Systèmes embarqués temps réel (GIF-3004)
Département de génie électrique et de génie informatique
Hiver 2019

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.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>

off_t dudir(char* filename) {              /* Calculer et afficher taille repertoire */
  DIR *dp;                                 /* Pour lire contenu de repertoire */
  struct stat filestat;                    /* Information sur fichier */
  off_t filesize;                          /* Mesure taille fichier */
  stat(filename, &filestat);               /* Pour obtenir taille fichier */
  filesize = filestat.st_size;             /* Obtenir taille fichier filename */
  dp = opendir(filename);                  /* Verifier si repertoire */
  if(dp != NULL) {                         /* NULL si n'est pas un repertoire */
    char childdir[256];                    /* Nom fichier enfant */
    char* fullpathchilddir;                /* Chemin complet vers fichier enfant */
    size_t fpcdszalloc;                    /* Taille alloue chemin complet */
    struct dirent *ep;                     /* Structure de lecture du repertoire */
    fullpathchilddir = NULL;               /* Nom complet du fichier enfant non alloue */
    fpcdszalloc = 0;                       /* Taille nom complet non alloue*/
    ep = readdir(dp);                      /* Entree sur premier fichier, NON REENTRANT */
    while(ep != NULL) {                    /* Tester si encore fichier a traiter */
      strncpy(childdir, ep->d_name, 256);  /* Copier nom fichier enfant */
      ep = readdir(dp);                    /* Aller a prochaine entree, NON REENTRANT */
      /* Ne pas traiter si repertoire courant (.) ou parent (..) */
      if((strcmp(childdir,".")!=0) && (strcmp(childdir,"..")!=0)) {
        const size_t fpcdsztarget = strlen(filename) + strlen(childdir) + 2;
        if(!fullpathchilddir) {            /* Si fullpathchilddir n'est pas alloue */
          fullpathchilddir = (char*)malloc(fpcdsztarget);
          fpcdszalloc = fpcdsztarget;
        } else if(fpcdszalloc<fpcdsztarget) { /* Si fullpathchilddir est trop petit */
          fullpathchilddir = (char*)realloc(fullpathchilddir, fpcdsztarget);
          fpcdszalloc = fpcdsztarget;
        }
        /* Creer nom complet repertoire enfant */
        strncpy(fullpathchilddir, filename, fpcdszalloc);
        if(strcmp(filename,"./")!=0) strncat(fullpathchilddir, "/", fpcdszalloc);
        strncat(fullpathchilddir, childdir, fpcdszalloc);
        /* Calculer taille repertoire enfant et ajouter a la somme */
        filesize += dudir(fullpathchilddir);
      }
    }
    if(fullpathchilddir != NULL) free(fullpathchilddir);
    closedir(dp);
  }
  printf("%llu\t%s\n", filesize, filename); /* Afficher taille et nom de fichier complet*/
  return filesize;                         /* Retourner taille complete repertoire */
}

int main(int argc, char** argv) {
  off_t totalsize = 0;
  if(argc == 1) totalsize = dudir(".");    /* Repertoire courant si aucun nom */
  else {
    for(int i=1; i<argc; ++i) {            /* Traiter tous les arguments recus */
      if(access(argv[i],F_OK) != -1) {     /* Tester si fichier existe*/
        totalsize += dudir(argv[i]);       /* Calculer taille et ajouter a somme */
      } else {
        fprintf(stderr, "File %s does not exist\n", argv[i]);
        return -1;
      }
    }
  }
  printf("%llu\n", totalsize);
  return 0;
}
```

Question 1 (35 points sur 100)

Soit le code de la page précédente, qui implémente un programme déterminant l'espace total occupé par les fichiers et répertoires donnés sur la ligne de commande, similairement au programme Unix `du`. Le programme calcule l'espace total occupé par tous les fichiers de l'arborescence des répertoires traités, par un appel récursif de la fonction `dudir`.

(15) (a) Donnez une nouvelle version de la fonction `main` qui fera une exécution parallèle multiprocessus du programme lorsque plusieurs noms de fichiers sont fournis sur la ligne de commande. Plus précisément, pour chaque fichier / répertoire spécifié sur la ligne de commande, un processus distinct doit être lancé en utilisant la fonction `fork` et utilisant des pipes pour la communication interprocessus. L'exécution doit être asynchrone, pour permettre le plein parallélisme dans l'exécution des processus. Ne vous préoccupez pas de l'ordre d'affichage dans la sortie standard (stdout).

