TP Temps Réel

Jérôme Pouiller <j.pouiller@sysmic.org>

Polytech' Paris - Janvier 2011

Table des matières

1 Avant de commencer **2**
1.1 Documentation . . . 3
1.1.1 Références des pages de man . . . 3
1.1.2 Standards . . . 3
1.2 Erreurs . . . 3
1.3 Compilation . . . 3

2 Watch file **4**
2.1 Par scrutation . . . 4
2.2 Par notification . . . 4
2.3 Gestion des interruptions . . . 5
2.4 Gestion en multitâche . . . 5

3 Ordonnancement **5**
3.1 Rate Monotonic . . . 5
3.2 Earliest Deadline First . . . 6
3.3 Deffered server . . . 6
3.4 Sporadic server . . . 7
3.5 Comparaison des serveurs . . . 7
3.6 Partage de ressources - Inversions de priorités . . . 7
3.7 Partage de ressources - Dead locks . . . 8
3.8 Contraintes de précédence . . . 8

4 Protection des structures de données **9**

5 Tricky conditions **10**

6 Structures de données et mutex **12**

7 Protection de ressources **13**

8 Threads et mutex **14**

9 Tâches et sémaphores Xenomai **15**

10 Implémentation d'ordonnanceurs **15**

10.1 Algorithmes d'ordonnancement de tâches indépendantes 15

10.2 Algorithmes d'ordonnancement de tâches apériodiques 16

10.3 Ordonnancement et ressources partagées 16

11 Libtimer **16**

11.1 Version synchrone 19

11.2 Version asynchrone 19

12 Calculs et temps réel **19**

12.1 Introduction 19

12.2 Ecriture du Watchdog 19

12.3 Ecriture de l'ISR 20

A Instructions d'installation de la machine virtuelle Xenomai **21**

A.1 Mise en service de l'image 21

A.2 L'environnement 21

A.2.1 Se connecter 21

A.2.2 Sauver son travail 21

A.2.3 Travailler avec emacs, etc... 21

B Écriture des programmes Xenomai **21**

B.1 Documentation de Xenomai 21

B.2 Compilation 22

B.3 Vérouillage de la mémoire 22

B.4 A propos des noms des objets Xenomai 22

B.5 Gestion des Erreurs 22

B.6 Exécution 22

C Écriture des modules noyau **22**

C.1 Compilation 22

C.2 Exécution 23

C.3 Log Buffer 23

D Modalités de rendu **23**

E Modalités de rendu **24**

1 Avant de commencer

Cette section est purement informative. Il n'y a pas de questions à l'intérieur.

1.1 Documentation

1.1.1 Références des pages de man

Comme le veut l'usage, les références des pages de man sont donnés avec le numéro de section entre parenthèses. Ainsi, *wait(2)* signifie que vous pouvez accéder à la documentation avec la commande `man 2 wait`. Si vous omettez le numéro de section, `man` recherchera la première page portant le nom `wait` (ce qui fonctionnera dans 95% des cas). Le numéro de section vous donne aussi une indication sur le type de documentation que vous aller trouver. D'après *man(1)*, voici les différentes sections :

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions (e.g. /etc/passwd)
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

Vous pouvez trouver une version en ligne de la plupart des pages de man sur The Friendly Manual (`http://tfm.cz`).

Vous trouverez en particulier les pages relatives à l'API du noyau Linux qui ne sont pas présentes en local sur vos machine.

Prenez néanmoins garde aux éventuelles différences de versions d'API (très rare et quasiment exclusivement sur les pages relatives à l'ABI du noyau).

1.1.2 Standards

Vous subissez aujourd'hui tout le poids de l'histoire de la norme Posix (30 ans d'histoire de l'informatique). Ne soyez pas étonnés de trouver de multiples interfaces pour une même fonctionnalité. C'est particulièrement vrai sur les fonction relatives au temps.

Nous basons ces exercice sur la norme POSIX.1-2001 que vous pourrez trouver sur `http://www.unix-systems.org/version3/online.html`

Vous trouverez plus d'informations sur l'histoire des différentes normes existantes sur *standards(7)*.

1.2 Erreurs

Pour indiquer quelle erreur s'est produite, la plupart des fonctions Posix utilisent :
– soit leur valeur de retour
– soit une variable globale de type `int` nommée `errno`
Utilisez *strerror(3)* pour obtenir le message d'erreur équivalent au code de retour.

Exemple :

```
1 if (err = pthread_create(&tid, NULL, f, NULL))
   printf("Task create: %s", strerror(err));
  if (-1 == pause())
   printf("Pause: %m");
```

1.3 Compilation

Pour la plupart des exercices, vous aurez besoin de linker[1] avec `-lpthread` et `-lrt`.

Par ailleurs, nous vous conseillons fortement de compiler avec l'option `-Wall` de `gcc` et d'utiliser un système de Makefile.

1. C'est à dire la phase de compilation où vous liez tous vous fichiers `.o` ensemble

2 Watch file

2.1 Par scrutation

Question 2.1. *Implémenter un programme* `watch_files` *prenant en paramètre un nom de fichier. Par scrutation, votre programme détectera chaque fois que le fichier est créé ou supprimé. Votre programme devra afficher un message à chaque fois qu'il détecte une modification. Vous devrez être le plus réactif possible lors de la modification de status du fichier.*

Documentation utile: *stat(2), touch(1), rm(1)*

Question 2.2. *Sans efectuer de mesure précise, quel est l'ordre de grandeur du temps réponse de votre programme ?*

Question 2.3. *Que constatez-vous sur le taux d'utilisation du CPU ?*

Documentation utile: *top(1)*

Question 2.4. *Afin de rendre notre programme moins gourmand, ajouter une temporisation de 1s entre chaque scrutation de l'état du fichier.*

Documentation utile: *sleep(3)*

Question 2.5. *Que constatez-vous sur le taux d'utilisation du CPU ?*

Documentation utile: *top(1)*

Question 2.6. *Quel est maintenant le temps de réponse de votre programme ?*

Question 2.7. *Comment évolurait le taux d'utilisation du CPU et le temps de réponse si nous devions surveiller l'état de plusieurs millions de fichiers changeant rarement ?*

2.2 Par notification

Afin d'améliorer notre programme, nous allons utiliser un système piloyté par évènements. Ce mode de fonctionnement est assez proche d'un système piloté par interruption.

