Systèmes embarqués temps réel (GIF-3004)
Département de génie électrique et de génie informatique
Hiver 2018

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 (30 points sur 100)

Soit le code suivant, implémentant un programme comptant le nombre d'occurrences d'un mot dans un fichier.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h>
5
6  unsigned int count_word(char* doc, size_t doc_size, char *word, size_t word_size)
7  {
8      unsigned int wc = 0;
9      for(int i=0; i<((doc_size-word_size)+1); ++i) {
10         unsigned int min_size = ((doc_size-i) < word_size) ? (doc_size-i) : word_size;
11         int cmp = strncmp(doc+i, word, min_size);
12         if(cmp == 0) ++wc;
13     }
14     return wc;
15 }
16
17 int main(int argc, char** argv)
18 {
19     size_t buffer_size = 1024, doc_size = 0, read_size = 1;
20     char *document = 0;
21     unsigned int wc = 0;
22
23     if(argc != 2) {
24         fprintf(stderr, "usage> %s word_to_count", argv[0]);
25         return 1;
26     }
27
28     document = (char*)malloc(buffer_size);
29     while(read_size > 0) {
30         if(buffer_size == doc_size) {
31             buffer_size *= 2;
32             document = (char*)realloc(document, buffer_size);
33         }
34         read_size = read(STDIN_FILENO, document+doc_size, buffer_size-doc_size);
35         doc_size += read_size;
36     }
37
38     wc = count_word(document, doc_size, argv[1], strlen(argv[1]));
39
40     printf("%i\n", wc);
41     free(document);
42     return 0;
43 }
```

Nous voulons faire différentes versions parallèles de ce code, fonctionnant sur des architectures multicœurs, selon une approche de type *map-reduce*, avec séparation de la tâche entre plusieurs fils ou processus (partie *map*) et intégration des résultats dans le fil ou processus parent (partie *reduce*)

(15) (a) En vous inspirant de ce code, faites une version multiprocessus de ce code C, pouvant fonctionner sur plusieurs cœurs. Cette implémentation doit suivre les étapes suivantes :

1. Le processus parent lit le fichier à traiter de l'entrée standard (STDIN) ;
2. Le processus parent lance N processus enfants de traitement (N peut être donné en dur dans le code) ;

3. Le fichier est divisé en N parties approximativement égales, chaque partie étant assignée à un processus enfant distinct ;
4. Chaque processus enfant compte le nombre de mots d'intérêt de sa partie ;
5. Chaque processus enfant retourne le résultat du compte au processus parent dans un pipe UNIX ;
6. Le processus parent fait la somme des comptes reçus des processus enfants et l'affiche à la sortie standard (STDOUT).

Pour les besoins de l'examen, ne vous souciez pas du cas de mots d'intérêt présents dans le document et qui seraient coupés entre deux parties traitées par deux processus enfants distincts.

Solution:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h>
5
6  #define N 4
7
8  unsigned int count_word(char* doc, size_t doc_size, char *word, size_t word_size)
9  {
10     unsigned int wc = 0;
11     for(int i=0; i<((doc_size-word_size)+1); ++i) {
12         unsigned int min_size = ((doc_size-i) < word_size) ? (doc_size-i) : word_size;
13         int cmp = strncmp(doc+i, word, min_size);
14         if(cmp == 0) ++wc;
15     }
16     return wc;
17 }
18
19 int main(int argc, char* argv[])
20 {
21     size_t buffer_size = 1024, doc_size = 0, read_size = 1, chunk_size = 0;
22     char *document = 0;
23     unsigned int wc = 0;
24
25     if(argc != 2) {
26         fprintf(stderr, "usage> %s word_to_count", argv[0]);
27         return 1;
28     }
29
30     document = (char*)malloc(buffer_size);
31     while(read_size > 0) {
32         if(buffer_size == doc_size) {
33             buffer_size *= 2;
34             document = (char*)realloc(document, buffer_size);
35         }
36         read_size = read(STDIN_FILENO, document+doc_size, buffer_size-doc_size);
37         doc_size += read_size;
38     }
39
40     pid_t pids[N];
41     int child_pipe[N];
42
43     for(int i=0; i<N; ++i) {
44         int fd[2];
45         if(pipe(fd) < 0) {
46             perror("Erreur avec pipe() fd");
47             return -1;
```

```
48          }
49          pids[i] = fork();
50          if(pids[i] < 0) {
51              perror("Erreur fork");
52              return -1;
53          }
54          else if(pids[i] == 0) {
55              chunk_size = doc_size/N;
56              if(i == N-1) chunk_size += (doc_size%N);
57              close(fd[0]);
58              size_t document_offset = doc_size/N * i;
59              wc = count_word(document + document_offset,
60                              chunk_size, argv[1], strlen(argv[1]));
61              write(fd[1], &wc, sizeof(unsigned int));
62              close(fd[1]);
63              exit(0);
64          }
65          else {
66              child_pipe[i] = fd[0];
67              close(fd[1]);
68          }
69      }
70
71      for(int i = 0; i<N; ++i) {
72          unsigned int child_wc = 0;
73          waitpid(pids[i], NULL, 0);
74          read(child_pipe[i], &child_wc, sizeof(unsigned int));
75          close(child_pipe[i]);
76          wc += child_wc;
77      }
78
79      printf("%i\n", wc);
80      free(document);
81      return 0;
82  }
```