Solution:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <dirent.h>
6  #include <sys/stat.h>
7  #include <sys/wait.h>
8
9  off_t dudir(char* filename) {           /* Calculer et afficher taille repertoire */
10   DIR *dp;                            /* Pour lire contenu de repertoire */
11   struct stat filestat;               /* Information sur fichier */
12   off_t filesize;                     /* Mesure taille fichier */
13   stat(filename, &filestat);          /* Pour obtenir taille fichier */
14   filesize = filestat.st_size;        /* Obtenir taille fichier filename */
15   dp = opendir(filename);             /* Verifier si repertoire */
16   if(dp != NULL) {                    /* NULL si n?est pas un repertoire */
17     char childdir[256];              /* Nom fichier enfant */
18     char* fullpathchilddir;          /* Chemin complet vers fichier enfant */
19     size_t fpcdszalloc;              /* Taille alloue chemin complet */
20     struct dirent *ep;               /* Structure de lecture du repertoire */
21     fullpathchilddir = NULL;         /* Nom complet du fichier enfant non alloue */
22     fpcdszalloc = 0;                 /* Taille nom complet non alloue*/
23     ep = readdir(dp);                /* Entree sur premier fichier, NON REENTRANT */
24     while(ep != NULL) {              /* Tester si encore fichier a traiter */
25       strncpy(childdir, ep->d_name, 256); /* Copier nom fichier enfant */
26       ep = readdir(dp);            /* Aller a prochaine entree, NON REENTRANT */
27       /* Ne pas traiter si repertoire courant (.) ou parent (..) */
28       if((strcmp(childdir,".")!=0) && (strcmp(childdir,"..")!=0)) {
29          const size_t fpcdsztarget = strlen(filename) + strlen(childdir) + 2;
30          if(!fullpathchilddir) { /* Si fullpathchilddir n?est pas alloue */
31            fullpathchilddir = (char*)malloc(fpcdsztarget);
32            fpcdszalloc = fpcdsztarget;
33          } else if(fpcdszalloc<fpcdsztarget) { /* Si fullpathchilddir trop petit */
34            fullpathchilddir = (char*)realloc(fullpathchilddir, fpcdsztarget);
35            fpcdszalloc = fpcdsztarget;
36          }
37          /* Creer nom complet repertoire enfant */
38          strncpy(fullpathchilddir, filename, fpcdszalloc);
39          if(strcmp(filename,"./")!=0) strncat(fullpathchilddir, "/", fpcdszalloc);
40          strncat(fullpathchilddir, childdir, fpcdszalloc);
41          /* Calculer taille repertoire enfant et ajouter a la somme */
42          filesize += dudir(fullpathchilddir);
43       }
44     }
```

```
45      if(fullpathchilddir != NULL) free(fullpathchilddir);
46      closedir(dp);
47    }
48    printf("%llu\t%s\n", filesize, filename); /* Afficher taille et nom de fichier */
49    return filesize; /* Retourner taille complete repertoire */
50  }
51
52  int main(int argc, char** argv) {
53    off_t totalsize = 0;
54    if(argc == 1) totalsize = dudir(".");   /* Repertoire courant si aucun nom */
55    else {
56      pid_t* pids;
57      int*   child_pipe;
58      pids = (pid_t*)malloc((argc-1)*sizeof(pid_t));
59      child_pipe = (int*)malloc((argc-1)*sizeof(int));
60      for(int i=1; i<argc; ++i) {         /* Traiter tous les arguments recus */
61        if(access(argv[i],F_OK) != -1) {    /* Tester si fichier existe*/
62          int index = i-1;
63          int fd[2];
64          /* Initialise les pipes */
65          if(pipe(fd)==-1) {
66            fprintf(stderr, "Pipe Failed" );
67            return 1;
68          }
69          /* Fork le processus*/
70          pids[index] = fork();
71          if(pids[index] < 0) {
72            fprintf(stderr, "fork Failed" );
73            return 1;
74          }
75          /* Parent */
76          else if (pids[index] > 0) {
77            child_pipe[index] = fd[0];
78            close(fd[1]);  /* Fermer le côté écriture */
79          }
80          /* Enfant */
81          else {
82            off_t size = 0;
