Extensions temps réel pour UNIX

Les outils POSIX
(1003.1b et 1003.1c)

Collectif SITREE – Association GETALL

Cours de Patrick H. E. Foubet

Plan

l. Généralités sur les extensions temps réel pour UNIX.

2. La norme POSIX 1003.1b/c.

3. Résumé.

4. Références.

Partie 1

Généralités sur les extensions temps réel pour UNIX

UNIX et le temps réel

- Environnement différent

- Déterminisme temporel plus faible (matériel et logiciel).

- Environnement de développement convivial

- Intégration technologies non temps réel : IHM, BD.

- Objectifs

- Applications temps réel plus faiblement contraintes (ex : applications multimédias).

- Cohabitation avec des applications temps partagé.

- Ou trouver ces extensions

- Extensions temps réel SVR4 *(ex : priocntl).*

- Norme POSIX 1003.1b/c : présente à la fois sur des systèmes temps réel embarqués que sur des UNIX classiques.

Abstractions et services offerts

- Proches de celles que l'on va trouver dans un exécutif temps réel

  - Abstraction de la concurrence.

  - Manipulation du temps.

  - Synchronisation et communication entre les différentes entités d'exécution du système.

  - Gestion des entrées/sorties.

- Mais plus flexible et garanties temporelles plus faibles. Ex:

  - Préemptivité réduite (Linux) sauf conception adaptée (Solaris), ou points de préemption.

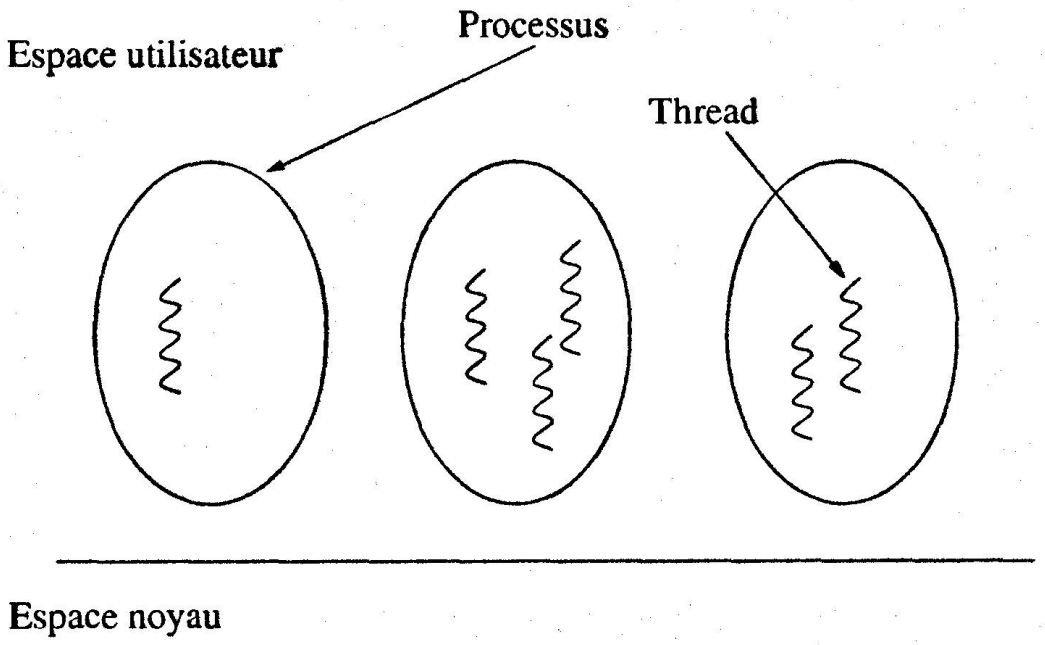
  - Inversion de priorité possible.

  - Précision plus faible des services liés à la manipulation du temps.

  - Services sans timeout.

  - Allocation de ressources parfois cachées.

La notion de thread (1)



- Synonymes : activité, fil d'exécution, processus léger.

- Concept clef : découpage de l'abstraction de processus en deux parties

  1. En flots d'exécution et leur contexte. Plusieurs flots peuvent alors cohabiter dans un même processus.

  2. En espace d'adressage et autres ressources. Partie partagée et vue par l'ensemble des flots d'exécution contenus dans le processus.