Question 2.8. *Modifiez votre programme afin d'être notifié des modifications sur votre fichier.*

Documentation utile: *inotify(7), inotify_init(2), inotify_add_watch(2), dirname(3), basename(3), strncpy(3), strcmp(3) read(2), close(2)*

Question 2.9. *Quel est maintenant le temps de réponse de votre programme ?*

Question 2.10. *Que constatez-vous sur le taux d'utilisation du CPU ?*

Documentation utile: *top(1)*

Question 2.11. *Comment évolurait le taux d'utilisation du CPU et le temps de réponse si nous devions surveiller l'état de plusieurs millions de fichiers changeant rarement ?*

2.3 Gestion des interruptions

Question 2.12. *Créez un programme* `watch_interrupt`. *Ce nouveau programme prendra lui aussi un nom de fichiers en paramètre. Lorsque vous recevez le signal USR1, affichez si le fichier existe ou non.*

Documentation utile: *signal(2), pause(3), stat(2), kill(1)*

Question 2.13. *Déplacez le traitement du signal USR1 dans la boucle principale du programme*

Documentation utile: *pause(3)*

2.4 Gestion en multitâche

Question 2.14. *Fusionnez les programmes* `watch_file` *et* `watch_interrupt` *en utilisant des processus. Vous nommerez ce programme* `watch_by_process`

Documentation utile: *fork(2)*

Question 2.15. *Fusionnez les programme* `watch_file` *et* `watch_interrupt` *en utilisant des threads. Vous nommerez ce programme* `watch_by_thread`

Documentation utile: *pthread_create(3)*

3 Ordonnancement

3.1 Rate Monotonic

Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	29	7	Fin de période
B	0	5	1	Fin de période
C	0	10	2	Fin de période

Question 3.1. *Calculez le taux d'utilisation du CPU avec l'algorithme Rate Monotonic. Le jeu de tâche est-il ordonnançable ?*

Question 3.2. *Dessinez les 30 premières unités de temps de l'ordonnancement généré par Rate Monotonic en version prémptive puis en non-préemptive. Que constatez-vous ?*

Modifions le jeu de tâches :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	30	6	Fin de période
B	0	5	3	Fin de période
C	0	10	2	Fin de période

Dans cet exemple, le jeu de tâche est dit *harmonique* car chaque période est multiple des autres périodes.

Question 3.3. *Calculez le taux d'utilisation du CPU avec l'algorithme Rate Monotonic. Le jeu de tâche est-il ordonnançable ?*

Question 3.4. *Dessinez les 30 premières unités de temps de l'ordonnancement généré par Rate Monotonic en version prémptive. Que constatez-vous ?*

Question 3.5. *Calculez les temps de réponse des différentes tâches. Que proposez-vous comme explication ?*

3.2 Earliest Deadline First

Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	12	5	Fin de période
B	0	6	2	Fin de période
C	0	24	5	Fin de période

Question 3.6. *Calculez le taux d'utilisation du CPU avec l'algorithme EDF. Le jeu de tâche est-il ordonnançable ?*

Question 3.7. *Déterminez le nombre d'unités de temps libres sur la période.*

Question 3.8. *Dessinez les 24 premières unités de temps de l'ordonnancement généré par EDF en version prémptive puis en non-préemptive.*

On considère le même jeu de tâches auquel on a ajouté deux tâches apériodiques :

Tâche	Arrivée	Période	Capacité	Echéance
D	7	-	2	9
E	12	-	3	21

Question 3.9. *Dessinez les 30 premières unités de temps de l'ordonnancement généré par EDF en version prémptive. Le jeux de tâches est-il ordonnançable par EDF ?*

3.3 Deffered server

Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	12	4	Fin de période
B	0	16	6	Fin de période
S	0	10	2	Fin de période

S est un serveur différé gérant les tâches asynchrones suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
1	2	-	1	Aucune
2	14	-	4	Aucune
3	28	-	2	Aucune

Question 3.10. *Dessinez les 30 premières unités de temps de l'ordonnancement généré par RM (en version préemptive).*

Question 3.11. *Mesurez les temps de réponse des différentes tâche.*

3.4 Sporadic server

Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	24	4	Fin de période
B	0	28	8	Fin de période
S	0	20	4	Fin de période

S est un serveur sporadique gérant les tâches asynchrones suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
1	4	-	3	Aucune
2	12	-	4	Aucune

Question 3.12. *Dessinez les 30 premières unités de temps de l'ordonnancement généré par RM (en version préemptive).*

Question 3.13. *Mesurez les temps de réponse des différentes tâche.*

3.5 Comparaison des serveurs

Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	7	2	Fin de période
S	0	6	4	Fin de période

S est un serveur différé gérant les tâches asynchrones suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
1	21	-	4	Aucune

Question 3.14. *Dessinez les 3 premières unités de temps de l'ordonnancement généré par RM (en version préemptive).*

Question 3.15. *Mesurez les temps de réponse des différentes tâche.*

3.6 Partage de ressources - Inversions de priorités

Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Prioritée
A	15	-	10	1
B	0	25	10	2
C	10	-	20	3

Question 3.16. *Dessinez les 50 premières unités de temps en utilisant un algorithme d'ordonnancement à priorité fixe (préemptif).*

Les tâches A et C accèdent maintenant à une ressource partagée protégée par un mutex.
- A acquière le mutex après 5 unité de temps et le relâche après 5 autres unités de temps
- B acquière le mutex après 1 unité de temps et le relâche après 15 autres unités de temps

Question 3.17. *Dessinez les 50 premières unités de temps en utilisant un algorithme d'ordonnancement à priorité fixe (préemptif).*

Question 3.18. *Protposez une modification du système de priorité afin de réduire la latence de A. Dessinez les 50 premières unités de temps en utilisant cette nouvelle stratégie d'ordonnancement.*

3.7 Partage de ressources - Dead locks

Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	10	5	Fin de période
B	0	25	8	Fin de période

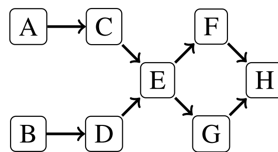
Les tâches A et B accèdent à deux ressources partagées protègée par de mutex.
- A acquière le mutex 1 après 1 unité de temps et le relâche après 3 autres unités de temps
- A acquière le mutex 2 après 3 unité de temps et le relâche après 2 autres unités de temps
- B acquière le mutex 1 après 6 unité de temps et le relâche après 2 autres unités de temps
- B acquière le mutex 2 après 3 unité de temps et le relâche après 5 autres unités de temps

Question 3.19. *Dessinez les 25 premières unités de temps en utilisant un ordonnancement EDF et un priority inheritance.*

Question 3.20. *Dessinez les 25 premières unités de temps en utilisant un ordonnancement EDF et un higtest lock (*aka *Immediate Ceiling Priority Protocol).*