(10) (b) Faites maintenant une version de se programme en utilisant N threads POSIX effectuant les traitements en parallèle, avec un espace mémoire partagé entre ceux-ci pour échanger l'information.

Solution:

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <pthread.h>
6
7   #define N 4
8
9   unsigned int count_word(char* doc, size_t doc_size, char *word, size_t word_size)
10  {
11      unsigned int wc = 0;
12      for(int i=0; i<((doc_size-word_size)+1); ++i) {
13          unsigned int min_size = ((doc_size-i) < word_size) ? (doc_size-i) : word_size;
14          int cmp = strncmp(doc+i, word, min_size);
15          if(cmp == 0) ++wc;
16      }
17      return wc;
18  }
19
20  struct arguments {
21      char *document;
22      size_t doc_size;
```

```
23      char *word;
24      int wc;
25  };
26
27  void* worker(void *ptr) {
28      struct arguments *args = (struct arguments*)ptr;
29      args->wc = count_word(args->document,
30                            args->doc_size,
31                            args->word,
32                            strlen(args->word));
33      return NULL;
34  }
35
36  int main(int argc, char* argv[])
37  {
38      size_t buffer_size = 1024, doc_size = 0, read_size = 1;
39      char *document = 0;
40      unsigned int wc = 0;
41
42      if(argc != 2) {
43          fprintf(stderr, "usage> %s word_to_count", argv[0]);
44          return 1;
45      }
46
47      document = (char*)malloc(buffer_size);
48      while(read_size > 0) {
49          if(buffer_size == doc_size) {
50              buffer_size *= 2;
51              document = (char*)realloc(document, buffer_size);
52          }
53          read_size = read(STDIN_FILENO, document+doc_size, buffer_size-doc_size);
54          doc_size += read_size;
55      }
56
57      pthread_t pts[N];
58      struct arguments args[N];
59
60      for(int i = 0; i<N; ++i) {
61          args[i].document = document + ((doc_size/N) * i);
62          args[i].doc_size = doc_size/N;
63          if(i == (N-1)) args[i].doc_size += (doc_size % N);
64          args[i].word = argv[1];
65          pthread_create(&pts[i], NULL, worker, (void *)(args+i));
66      }
67
68      for(int i = 0; i < N; ++i) {
69          pthread_join(pts[i], NULL);
70          wc += args[i].wc;
71      }
72
73      printf("%i\n", wc);
74      free(document);
75      return 0;
76  }
```

(5) (c) Faites maintenant une troisième version de se programme, qui est multiprocessus, comme à la sous-question (a), mais où l'échange d'information entre les processus est fait via un espace mémoire partagé, comme à la sous-question (b).

Solution:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
```

```
4  #include <unistd.h>
5  #include <sys/mman.h>
6
7  #define N 4
8
9  unsigned int count_word(char* doc, size_t doc_size, char *word, size_t word_size)
10 {
11     unsigned int wc = 0;
12     for(int i=0; i<((doc_size-word_size)+1); ++i) {
13         unsigned int min_size = ((doc_size-i) < word_size) ? (doc_size-i) : word_size;
14         int cmp = strncmp(doc+i, word, min_size);
15         if(cmp == 0) ++wc;
16     }
17     return wc;
18 }
19
20 int main(int argc, char* argv[])
21 {
22     size_t buffer_size = 1024, doc_size = 0, read_size = 1, chunk_size = 0;
23     char *document = 0;
24     unsigned int wc = 0;
25
26     if(argc != 2) {
27         fprintf(stderr, "usage> %s word_to_count", argv[0]);
28         return 1;
29     }
30
31     document = (char*)malloc(buffer_size);
32
33     while(read_size > 0) {
34         if(buffer_size == doc_size) {
35             buffer_size *= 2;
36             document = (char*)realloc(document, buffer_size);
37         }
38         read_size = read(STDIN_FILENO, document+doc_size, buffer_size-doc_size);
39         doc_size += read_size;
40     }
41
42     int* process_wc = mmap(NULL, sizeof(int)*N,
43                            PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
44     pid_t pids[N];
45
46     for (int i = 0; i < N; ++i) {
47         pids[i] = fork();
48         if(pids[i] < 0) {
49             perror("Fork error");
50             return -1;
51         }
52         else if(pids[i] == 0) {
53             chunk_size = doc_size/N;
54             if(i == N-1) chunk_size += doc_size%N;
55             size_t document_offset = doc_size/N * i;
56             process_wc[i] = count_word(document + document_offset,
57                                        chunk_size, argv[1], strlen(argv[1]));
58             exit(0);
59         }
60     }
61
62     for(int i=0; i<N; ++i) {
63         waitpid(pids[i], NULL, 0);
64         wc += process_wc[i];
65     }
66
67     printf("%i\n", wc);
68     munmap(process_wc, sizeof(int)*N);
```

```
69      free(document);
70      return 0;
71  }
```