- Mise en oeuvre : threads utilisateur, threads noyau [DEM 94].

La notion de thread (2)

- Historique

- Initié par Dennis et Van Horn [DEN 66] dans les années 60 .

- Concept utilisé de tout temps dans les systèmes temps réel [GHO 94].

- Par la suite, utilisation dans les systèmes temps partagé dans des domaines autres que le temps réel.

- Aujourd'hui présent dans presque tous les systèmes d'exploitation. Émergence du concept dans les UNIX commerciaux : vers la moitié des années 90.

La notion de thread (3)

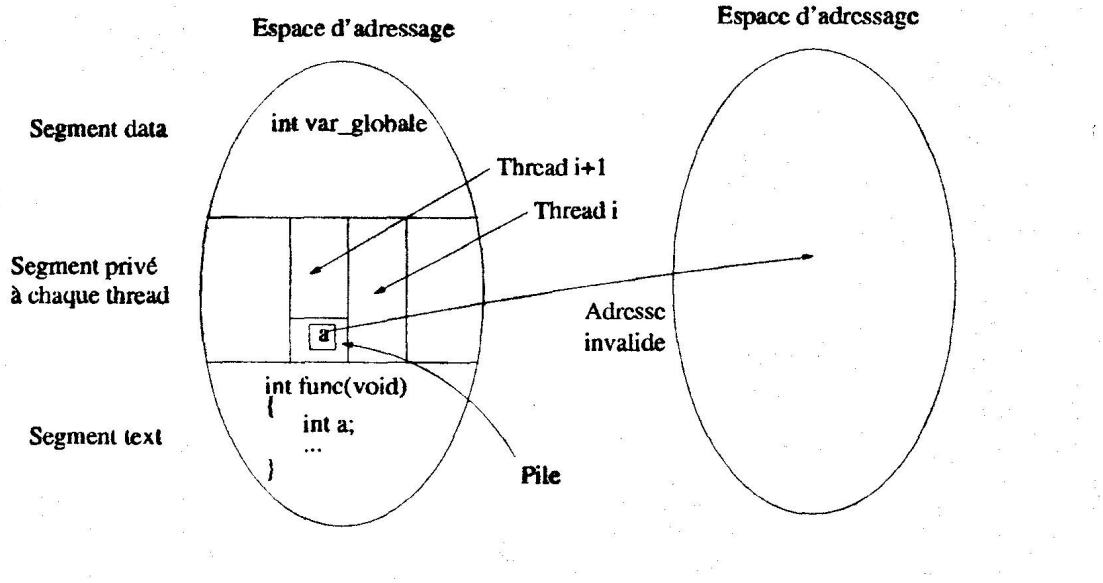
- Le processus met en oeuvre des mécanismes de protection mémoire trop coûteux.
- Utilité des threads
  1. Partage aisé de l'information (pas d'IPC).
  2. Commutation de contexte moins onéreuse (pas d'opération MMU[1]).
  3. Permet de maximiser le débit d'une application (recouvrement du temps de blocage, de communication, exploitation des architectures SMP[2]).
  4. Flexibilité de la gestion des ressources avec des gestionnaires dits "utilisateurs" (ex ordonnancement, mémoire, etc).
  5. Modèle de programmation.

[1]MMU Pour Memory Management Unit.

[2]SMP pour Symmetric MultiProcessing (Shared Memory Multiprocessor).

La notion de thread (4)

- Rappel sur les espaces d'adressage



- Adressage virtuel (32 bits → 4 giga). Adresse virtuelle ≠ adresse physique.

- Frontière de la protection mémoire = espace d'adressage virtuel.

- Généralement découpé en "segments" (text, data, BSS, static) . Segment "privé" au sens "caché".

- La segmentation dépend du système, de l'architecture matériel, du compilateur/éditeur de liens.

La notion de thread (5)

• Attention à l'ordonnancement et à la concurrence d'accès aux ressources (ressources systèmes et données de l'application) !