Question 3.21. *Dessinez les 25 premières unités de temps en utilisant un ordonnancement EDF et un* original ceiling priority protocol *(*OCPP*).*

3.8 Contraintes de précédence



Soit les tâches suivantes :

Tâche	Arrivée	Période	Capacité	Echéance
A	0	-	3	12
B	0	-	1	10
C	0	-	3	20
D	0	-	1	15
E	0	-	5	25
F	0	-	4	37
G	0	-	1	20
H	0	-	1	40

Question 3.22. *Dessinez les 20 premières unités de temps en utilisant un ordonnancement EDF (préemptif) sans prendre en compte les contraintes de précédance. Est-ce conforme aux besoins de l'application ?*

Question 3.23. *Utilisez la méthode de Blazewicz/Chetto pour supprimer les contraintes de précédence.*

Question 3.24. *Calculez de nouveau le taux d'utilisation et redessinez l'ordonnancement ainsi généré sur sa période d'étude. Conclure sur l'ordonnançabilité de l'application.*

4 Protection des structures de données

Nous avons implémenté une structure de liste simplement chainée :

```
/*
 * Creation date: 2010-01-03 11:22:08+01:00
 * Licence: GPL
 * Main authors:
 *   - éôJrme Pouiller <jerome@sysmic.org>
 *
 * Sample use of mutexes.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct list_item_s {
   struct list_item_s *next;
   int value;
} list_item_t;

/* Add an element in list */
void list_add(list_item_t **list, int n) {
   list_item_t *item = malloc(sizeof(list_item_t));
   item->value = n;
   item->next = *list;
   *list = item;
}

/* Remove an element */
void list_del(list_item_t **list, int n) {
   list_item_t *i, *tmp;

   if ((*list)->value == n) {
      tmp = *list;
      *list = (*list)->next;
      free(tmp);
      return ;
   }
   for (i = *list; i->next; i = i->next)
      if (i->next->value == n) {
         tmp = i->next;
         i->next = tmp->next;
         free(tmp);
         return ;
      }
   printf("Item %d not found\n", n);
}
```

Nous avons écrit un scénario de test comme il se doit :

```
éô
#define N_ITERATIONS 110
void *task(void *arg) {
   list_item_t **l = arg;
   int i;

   list_add(l, 0);
   for (i = 1; i < N_ITERATIONS; i++) {
      list_add(l, i);
      list_del(l, i - 1);
      if (!(i % 100))
         printf("%08lX: %d\n", pthread_self(), i);
   }
   list_del(l, N_ITERATIONS - 1);

   return NULL;
}

#define NB_THREAD 2
int main(int argc, char **argv) {
   // Initialize an empty list
   list_item_t *l = NULL;
   pthread_t id[NB_THREAD];
   int i;

   for (i = 0; i < NB_THREAD; i++)
      pthread_create(&id[i], NULL, task, (void *) &l);

   for (i = 0; i < NB_THREAD; i++)
      pthread_join(id[i], NULL);

   if (l)
      printf("Error: list is not empty\n");
   return 0;
}
```

Néanmoins, le résultat n'est pas celui que nous attendions.

Question 4.1. *Observez le comportement avec différentes valeurs de* `N_ITERATIONS` *et* `NB_THREAD`. *Quel est le problème ?*

Remarque: La question peut se faire en environ 2 lignes.

Documentation utile: *pthread_create(3)*

Question 4.2. *Résolvez le problème. Vous prendrez soin de limiter la portée de vos sections critiques.*

Remarque: La question peut se faire en moins de 10 lignes.

Documentation utile: *pthread_mutex_lock(3), pthread_mutex_unlock(3)*

5 Tricky conditions

Nous avons un problème avec le code ci-dessous :

```
/*
 * Creation date: 2010-01-03 11:22:08+01:00
 * Licence: GPL
 * Main authors:
```

```
5   *  - éôJrme Pouiller <jerome@sysmic.org>
    *
    * Sample use of conditions.
    *
    */
   #include <stdio.h>
   #include <unistd.h>
   #include <stdlib.h>
   #include <pthread.h>

15 static pthread_cond_t c = PTHREAD_COND_INITIALIZER;
   static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
   static int count = 0;

   void *sender(void *arg) {
      int i;

      for (i = 0; i < 10; i++) {
         count++;
         printf("%08lX: Increment count: count=%d\n", pthread_self(), count);
25       if (count == 3) {
            pthread_cond_signal(&c);
            printf("%08lX: Sent signal. count=%d\n", pthread_self(), count);
         }
      }
      return NULL;
   }

   void *receiver(void *arg) {
      printf("%08lX: Begin waiting\n", pthread_self());
35    pthread_cond_wait(&c, &m);
      printf("%08lX: Received signal. count=%d\n", pthread_self(), count);
      return NULL;
   }

   int main(int argc, char **argv) {
      pthread_t id[2];

      pthread_mutex_lock(&m);
      pthread_create(&id[0], NULL, sender, NULL);
45    pthread_create(&id[1], NULL, receiver, NULL);

      pthread_join(id[0], NULL);
      pthread_join(id[1], NULL);
      pthread_mutex_unlock(&m);

      return 0;
   }
```

De temps en temps, le signal envoyé par la thread *sender* n'est pas recu par la thread *receiver*.

Question 5.1. *Exacerber le problème en plaçant une temporisation dans le code.*

Remarque: La question peut se faire en 2 lignes.

Documentation utile: *sleep(3)*

Question 5.2. *Utilisez le mutex* m *de manière appropriée pour résoudre le problème.*

Remarque: La question peut se faire en 4 lignes.

Documentation utile: *pthread_cond_wait(3),pthread_mutex_lock(3), pthread_mutex_unlock(3)*

Question 5.3. *Modifiez votre programme de façon à lancer 3 threads* receiver *et une thread* sender. *Puis, modifiez votre programme de manière à ce que la thread* sender *débloque correctement les 3 threads* receiver.

Remarque: Nous attirons votre attention sur ce paragraphe de la norme Posix :

> The effect of using more than one mutex for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable is undefined ; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when thewait returns.

Remarque: Pour résoudre élégamment ce problème, nous aurions aimé que `pthread_cond_wait` prenne en paramètre un sémaphore plutôt qu'un mutex. Nous allons donc devoir ajouter un semaphore à notre programme pour palier ce manque.

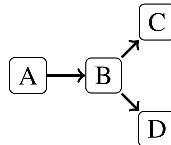
Remarque: La question peut se faire en environ 20 lignes.

