) { perror("socket() failed"); exit(1); }
52
53   memset(&serv_addr, 0, sizeof(serv_addr));
54   serv_addr.sun_family = AF_UNIX;
```

```
55    strncpy(serv_addr.sun_path, roomn, sizeof(serv_addr.sun_path)-1);
56    rc = bind(serv_sockfd, (struct sockaddr*)&serv_addr, SUN_LEN(&serv_addr));
57    if(rc < 0) { perror("bind() failed"); exit(1); }
58
59    rc = listen(serv_sockfd, MAXCLIENTS);
60    if(rc < 0) { perror("listen() failed"); exit(1); }
61
62    pthread_mutex_init(&mutx, NULL);
63    pthread_mutex_init(&mutxp, NULL);
64    pthread_cond_init(&condc, NULL);
65    pthread_cond_init(&condp, NULL);
66    i=0;
67    sockfdc=0;
68    for(j=0; j<MAXCLIENTS; ++j) {
69      rc = pthread_create(&(workers[j]), NULL, threadc, NULL);
70      if(rc < 0) perror("pthread_create() failed");
71    }
72
73    for(j=0; j<INT_MAX; ++j) {
74      clnt_sockfd = accept(serv_sockfd, NULL, NULL);
75      if(clnt_sockfd < 0) { perror("accept() failed"); exit(1); }
76      pthread_mutex_lock(&mutx);
77      sockfdc = clnt_sockfd;
78      pthread_cond_signal(&condc);
79      pthread_cond_wait(&condp, &mutx);
80      pthread_mutex_unlock(&mutx);
81    }
82
83    for(j=0; j<MAXCLIENTS; ++j) {
84      pthread_cancel(workers[j]);
85      pthread_join(workers[j], NULL);
86    }
87    pthread_cond_destroy(&condp);
88    pthread_cond_destroy(&condc);
89    pthread_mutex_destroy(&mutxp);
90    pthread_mutex_destroy(&mutx);
91
92    close(serv_sockfd);
93    unlink(roomn);
94    return 0;
95  }
```

# 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.

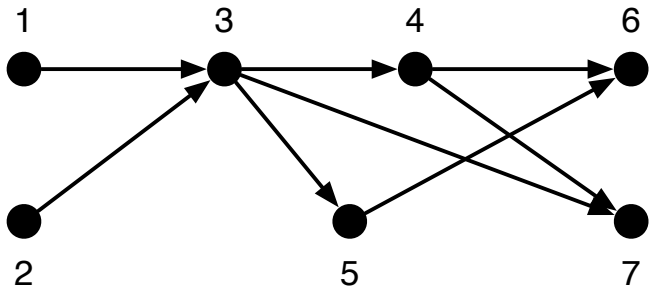
**Solution:**

| 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 (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.

**Solution:** La ligne de commande fera la liste détaillée des fichiers présents dans le répertoire `/etc` (`ls -l /etc`), la liste étant ensuite filtrée pour conserver seulement les lignes de la liste de fichier qui compter les caractères `conf` (`grep conf`) pour ensuite être affichée dans un programme interactif pour défiler le résultat (`less`). Trois processus seront ainsi lancés par cette ligne de commande.

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

**Solution:** Des reader-writer locks permettent des accès simultanés en lecture à certains espaces mémoire, permettant ainsi plus de concurrence dans l'exécution de programmes concourant, ce qui n'est pas possible avec 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`.

**Solution:** Avec l'ordonnanceur `SCHED_DEADLINE` une tâche cède la main par l'appel à la fonction `sched_yield`. L'ordonnanceur considère alors la tâche complétée pour le cycle courant, la tâche étant activée seulement au prochain cycle d'exécution. Si une tâche ne cède pas la main, elle va ainsi monopoliser un processeur durant tout son cycle, empêchant ainsi une exécution de toutes les tâches dans le respect des contraintes de temps réel.

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

**Solution:**

1. La patch PREEMPT_RT modifie la structure du noyau pour le rendre en bonne partie préemptible, limitant la latence dans le fonctionnement du système.
2. La patch fait également les changements aux primitives de synchronisation utilisées dans le noyau, pour utiliser des primitives gérant adéquatement le problème d'inversion de priorité.

(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.

**Solution:** L'approche *copy-on-write* permet d'éviter des duplications inutiles de pages de mémoire, en partageant un emplacement entre plusieurs processus tant que les accès sont faits uniquement en lecture. Lors d'une écriture, la page une copie spécifique de la mémoire est faite pour le processus ayant amorcé une étape d'écriture. Le délai induit par cette copie peut s'avérer important à l'échelle de la tâche, en plus d'être difficilement prévisible, produisant ainsi de scénarios de pires cas difficiles à ordonnancer.

(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.

**Solution:** Dans des systèmes temps réel, la durée d'une tâche correspond généralement au pire cas. Si la durée réelle comporte une importante variabilité, la durée utilisée pour l'ordonnancement sera le pire cas, ce qui risque de compliquer, voire rendre impossible l'ordonnancement de tâches respectant les contraintes de temps réel.

(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*).

**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.