Question 2 (30 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
Temps de lancement (r_i)	1	1	6	9	2	5
Durée d'exécution (e_i)	3	4	2	1	3	7
Échéance (d_i)	5	9	13	12	17	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	1,5
3	1	5,2
4	5	2,1
5	2	1,5,6
6	**1**	5,6,2,3
7	5	6,2,3
8	6	2,3,5
9	2	3,5,6,4
10	3	5,6,4,2
11	**5**	6,4,2,3
12	6	4,2,3
13	**4**	2,3,6
14	**2**	3,6
15	**3**	6
16	6	
17	6	
18	6	
19	6	
20	**6**	

Les contraintes temps réel ne sont pas respectées pour les tâches 1, 2, 3 et 4, comme leurs temps de terminaison ne respectent pas leurs échéances respectives.

(6) (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	1	1	1	2	2	2	2	3	4	3
Temps	11	12	13	14	15	16	17	18	19	20
Tâche	5	5	5	6	6	6	6	6	6	6

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

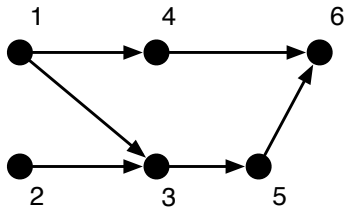
(6) (c) Donnez maintenant l'ordonnancement de ces tâches pour exploiter **deux** processeurs, toujours en utilisant l'algorithme EDF, avec préemption et en supposant que les tâches peuvent migrer entre les processeurs même si elles sont déjà démarrées (système dynamique). À la lumière de cet ordonnancement, 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	11	12
Tâche proc 1	1	1	1	5	5	5	6	6	6	6	6	6
Tâche proc 2	2	2	2	2	6	3	3		4			

Il aurait été possible d'être plus efficace en assignant la tâche 3 au processeur 1 et préemptant ainsi la tâche 5 aux temps 6 et 7. Ceci aurait permis de compléter l'exécution des tâches au temps 11.

(7) (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
Temps de lancement effectif (r'_i)	1	1	6	9	**8**	**11**
Durée d'exécution (e_i)	3	4	2	1	3	7
Échéance effective (d'_i)	5	**8**	**10**	12	**13**	20

(6) (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	1	2	2	2	2	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

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) 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) (b) Indiquez pourquoi dit-on qu'il est difficile d'implémenter des systèmes réels durs avec des systèmes embarqués basés Linux.