Documentation utile: *pthread_cond_broadcast(3), pthread_cond_wait(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), sem_init(3), sem_wait(3), sem_post(3)*

---

6 Structures de données et mutex

Un réseau de chemin de fer relie les villes A, B, C et D. Deux trains circulent sur ce réseau :
- Le train circulant entre A, B et C
- Le train circulant entre A, B et D



On souhaite effectuer une simulation du fonctionnement de ce réseau de chemin de fer sachant que chaque segment de la ligne ne peut être utiliser que par un train à la fois. Néanmoins, un nombre illimité de trains peuvent se trouver dans la même gare.

Nous avons décidé de modéliser chaque train par une thread et le déplacement d'une ville à une autre par une temporisation d'1s pendant laquelle le segment de ligne est occupé.

Nous utilisons une structure de graphe pour représenter notre réseau ferré. Ce choix nous permettra de facilement étendre notre algorithme à de nouveaux cas.

Question 6.1. *Implémenter cette simulation.*

Remarque: Une manière commune de représenter un graphe de X nœuds en informatique est d'utiliser une matrice de $X \times X$ booléens. Chaque valeur représente l'existence (ou l'absence) d'une arête. Dans le cas d'un graphe non-orienté, la matrice sera symétrique :

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Remarque: La question peut se faire en environ 80 lignes.

Documentation utile: *pthread_create(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), sleep(3)*

---

7 Protection de ressources

Nous avons besoin d'envoyer 4 messages vers un serveur distant. Nous utilisons le code ci-dessous :

```c
#define SZ 7

void *task(void *arg) {
   char *buf = (char *) arg;

   send(buf, SZ);
   return NULL;
}

int main(int argc, char **argv) {
   pthread_t id[4];
   int i;
   char msgs[4][SZ] = { "123456", "ABCDEF", "abcdef", "[{()}]" };

   for (i = 0; i < 4; i++)
      pthread_create(&id[i], NULL, task, msgs[i]);

   for (i = 0; i < 4; i++)
      pthread_join(id[i], NULL);

   printf("\n");
   return 0;
}
```

Dans le cadre de notre étude, nous n'avons pas d'accès réel au serveur, nous avons donc écrit une fonction de simulation de l'envoie des donnée :

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void send_char(char data) {
   double time = (double) random() / RAND_MAX * 8;
   struct timespec t;
   t.tv_sec = time / 1000;
   t.tv_nsec = (int) (time * 1000000) % 1000000000;
   nanosleep(&t, NULL);
   write(1, &data, 1);
}

void send(char *data, size_t len) {
   unsigned i;

   for (i = 0; i < len; i++)
      send_char(data[i]);
}
```

Malheureusement, il semblerait qu'il y ait un problème car les messages reçu par le serveurs ne sont pas cohérents :

```
Aa[1b{cB2(deC3)}DE45Ff]6
```

Or, pour que le serveur puisse interpréter les données, il ne faut pas que les messages des *tâches* concurrentes soit enchevêtrés.

On propose d'emballer `send` dans une nouvelle fonction alternative permettant de régler le problème :

```c
int safe_send(char *data, size_t len);
```

Question 7.1. *Implémentez* `safe_send`.

Remarque: La question peut se faire en moins de 10 lignes.

Documentation utile: *pthread_mutex_lock(3), pthread_mutex_unlock(3)*

La fonction `safe_send` résout le problème de cohérence, mais cause des problèmes de latences dans la fonction `task`. Dans certains cas, celle-ci s'exécute beaucoup plus lentement qu'avant.

Question 7.2. *Écrivez un scénario de test permettant de mettre en évidence et de mesurer le phénomène de latence.*

Remarque: La question peut se faire en moins de 20 lignes.

Documentation utile: *clock_gettime(3), memset(3)*

Nous allons essayer de résoudre le problème de latence. Pour cela, nous allons créer une nouvelle *tâche* permettant d'envoyer les données de manière asynchrone.

Afin de transmettre à la *tâche* les données à envoyer, nous devons mettre en place une structure de données adéquate. Nous proposons d'utiliser une file (FIFO) sous la forme d'un *buffer circulaire*.

Enfin, notre *buffer circulaire* nécessitera un mécanisme de réveil de la *tâche* d'envoi lorsque des données sont disponible dans le *buffer*.

Question 7.3. *Implémentez cette solution.*

Remarque: La question peut se faire en moins de 40 lignes.

Documentation utile: *pthread_create(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), pthread_cond_wait(3), pthread_cond_sign*

8 Threads et mutex

Question 8.1. *Écrivez un programme avec N taches, N étant une constante définie à la compilation. Chacune des taches contiendra une boucle incrémentant et affichant un compteur. Utilisez des sémaphores pour synchroniser les boucles. Vous devriez obtenir le comportement suivant ($N = 4$) :*

```
$ ./concurence
Task: 0, count: 0
Task: 1, count: 0
Task: 2, count: 0
Task: 3, count: 0
Task: 0, count: 1
Task: 1, count: 1
Task: 2, count: 1
Task: 3, count: 1
Task: 0, count: 2
Task: 1, count: 2
Task: 2, count: 2
[...]
```

Documentation utile: *pthread_create(3), pthread_cancel(3), pthread_join(3), pthread_mutex_init, pthread_mutex_lock(3), pthread_mutex_unlock(3), pthread_mutex_destroy(3)*

Question 8.2. *Modifiez votre code pour créer un* dead lock *sur toutes les taches.*

Question 8.3. *Fixez N à 4. Ajoutez une cinquième tache. La cinquième tache devra monopoliser processeur (ie :* `for(;;);`*). En modifiant les priorités des taches, produisez une inversion de priorité.*

Remarque: Sur un système multicoeur, il est possible que vous ayez besoin de créer autant de tâches que de coeurs afin de monopoliser le système.

Documentation utile: *pthread_setschedparam(3), pthread_setschedpolicy(3)*

9 Tâches et sémaphores Xenomai

Afin de faire ce TP, nous vous avons fourni un Linux avec un noyau patché avec Xenomai. Ce patch nous permet d'avoir un système temps réel et profiter d'une API spécialisée pour le temps réel. Notez que nous utiliserons exclusivement l'API Native durant ce TP.

Reportez-vous à l'annexe B pour les informations concernant l'usage de la bibliothèque Xenomai.

Question 9.1. *Écrivez un programme avec N taches, N étant une constante définie à la compilation. Chacune des taches contiendra une boucle incrémentant et affichant un compteur. Utilisez des sémaphores pour synchroniser les boucles. Vous devriez obtenir le comportement suivant ($N = 4$) :*