83            close(fd[0]); /* Fermer le côté lecture*/
84            size = dudir(argv[i]); /* Calculer taille et ajouter a somme */
85            write(fd[1], &size, sizeof(off_t));
86            close(fd[1]);
87            exit(0);
88          }
89        } else {
90          fprintf(stderr, "File %s does not exist\n", argv[i]);
91          return -1;
92        }
93      }
94      for(int i=0; i<argc-1; ++i) {
95        waitpid(pids[i], NULL, 0);                  /* Attendre processus enfants */
96        off_t size = 0;
97        read(child_pipe[i], &size, sizeof(off_t));  /* Lecture des données */
98        close(child_pipe[i]);                       /* Fermeture des pipes*/
99        totalsize += size;
100     }
101     free(child_pipe);
102     free(pids);
103   }
104   printf("%llu\n", totalsize);
105   return 0;
106 }
```

(20) (b) Donnez maintenant une nouvelle version du programme qui fait une exécution parallèle multithreadée. Similairement à la sous-question précédente, un thread est lancé pour chaque fichier / répertoire spécifié sur la ligne de commande. Notez cependant que la fonction `readdir` (lignes 22 et 25) n'est pas réentrante, la valeur de type `struct dirent*` retournée par la fonction `readdir` est une valeur globale modifiée à chaque appel de la fonction. Ceci implique que les séquences d'appels et accès aux valeurs obtenues doivent être protégées par des primitives de synchronisation, pour assurer l'intégrité des valeurs obtenues. (N'utilisez pas la version réentrante de la fonction, `readdir_r`.)

Solution:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <dirent.h>
6  #include <sys/stat.h>
7  #include <sys/wait.h>
8  #include <pthread.h>
9
10 pthread_mutex_t mutex;
11
12 off_t dudir(char* filename) {          /* Calculer et afficher taille repertoire */
13   DIR *dp;                            /* Pour lire contenu de repertoire */
14   struct stat filestat;               /* Information sur fichier */
15   off_t filesize;                     /* Mesure taille fichier */
16   stat(filename, &filestat);          /* Pour obtenir taille fichier */
17   filesize = filestat.st_size;        /* Obtenir taille fichier filename */
18   dp = opendir(filename);             /* Verifier si repertoire */
19   if(dp != NULL) {                    /* NULL si n?est pas un repertoire */
20     char childdir[256];               /* Nom fichier enfant */
21     char* fullpathchilddir;           /* Chemin complet vers fichier enfant */
22     size_t fpcdszalloc;               /* Taille alloue chemin complet */
23     struct dirent *ep;                /* Structure de lecture du repertoire */
24     fullpathchilddir = NULL;          /* Nom complet du fichier enfant non alloue */
25     fpcdszalloc = 0;                  /* Taille nom complet non alloue*/
26     char* d_name = NULL;
27     pthread_mutex_lock(&mutex);
28     ep = readdir(dp); /* Entree sur premier fichier, NON REENTRANT */
29     if(ep != NULL) strncpy(childdir, ep->d_name, 256);
30     pthread_mutex_unlock(&mutex);
31     while(ep != NULL) { /* Tester si encore fichier a traiter */
32       /* Ne pas traiter si repertoire courant (.) ou parent (..) */
33       if((strcmp(childdir,".")!=0) && (strcmp(childdir,"..")!=0)) {
34         const size_t fpcdsztarget = strlen(filename) + strlen(childdir) + 2;
35         if(!fullpathchilddir) { /* Si fullpathchilddir n?est pas alloue */
36           fullpathchilddir = (char*)malloc(fpcdsztarget);
37           fpcdszalloc = fpcdsztarget;
38         } else if(fpcdszalloc<fpcdsztarget) { /* Si fullpathchilddir est trop petit */
39           fullpathchilddir = (char*)realloc(fullpathchilddir, fpcdsztarget);
40           fpcdszalloc = fpcdsztarget;
41         }
42         /* Creer nom complet repertoire enfant */
43         strncpy(fullpathchilddir, filename, fpcdszalloc);
44         if(strcmp(filename,"./")!=0) strncat(fullpathchilddir, "/", fpcdszalloc);