Solution: Linux est à la base un système d'exploitation général offrant de nombreuses fonctionnalités, mais qui n'est pas conçu spécifiquement pour faire du temps réel. Même avec des optimisations comme la patch `PREEMPT_RT`, les délais dans Linux peuvent être assez importants, rendant difficile le respect strict de certaines contraintes temps réel.

(4) (c) Expliquez ce qui fait qu'UNIX est réputé portable au niveau du code.

Solution: UNIX comprend une interface système standardisée, nommé POSIX, offrant un ensemble de fonctionnalité sous la forme de fonctions C, disponibles sur les différentes variantes modernes du système. Ainsi, un programme respectant cette interface est portable sur les différentes implémentations au niveau du code, la compilation de ce code permettant de produire des binaires fonctionnels sur différentes architectures.

(4) (d) Indiquez à quoi correspond l'opération effectuée par la ligne de commande suivante dans un shell UNIX : `ls | grep txt | sort`

Solution: Cette commande va lister le nom des fichiers présents dans le répertoire courant (commande `ls`), dont on va filtrer les fichiers contenant uniquement les caractères `txt` dans leur nom (commande `grep txt`), dont les résultats retenus seront triés en ordre alphabétique (commande `sort`).

(4) (e) Expliquez l'utilité du signal `SIGCHLD`.

Solution: Le signal `SIGCHLD` est lancé au processus parent lorsque l'un de ses processus enfants a terminé. Ainsi le programme du processus parent peut faire un `wait` pour compléter la terminaison du processus enfant et libérer les ressources du système d'exploitation qui y étaient assignées.

(4) (f) Expliquez les principales différences entre un programme multiprocessus et un programme multithreadé sous UNIX.

Solution: Dans un programme multiprocessus, les différents processus ne partagent généralement pas d'espace mémoire et gèrent leurs propres ressources (fichiers ouverts, traitement des signaux, etc.). De plus, la communication entre les processus se font généralement par des mécanismes tels que les pipes ou les sockets. Dans des programmes multithreadés, la mémoire et les ressources sont partagées entre les différents fils d'exécution. En général, les fils échangent de l'information entre eux via des espaces mémoires partagés, en prenant soin d'utiliser des primitives de synchronisation (ex. mutex, conditions, sémaphores) pour gérer les accès simultanés.

(4) (g) Expliquez en quoi la stratégie d'ordonnancement pour un poste de travail avec interface graphique interactif doit différer de l'approche à suivre pour des nœuds de traitement d'un supercalculateur.

Solution: Sur un poste de travail avec interface interactif, le quantum du système doit être assez court pour permettre une réponse rapide aux requêtes de l'utilisateur, à l'intérieur de quelques centaines de millisecondes, afin que celui-ci ne perçoive pas de délais incommodants. Avec des supercalculateurs, la performance importante, les nœuds n'étant pas utilisés de façon interactive. Ainsi, les délais de réponse peuvent être plus longs afin de minimiser l'effet des changements de contexte et donc maximiser l'utilisation des ressources.

(4) (h) Sous UNIX, la priorité de base est donnée par le nice. Donnez la valeur minimale et maximale du nice, en précisant celle où la priorité est la plus élevée, ainsi que la valeur par défaut pour de nouveaux processus.

Solution: La valeur du nice varient entre -20 (plus haute priorité) et +19 (priorité la plus basse). La valeur par defaut du nice est hérité du processus parent. Lors d'une connexion à une session, la priorité par défaut est de 0.

(4) (i) 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 un étape d'écriture. Le délais 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) (j) Expliquez le problème d'inversion de priorité.

Solution: Le problème d'inversion de priorité survient lorsqu'un processus H, d'assez haute priorité, attend sur une ressource utilisé par un processus B de plus basse priorité. Dans ce contexte, si plusieurs processus de moyenne priorité (entre H et B) sont en cours d'exécution, il y a de bonnes chances que le processus B soit continuellement en arrêt, bloquant ainsi le processus H qui est en attente sur la ressource.