```
$ ./concurence
Task: 0, count: 0
Task: 1, count: 0
Task: 2, count: 0
Task: 3, count: 0
Task: 0, count: 1
Task: 1, count: 1
Task: 2, count: 1
Task: 3, count: 1
Task: 0, count: 2
Task: 1, count: 2
Task: 2, count: 2
[...]
```

Documentation utile: *rt_task_create, rt_task_start, rt_task_delete, rt_sem_create, rt_sem_delete, rt_sem_p, rt_sem_v, mlockall(2)*

Question 9.2. *Vérifiez le bon fonctionnement de votre programme avec $N = 1000$.*

10 Implémentation d'ordonnanceurs

Le fichier sched.c implémente un environnement capable de simuler des ordonnanceurs. Ce simulateur est basé sur les fonction Posix `makecontext` et `swapcontext` qui permettent de controller manuellement le changement de contexte entre plusieurs tâches.

Question 10.1. *Etudiez* `sched.c`.

Documentation utile: *makecontext(3), swapcontext(3)*

10.1 Algorithmes d'ordonnancement de tâches indépendantes

Question 10.2. *A partir de* `sched.c`*, modifiez la fonciton* `elect` *afin d'implémenter un ordonnanceur à priorité statique.*

Question 10.3. *Plutôt que d'utiliser une durée de simulation arbitraire, calculez celle-ci afin que la durée de la simulation soit suffisante pour faire simulation hors-ligne.*

Question 10.4. *Implémentez un ordonnanceur Rate Monotonic. Bien entendu le membre* `priority` *de la structure* `task_t` *ne sera pas utilisé pour cet algorithme.*

Question 10.5. *Implémentez un ordonnanceur Earliest Dealine First. Bien entendu le membre* `priority` *de la structure* `task_t` *ne sera pas utilisé pour cet algorithme.*

10.2 Algorithmes d'ordonnancement de tâches apériodiques

Nous souhaitons ajouter des tâche apériodiques. Nous les intégrons à notre système en utilisant des tâche avec une periode nulle.

Question 10.6. *Ajoutez la gestion de ces tâches. Utilisez le temps CPU non-utilisé par les tâches périodiques pour éxecuter ces tâches. Vous n'avez pas besoin de gérer de priorités entre les tâches asynchrones.*

Question 10.7. *Effectuez la gestion des tâches aperiodiques dans un serveur de scrutation (*polling server*).*

Question 10.8. *Effectuez la gestion des tâches aperiodiques dans un serveur différé (*defferable server*)*

Question 10.9. ***(Bonus)*** *Effectuez la gestion des tâches aperiodiques dans un serveur sporadique (*sporadic server*).*

Question 10.10. *Implémentez l'algorithme de Round-Robin.*

10.3 Ordonnancement et ressources partagées

Nous souhaitons ajouter la gestion des mutex dans notre système. Le patch `sched_mutex.patch` ajoute les structures nécessaire pour la gestion des mutex.

Question 10.11. *Appliquez* `sched_mutex.patch` *à votre implémentation d'ordonnanceur à priorités statiques et implémentez les fonctions* `lock` *et* `unlock` *;*

Question 10.12. *Implémentez un scénario avec un* deadlock.

Question 10.13. *Implémentez un scénario avec une inversion de priorité.*

Question 10.14. *Implémentez l'algorithme* ICPP *(*Immediate Ceiling Priority Protocol*).*
Documentation utile: *qsort(3)*

Question 10.15. ***(Bonus)*** *Implémentez l'algorithme d'héritage de priorité.*

Documentation utile: *qsort(3)*

Question 10.16. ***(Bonus difficile)*** *Dans notre simulation, les date de préemption sont connue à l'avance. Par conséquent, nous avons pu nous affranchir des problème de réentrance. Modifier votre implémentation d'héritage de priorité afin de la rendre réentrante.*

Question 10.17. ***(Sujet de thèse)*** *Ajoutez à l'ordonnanceur la possibilité d'ordonnacer les tâches sur plusieurs processeurs simultanément. Maintenez votre implémentation d'héritage de priorité fonctionnelle (et réentrante). Garantissez des temps de réponse de vos algorithme en $o(n)+o(s)$ (où n est le nombre de tâche et s le nombre de sections critiques). Vos propositions sont à envoyer à* Ingo Molnár, *auteur des mutex à héritage de priorité, de l'ordonnanceur* Completly Fair *et de la préemption dite* Temps Réelle *sur le kernel Linux.*

Question 10.18. *Implémentez l'algorithme* OCPP *(*Original Ceiling Priority Protocol*).*

Documentation utile: *qsort(3)*

Question 10.19. *Détectez les possibles inversions de priorités dans les fonctions* `m_lock`, `m_unlock`.

Question 10.20. *Détectez les Dead-locks dans les fonctions* `m_lock`, `m_unlock`.

---

11 Libtimer

On se propose d'implémenter une bibliothèque `libtimer` permettant de lancer des *tâches* à une date fixe ou après une certaine période. Le fonctionnement de cette bibliothèque est décrit dans `timer.h` :

```
/*
 * Author: éôJrme Pouiller <jerome@sysmic.fr>
 * Created: Fri Dec 11 22:57:13 CET 2009
 * Licence: GPL
 *
 * Define interface for timer services.
 */

#ifndef MYTIMER_H
#define MYTIMER_H
#include <pthread.h>
#include <time.h>

/* Helper to define a task */
typedef void (*task_t)();

/* INSERT YOUR mytimer_s DEFINITION HERE */

/* Contains all needed data */
typedef struct mytimer_s mytimer_t;

/*
 * All following functions return
 * - 0 on success
 * - another value on error
 */

/*
```

```
 * Initialize new timer instance.
 * Instance is returned using parameter out.
 * User have to call this function first
 */
int mytimer_run(mytimer_t *out);

/* Stop timer instance */
int mytimer_stop(mytimer_t *mt);

/* Add a task to timer. Task will be executed at time tp */
int mytimer_add_absolute(mytimer_t *mt, task_t t, const struct timespec *tp);

/* Add a task to timer. Task will be executed in ms milliseconds */
int mytimer_add_msecond(mytimer_t *mt, task_t t, unsigned long ms);

/* Add a task to timer. Task will be executed in s seconds */
int mytimer_add_second(mytimer_t *mt, task_t t, unsigned long sec);

/* Add a task to timer. Task will be executed in h hours */
int mytimer_add_hour(mytimer_t *mt, task_t t, unsigned long h);

/* Remove first occurence of task t from timer. */
int mytimer_remove(mytimer_t *mt, task_t t);