45         strncat(fullpathchilddir, childdir, fpcdszalloc);
46         /* Calculer taille repertoire enfant et ajouter a la somme */
47         filesize += dudir(fullpathchilddir);
48       }
49       pthread_mutex_lock(&mutex);
50       ep = readdir(dp); /* Aller a prochaine entree, NON REENTRANT */
51       if(ep != NULL) strncpy(childdir, ep->d_name, 256);
52       pthread_mutex_unlock(&mutex);
```

```
53      }
54      if(fullpathchilddir != NULL) free(fullpathchilddir);
55      closedir(dp);
56    }
57    printf("%llu\t%s\n", filesize, filename); /* Afficher taille et nom de fichier */
58    return filesize; /* Retourner taille complete repertoire */
59  }
60
61  struct arguments {
62    char* file_name;
63    off_t size;
64  };
65
66  void* worker(void * ptr){
67    struct arguments* args = (struct arguments*)ptr;
68    args->size = dudir(args->file_name);
69    return NULL;
70  }
71
72  int main(int argc, char** argv) {
73    pthread_mutex_init(&mutex, NULL); /* Initialiser mutex */
74    off_t totalsize = 0;
75    if(argc == 1) totalsize = dudir(".");   /* Repertoire courant si aucun nom */
76    else {
77      pthread_t* pts = NULL;
78      struct arguments* args = NULL;
79      pts = (pthread_t*)malloc((argc-1)*sizeof(pthread_t));
80      args = (struct arguments*)malloc((argc-1)*sizeof(struct arguments));
81      for(int i=1; i<argc; ++i) {           /* Traiter tous les arguments recus */
82        if(access(argv[i],F_OK) != -1) {    /* Tester si fichier existe */
83          args[i-1].file_name = argv[i];
84          pthread_create(&pts[i-1], NULL, worker, (void*)(args+i-1));
85        } else {
86          fprintf(stderr, "File %s does not exist\n", argv[i]);
87          return -1;
88        }
89      }
90      for(int i=0; i<argc-1; ++i) {
91        pthread_join(pts[i], NULL);
92        totalsize += args[i].size;
93      }
94      free(pts);
95      free(args);
96    }
97    printf("%llu\n", totalsize);
98    pthread_mutex_destroy(&mutex); /* Détruire mutex */
99    return 0;
100 }
```

Question 2 (25 points sur 100)

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

Tâche	1	2	3	4	5	6	7
Temps de lancement (r_i)	1	1	3	7	8	10	10
Durée d'exécution (e_i)	2	4	3	2	4	5	3
Échéance (d_i)	5	7	10	18	16	23	20

(5) (a) Effectuer un ordonnancement de type *round robin* sur un processeur, en démarrant l'exécution avec la tâche 1. Indiquez également si les contraintes d'exécution temps réel des tâches sont respectées.

Solution:

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

Les contraintes temps réel ne sont pas respectées pour les tâches 2 et 5, 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	11	12
Tâche	1	1	2	2	2	2	3	3	3	5	5	5

Temps	13	14	15	16	17	18	19	20	21	22	23	
Tâche	5	4	4	7	7	7	6	6	6	6	6	

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 mais **sans migration** possible entre les processeurs d'une tâche lorsqu'elle est déjà démarrée (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 globaux de traitement, en justifiant votre réponse.

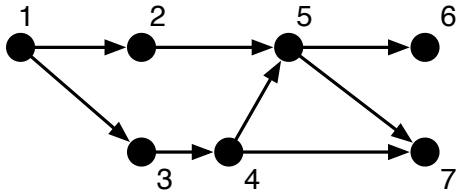
Solution:

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

Temps	10	11	12	13	14
Tâche proc 1	5	5	7	7	7
Tâche proc 2	6	6	6	6	6

L'ordonnancement est optimal, le temps de fin de l'exécution (cycle 14) correspond à la valeur maximale du temps de lancement plus durée d'exécution parmi toutes des tâches, soit la tâche 6 ($r_6 + e_6 - 1 = 14$).

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



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)	1	**3**	3	7	**9**	**13**	**13**
Durée d'exécution (e_i)	2	4	3	2	4	5	3
Échéance effective (d'_i)	**3**	7	10	**12**	16	23	20

(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	1	1	2	2	2	2	3	3	3	4
Temps	11	12	13	14	15	16	17	18	19	20
Tâche	4	5	5	5	5	7	7	7	6	6
Temps	21	22	23							
Tâche	6	6	6							

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 (40 points sur 100)

Répondez aussi brièvement et clairement que possible aux questions suivantes.