- Exemple

```
#include <stdio.h>

int a=100;

int main(int argc, char* argv[])

{

        if (fork()==0)

                a+=100;

      else          {

                sleep(1);

                printf ( "a = %d\n" , a) ;

        }

}
```

- Quelle est la valeur affichée après le fork() ?
- Idem avec des threads ?

La notion de thread (6)

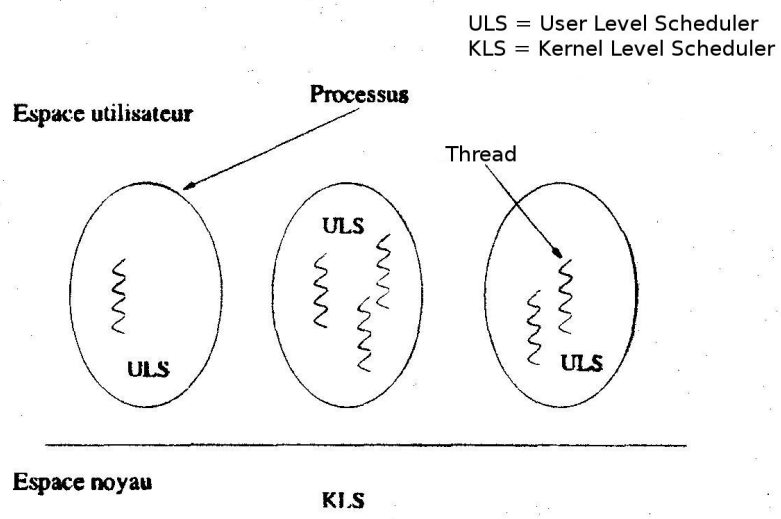
- Exemple 2 :

```
#include <stdio.h>
int a=100;
int main(int argc, char* argv[])
{
        if (fork()==0) a+=100;
        else   a+=200;
           sleep(1);
           printf("a = %d\n",a);
}
```

- Quelle est la valeur affichée après le fork() ?
- Idem avec des threads ?

Mise en oeuvre (1)

L'approche "thread utilisateur" : la notion de thread est offerte par une bibliothèque. Le noyau ne connaît pas l'abstraction de thread.



- Grande flexibilité (ex : Ordonnanceur utilisateur).

- Pas besoin de toucher au système d'exploitation.

- Ordonnancement global délicat : peu adapté au temps réel.

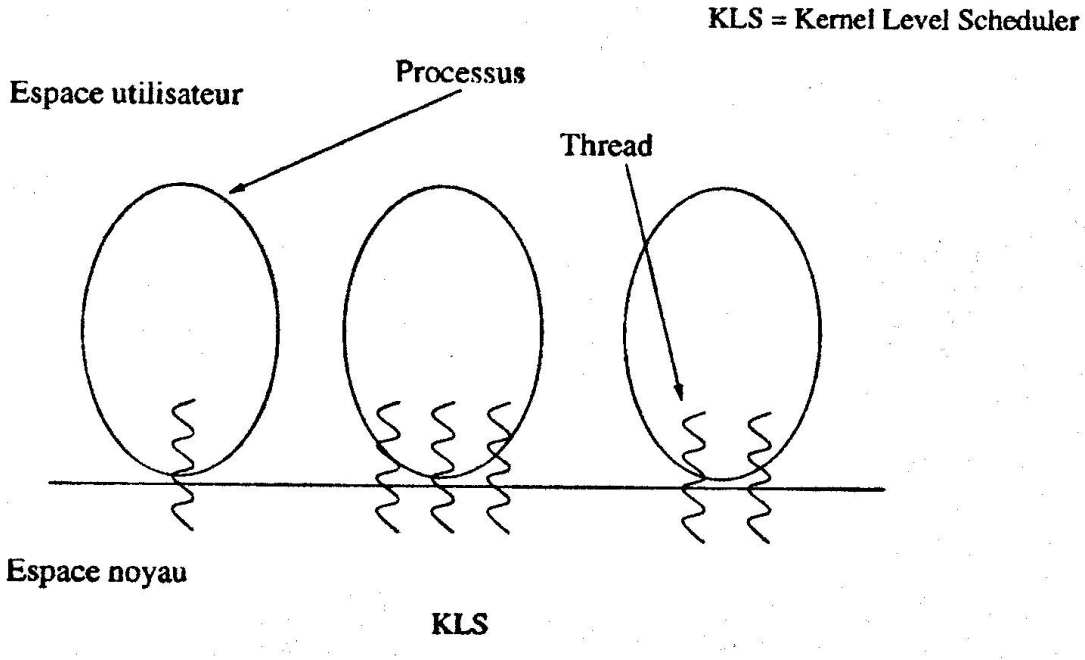
- Coût de commutation entre threads d'un même processus quasi nul (co-routinage).