#endif /* MYTIMER_H */
```

- La structure `mytimer_s` (que vous définirez) contient tout le contexte d'exécution.
- Les *tâches* à lancer sont décrite par le type `task_t`.
- L'utilisateur de la bibliothèque commencera par un appel à `mytimer_run`.
- Il ajoutera les *tâches* avec les fonctions `mytimer_add`.
- `mytimer_add_absolute` exécute la *tâche* à une heure fixe donnée tandis que les autres fonctions `mytimer_add` exécutent la *tâche* à un temps relatif à l'instant présent. Par exemple, `mytimer_add_second(s, t, 3)` exécutera `t` 3 secondes après l'appel à cette fonction.
- Toutes les fonctions de `libtimer` retournent un entier. Il s'agit d'un code d'erreur. Vous êtes libre dans le choix des codes de retour.

On considère que les *tâches* sont exemptes de bugs et qu'elles ne tentent pas de mettre en péril la stabilité de votre bibliothèque et du système.

`libtimer` tentera d'être le plus précis possible dans l'heure de lancement des *tâches*. Néanmoins, on ignorera le biais introduit par le système d'exploitation et les autres processus.

Afin de vous aider, vous trouverez une implémentation de liste chainée dans `list.h` et `list.c` :

```
/*
 * Author: éôJrme Pouiller <jerome@sysmic.fr>
 * Created: Fri Dec 11 22:57:54 CET 2009
 * Licence: GPL
 *
 */
#ifndef LIST_H
#define LIST_H
#include <stdlib.h>
#include <stdbool.h>

typedef struct list_item_s {
   /* ADD YOUR MEMBERS HERE */
   struct list_item_s *next;
} list_item_t;

#define EMPTY_LIST NULL
```

```
#define NEXT(s) (s)->next
#define FOREACH(I, LIST) for (I = LIST; I; I = NEXT(I))

bool list_isEmpty(list_item_t *l);

int list_size(list_item_t *list);

/* Add an element. Don't forget to allocate it before */
void list_add(list_item_t **list, list_item_t *item);

/* Remove an element. Return removed element. Don't forget to free it AFTER removing
   */
list_item_t *list_remove(list_item_t **list, list_item_t *item);

#endif /* LIST_H */
```

```
/*
 * Author: éôJrme Pouiller <jerome@sysmic.fr>
 * Created: Fri Dec 11 22:57:54 CET 2009
 * Licence: GPL
 *
 */
#include "list.h"

bool list_isEmpty(list_item_t *l) {
   return (l == EMPTY_LIST);
}

int list_size(list_item_t *list) {
   int count = 0;
   list_item_t *i;

   FOREACH(i, list)
      count++;
   return count;
}

/* Add an element. Don't forget to allocate it before */
void list_add(list_item_t **list, list_item_t *item) {
   item->next = *list;
   *list = item;
}

/* Remove an element. Return removed element. Don't forget to free it AFTER removing
   */
list_item_t *list_remove(list_item_t **list, list_item_t *item) {
   list_item_t *i;

   if (*list == item) {
      *list = (*list)->next;
      return item;
   }
   FOREACH(i, *list)
      if (i->next == item) {
         i->next = item->next;
         return item;
      }
   // Nothing found
```

```
    return NULL;
}
```

11.1 Version synchrone

On considère pour l'instant que le temps d'exécution des *tâches* est nul.

Question 11.1. *Implémentez* `libtimer` *tel que décrit dans le fichier d'entête* `timer.h`.

Documentation utile: *pthread_cond_timedwait(3), pthread_cond_wait(3), pthread_cond_signal(3), clock_gettime(2), sleep(3), pthread_create(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), pthread_mutex_init(3), pthread_cond_init(3), pthread_cancel(3), ctime_r(3)*

11.2 Version asynchrone

Nous avons supposé dans que le temps d'exécution des *tâches* étaient nul. Nous souhaitons maintenant écrire la version finale de la bibliothèque exécutant les *tâches* de manière asynchrone.

Question 11.2. *Modifiez votre bibliothèque de manière à exécuter les* tâches *de manière concurrente.*

Documentation utile: *pthread_attr_setdetachstate(3), pthread_create(3), pthread_attr_init(3)*

12 Calculs et temps réel

12.1 Introduction

Ce TP est inspiré du fonctionnement d'un téléphone portable. Sur ce système, un microphone utilise un DMA pour placer les données acquises en mémoire. Une interruption est ensuite déclenchée lorsque le DMA contient assez de données par être traitées. Le téléphone portable utilise ensuite un système de compression par paquet dont le temps de traitement est variable. Enfin, le téléphone doit envoyer ses données à l'antenne toutes les 4.6ms, soit à une fréquence de 217Hz environ.

Afin de faire ce TP, nous vous avons fourni un Linux avec un noyau patché avec Xenomai. Ce patch nous permet d'avoir un système temps réel et profiter d'une API spécialisée pour le temps réel.

12.2 Ecriture du Watchdog

Nous allons utiliser le périphérique *Real Time Clock (RTC)* présent sur les architectures i386 pour générer des interruptions nécéssaires à la suite de l'exercice.

A chaque période, la RTC produit une interruption. Le driver RTC traduit cette interruption par l'écriture d'un octet sur le fichier `/dev/rtc0`. Le fichier `rtc.c` vous donne un exemple de programmation du périphérique RTC.

Question 12.1. *Etudiez le programme, compilez-le, puis programmez la RTC à 2 Hz et lancez trois périodes.*

Nous utiliserons la bibliothèque Xenomai pour les questions suivantes. Reportez-vous à l'annexe B pour plus d'informations sur son utilisation.

Nous allons dans la suite du TP travailler avec l'interruption produite par la RTC. Une mauvaise gestion de l'interruption pourra produire un IRQ storm. Nous allons donc mettre en place watchdog afin de rendre notre développement plus confortable. Notre watchdog nous permettra de détecter si notre système ne répond plus et de reprendre la main dessus.

Question 12.2. *Modifiez* `rtc.c` *de manière à ajouter un watchdog. Déclencher votre watchdog sur 1 seconde. Lorsque le watchdog se déclenche, désactivez la RTC et avertissez l'utilisateur.*

Documentation utile: *rt_alarm_start, rt_alarm_create, rt_alarm_delete, rt_alarm_wait, rt_task_create, rt_task_delete, rt_task_start, rt_timer_ns2ticks*

12.3 Ecriture de l'ISR

Maintenant que nous avons un bon outil de test, nous pouvons commencer à travailler sur l'interruption en elle-même.

Les interruptions doivent forcement être éxécutées dans le noyau de Linux. Pour cette raison les questions suivantes sont à écrire sous forme de module. Reportez-vous à l'annexe C pour les informations concernant le développement des modules noyau.

Question 12.3. *Placez une ISR sur le l'interruption 8 (Real Time Clock). Dans votre ISR, gérez un compteur d'appel n. Afficher n dans le buffer du noyau lorsqu'une interruption survient.*

Documentation utile: *rt_intr_create, rt_intr_enable, rt_intr_delete*

Remarquez le fonctionnement du nano-kernel Adeos. Votre interruption apparait dans `/proc/ipipe/Xenomai` alors que les autres interruption apparaissent dans `/proc/ipipe/Linux`. Adeos gère Xenomai et Linux comme deux OS différents.

Nous allons maintenant simuler le traitement de données. Nous choisissons de calculer la suite de Fibonacci pour simuler le temps de traitement des données. Nous utilisons cette fonction en raison de sa grande amplitude de temps de calcul. Ainsi, `fibo(34)` se calcule 4 milliards de fois plus lentement que `fibo(2)`.

Question 12.4. *Faites une implémentation naive (recursive) de Fibonacci. Affichez* `fibo(n % M)` *lorsqu'une interruption survient, M étant une constante définie à la compilation.*

Augmentez M jusqu'à ce que le temps passé dans l'interruption soit mesurable (> 100ms) lorsque n approche un multiple de M.

Remarque: Vous pouvez utilisez le timestamp produit dans le log buffer du noyau comme mesure approximative.

Question 12.5. *Augmentez la fréquence de la RTC à 128Hz. Vérifiez le fonctionnement de l'interruption.*

Remarque: Vous pouvez utilisez le timestamp produit par le log buffer. Vu que l'on casse le temps réel du noyau, ce timestamp n'est sont pas fiable en absolu. Néanmoins, le temps en moyenne est fiable. On peut calculer le nombre d'interruptions entre le début et la fin du test et ainsi voir si on rate des interruption.

Question 12.6. *Déportez le calcul et l'affichage de* `fibo` *dans une seconde tache à l'aide d'une queue. Affichez le nombre d'éléments dans la queue. N'oubliez pas de gérer les erreurs.*

Documentation utile: *rt_queue_create, rt_queue_delete, rt_queue_write, rt_queue_read, rt_queue_inquire, rt_task_create, rt_task_start, rt_task_delete*

Question 12.7. *Ajoutez la gestion d'une file de résultats. Dans l'ISR, dépiler un résultat et affichez-le. Gérez intelligemment le cas où la file est vide.*

Documentation utile: *rt_queue_create, rt_queue_delete, rt_queue_write, rt_queue_read*

Question 12.8. *Chargez le CPU à l'aide des commande dd(1) et ping(1). Vérifiez la qualité de vos taches temps réelles.*

A Instructions d'installation de la machine virtuelle Xenomai

A.1 Mise en service de l'image

Nous avons besoin de configurer votre compte pour utiliser l'image virtuelle patché avec Xenomai.

Ouvrez un terminal. Commencez par importer la configuration de la machine virtuelle :