(4) (a) Il a été dit dans le cours qu'il y a deux approches de contrôle dans les systèmes temps réels, soit le contrôle activé par le temps et le contrôle activé par les événements. Indiquez quelle approche prédomine généralement dans les systèmes temps réels durs, en justifiant votre réponse.

Solution: Dans des systèmes temps réels durs, un contrôle activé par le temps prédomine généralement. En effet, ce type de contrôle peut être bien caractérisé durant la conception du système et limite les variations dans les tâches à effectuer comparativement au contrôle activé par les événements.

(4) (b) 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) (c) Indiquez ce que va produire la commande `man man` dans un terminal Unix.

Solution: La commande `man man` va afficher le manuel Unix du programme `man`.

(4) (d) Indiquez ce qu'il se produit dans Unix lorsqu'un processus parent se termine alors que certains de ses processus enfants sont toujours en cours d'exécution.

Solution: Les processus enfants en cours d'exécution seront adoptés par le processus `init`, qui lira leur statut aussitôt qu'ils vont se terminer.

(4) (e) Expliquez l'avantage des reader-writer locks comparativement aux mutex.

Solution: Avec des reader-writer locks, on peut permettre plusieurs accès simultanés en lecture aux données, contrairement aux mutex ou les accès sont mutuellement exclusifs. Ceci peut dont éviter certains bloquages inutiles. Cependant, les accès en écriture (incluant écriture-lecture) restent mutuellement exclusifs.

(4) (f) Expliquez les différences principales entre les pipes et les sockets Unix.

Solution: Les pipes permettent des échanges entre des processus ayant une filiation, par exemple un processus parent et un enfant, ou deux processus partageant un même parent, comme les pipes sont créés et partagés juste avant qu'un `fork` soit effectué. Les sockets Unix permettent de mettre en relation des processus qui n'ont pas nécessairement de filiation, en utilisant un nom précis pour établir la connexion. De plus, les pipes sont des canaux de communication unidirectionnels, alors que les socket permettent des échanges duplex (bidirectionnels).

(4) (g) Expliquez pourquoi il est important de faire l'allocation dynamique de la mémoire (ex. avec fonction `malloc`) dans des programmes temps réel lors du lancement de ceux-ci, avant d'atteindre les sections critiques.

Solution: Un appel à la fonction `malloc` peut impliquer de délais importants et pas prévisibles à l'avance, comme l'allocation dynamique peut dépendre de l'utilisation globale de la mémoire dans le système. Comme on veut éviter d'introduire inutilement des éléments de variabilités dans les sections contraintes de programmes temps réels.

(4) (h) Expliquez les rôles respectifs des fonctions `shm_open` et `mmap` pour créer un espace mémoire partagé entre deux processus,

Solution: La commande `shm_open` permet de créer et d'ouvrir un espace mémoire accessible selon un nom donné, accessible avec les routines habituelles de lecture et écritures à des fichiers. La commande `mmap` permet de faire la correspondance avec une adresse mémoire, permettant des accès mémoire directs.

(4) (i) Expliquez comment l'ordonnanceur CFS (*Completely Fair Scheduler*) fait les ajustements pour permettre à ce que chaque processus obtiennent leur « juste part » du processeur.

Solution: L'ordonnanceur CFS calcule d'abord la part que chaque processus doit avoir d'un processeur, selon sa priorité et la priorité des autres processus actifs. Ensuite, la durée du quantum d'exécution du processus est recalculée à chaque fois, afin que la part de temps du processus y soit assignée.

(4) (j) On dit souvent qu'Unix est un système portable au niveau du code. Expliquez en quoi POSIX a contribué à permettre cette portabilité.

Solution: POSIX est une interface de programmation (API) en langage C standardisé. Ainsi, les fonctionnalités principales des systèmes d'exploitation Unix définis dans un standard portable faisant en sorte que les programmes en C se limitant à l'utilisation de cette interface de programmation peuvent être compilés sur les machines très différentes, dans le mesure que le système d'exploitation utilisé implémente le standard POSIX.