```
vboxmanage import "SRC_DIR/Ubuntu 9.04 Xenomai 32bits-nodisk.ovf"
```

Cette configuration ne contient pas de dique dur. Nous en attachons donc un :

```
vboxmanage storageattach "Ubuntu 9.04 Xenomai 32bits" --storagectl "SATA Controller"
   --port 1 --type hdd --medium "SRC_DIR/Ubuntu 9.04 Xenomai 32bits_disk1.vmdk"
```

Le disque dur que nous avons ajouté n'est pas accessible en lecture. Nous allons utiliser le système d'image incrémentale proposé par Virtual Box. Ainsi, vous n'avez besoin d'avoir un accès en écriture que sur le fichier contenant le delta. Ce delta pourra tout de même atteindre quelques centaine de mégaoctet. Nous choisissons donc un emplacement en local pour le contenir :

```
vboxmanage modifyvm "Ubuntu 9.04 Xenomai 32bits" --snapshotfolder /var/tmp/snapshots-
   LOGIN/
```

Nous pouvons maintenant créer le premier incrément :

```
vboxmanage snapshot "Ubuntu 9.04 Xenomai 32bits" take "Initial"
```

Vous pouvez démarrer VirtualBox et démarrer la machine virtuelle "Ubuntu 9.04 Xenomai 32bits"

A.2 L'environnement

A.2.1 Se connecter

login : user
password : user

A.2.2 Sauver son travail

Dans un terminal utiliser la commande *scp(1)* pour vous connecter à votre machine hôte par l'IP 10.0.2.2 :

```
scp -r exo1 pouiller@10.0.2.2:
```

L'inverse fonctionne aussi évidement :

```
scp -r pouiller@10.0.2.2:exo1 .
```

A.2.3 Travailler avec emacs, etc...

Vous avez la possibilité d'éxécuter des commandes en root avec la commande *sudo(1)* :

```
sudo apt-get install emacs
```

B Écriture des programmes Xenomai

B.1 Documentation de Xenomai

Vous pourrez trouvez la documentation de l'API de Xenomai dans `/usr/share/doc/xenomai-doc/html/api` ou en ligne sur `http://www.xenomai.org/documentation/xenomai-2.4/html`.

Vous pouvez trouver les headers à inclure en explorant l'onglet *files*.

B.2 Compilation

Les commandes `xeno-config --xeno-cflags` et `xeno-config --xeno-ldflags` permettent d'obtenir respectivement les options de compilation et les options de link à utiliser pour que votre programme fonctionne avec Xenomai. Vous devrez de plus préciser que vous utilisez la skin Native avec l'option `-lnative`.

Par ailleurs, nous vous conseillons fortement de compiler avec l'option `-Wall` de `gcc` et d'utiliser un système de Makefile.

N'oubliez pas que dans un Makefile, il faut utiliser le mot-clef `shell` pour lancer une commande. Par exemple :

```
bin: file.o
	gcc -Wall $(shell xeno-config --xeno-ldflags) -lnative $< -o $@
file.o: file.c
	gcc -Wall $(shell xeno-config --xeno-cflags) -c $< -o $@
```

B.3 Vérouillage de la mémoire

En cas de nécessité, le gestionnaire de mémoire du noyau Linux peut-être amené à placer des pages de mémoire virtuelle sur le disque dur (processus de *swaping*). Lorsqu'une de ces pages de mémoire est nécessaire, elle doit être replacée en mémoire avant d'être utilisée. Cela crée une latence qui peut poser problème dans le cas d'application temps réelles.

Un appel à la fonction *mlockall(1)* permet de demander au noyau de vérouiller les pages de mémoire en mémoire physique :

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

La mémoire doit être vérouillée avant tout appel à l'API Xenomai.

B.4 A propos des noms des objets Xenomai

Il est possible dans Xenomai d'associer chaque objet à un nom unique sur le système. Cela permet de *binder* des objets entre des processus ou entre des domaines d'exécution.

L'utilisation des noms oblige à être rigoureux et à correctement détruire les objets avant de sortir des processus. Les instances des objets resteront en mémoire.

Par conséquent, nous vous conseillons de ne pas utiliser cette fonctionnalité des les exercices suivants et à passer un pointeur `NULL` comme nom.

B.5 Gestion des Erreurs

La plupart des fonctions Xenomai retournent une valeur négative pour indiquer qu'une erreur s'est produite. Vous pouvez utiliser *strerror(3)* pour obtenir le message d'erreur équivalent.

Exemple classique :

```
if (err = rt_task_start(t, f, NULL))
  printf("Task start: %s", strerror(-err));
```

B.6 Exécution

Vous aurez besoin d'exécuter les programmes que vous créerez avec les droits *root*. Nous vous déconseillons l'utilisation de *sudo(1)* pour lancer vos programmes car `Ctrl+C` sera moins réactif.

C Écriture des modules noyau

C.1 Compilation

La manière classique de compiler un module noyau est de créer un Makefile contenant :

```
EXTRA_CFLAGS=-I/lib/modules/$(shell uname -r)/build/include/xenomai
obj-m += module.o
```

Puis lancer la compilation avec :

```
make -C /lib/modules/$(uname -r)/build SUBDIRS=$(pwd) modules
```

Le système de compilation du noyau se chargera de compiler votre module avec les bonnes options.

Nous vous fournissons le fichier `Makefile` reprenant ces lignes.

C.2 Exécution

Les modules ne sont pas des applications. Ils n'ont pas de fonction `main`. Comme c'est souvent le cas lorsque l'on souhaite étendre une application en cours de fonctionnement, vous avez la possibilité de déclarer des fonctions qui seront appelées lors de certain évènements. Les fonctions les plus utiles sont déclarées par `MODULE_INIT` (chargement du module) et `MODULE_EXIT` (déchargement du module).

Vous pourrez charger le module à l'aide de *insmod(1)* et le décharger à l'aide de *rmmod(1)*.

Nous vous fournissons le fichier `module.c` qui vous donne un squelette de module pour le noyau Linux.

C.3 Log Buffer

Le noyau possède un buffer circulaire permettant de logger des messages. Il est possible d'écrire dans le buffer depuis un module à l'aide de *printk(9)* et des macros *pr_debug*, *pr_info*, *pr_notice*, *pr_warning*, *pr_err*, etc.... Il est possible de dumper le contenu du buffer avec la commande *dmesg(1)* (utilisant l'appel système *syslog(2)*). Il est aussi possible de suivre l'évolution du buffer avec la commande :

```
tail -f /var/log/kern.log
```

D Modalités de rendu

Votre rendu fera l'objet d'un traitement automatisé. Afin d'éviter tout problème, veuillez suivre les règles suivantes.

Vous devez rendre votre travail dans une archive gzippée. L'archive devra se nommer `TR-nom.tar.gz` et se décompresser dans le répertoire du même nom.

Votre archives ne pas contenir de fichiers binaires. Il ne doit contenir que vos sources (fichiers sources, fichiers entêtes, Makefile, etc...).

Aucun de vos noms de fichiers ne doit contenir de caractères d'espacements, accentués ou non-imprimables.

Exemple de création de rendu :

```
make clean
cd ..
tar cvzf TR-pouiller.tar.gz TR-pouiller
```

Vous devrez envoyer votre rendu par e-mail avant le 31 Janvier 2013 à 23h59à `<rendus@sysmic.fr>`. Votre e-mail ne devra comporter que votre rendu comme unique pièce jointe.

Le sujet du mail devra suivre la nomenclature suivante : `[TR] nom` (exemple : `[TR] pouiller`).

Enfin, le corps du mail devra contenir une ligne indiquant votre prénom, votre nom et votre email sous la forme :

```
* Prénom Nom <email@entreprise.dom>
```

Exemple :

```
Veuillez trouver en pièce jointe le rendu de projet temps réel de :
* Jérôme Pouiller <j.pouiller@sysmic.fr>

Cordialement,

--
Jérôme Pouiller, Sysmic
Expert Linux embarqué
```

E Modalités de rendu

Votre rendu fera l'objet d'un traitement automatisé. Afin d'éviter tout problème, veuillez suivre les règles suivantes.

Vous devez rendre votre travail dans une archive gzippée. L'archive devra se nommer `TR-nom.tar.gz` où `nom` sera le nom d'un membre de votre équipe que nous désignerons par *contact administratif*. Votre archive devra et se décompresser dans le répertoire du même nom.

Votre archives ne pas contenir de fichiers binaires. Il ne doit contenir que vos sources (fichiers sources, fichiers entêtes, Makefile, etc...).

Aucun de vos noms de fichiers ne doit contenir de caractères d'espacements, accentués ou non-imprimables.

Votre archive devra contenir à sa racine un fichier nommé `AUTHORS`. Ce fichier devra au moins contenir les prénoms, les nom et les email des membre de votre équipe sous la forme :

```
* Prénom1 Nom1 <email1@entreprise1.dom>
* Prénom2 Nom2 <email2@entreprise2.dom>
[...]
```

Les lignes ne correspondant pas à cette nommenclature sont ignorées

Exemple de création de rendu :

```
make clean
echo '* Jérôme Pouiller <j.pouiller@sysmic.fr>' >> AUTHORS
echo '* Thibaud Hilaire <t.hilaire@sysmic.fr>' >> AUTHORS
cd ..
tar cvzf TR-pouiller.tar.gz TR-pouiller
```

Vous devrez envoyer votre rendu par e-mail avant le 31 Janvier 2013 à 23h59à `<rendus@sysmic.fr>`. Votre e-mail ne devra comporter que votre rendu comme unique pièce jointe.

Le sujet du mail devra suivre la nomenclature suivante : `[TR] nom` où `nom` est le nom de votre *contact administratif* (exemple : `[TR] pouiller`).

Enfin, le corps du mail devra se conformer au même formalisme que le fichier `AUTHORS`.

Exemple :

```
Veuillez trouver en pièce jointe le rendu de projet temps réel de :
* Jérôme Pouiller <j.pouiller@sysmic.fr>
* Thibaud Hilaire <t.hilaire@sysmic.fr>

Cordialement,

--
Jérôme Pouiller, Sysmic
Expert Linux embarqué
```