- Difficultés avec les appels système bloquants.

- Ex : thread SunOS, Cthread, etc.

Mise en oeuvre (2)

L'approche '"thread noyau" : la notion de thread est une abstraction offerte par le système d'exploitation.



- Services offerts par appels système.

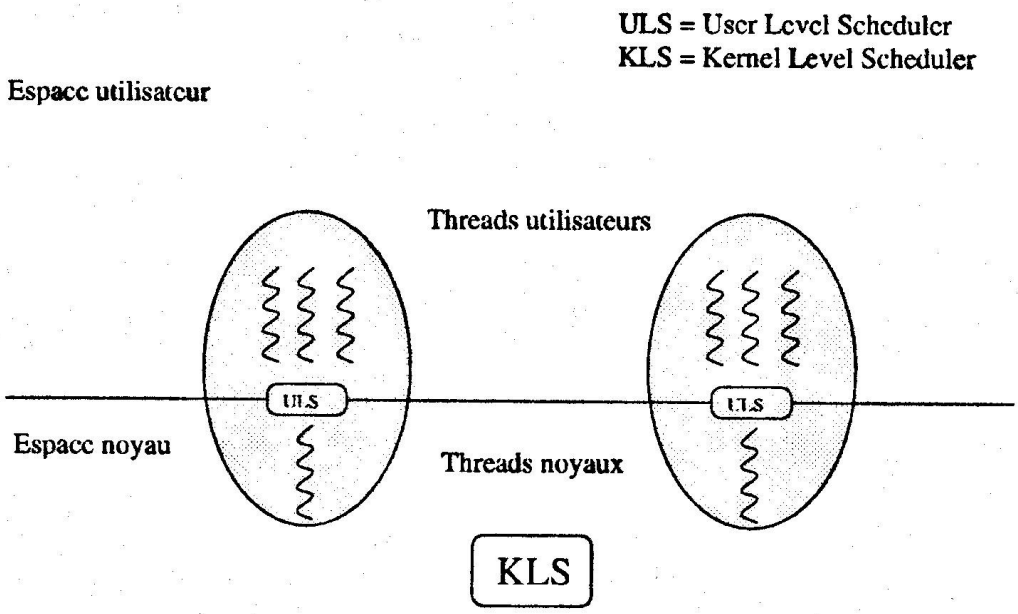
- Impose une refonte du système d'exploitation (ex Solaris).

- Ordonnancement global possible.

- Coût de commutation plus élevé (mais néanmoins toujours plus faible qu'avec des processus). - Ex : CHORUS, Solaris, etc.

Mise en oeuvre (3)

L'approche "hybride"



- Apporte l'efficacité et la flexibilité des threads utilisateur sans leurs problèmes liés aux appels bloquants et à l'ordonnancement "global" du système.

- ULS + KLS : coordination par événements et mémoire partagée.
Ordonnancement à deux niveaux.

- Peu utilisé à ce jour (pb de sécurité et refonte des noyaux actuels).

- Ex : Split level scheduling [GOV 91], Scheduler activation [AND 92], etc.

Mise en oeuvre (4)

- Sur Linux

- Equivalence de thread noyau bien que la notion de thread n'existe pas dans le noyau.

- Appel système *clone()* => duplication d'un processus avec partage de la mémoire et des descripteurs de fichiers.

- Implantation des threads par une bibliothèque.

- Interface de thread POSIX 1003.1c. Implantation très incomplète et sémantique parfois différente.

- Sur Solaris

- Interface des threads POSIX.1003.1c + interface propriétaire.

- Thread noyau (LWP pour Light Weight Process) et thread utilisateur. Grande flexibilité sur le couplage entre thread noyau et thread utilisateur.

Partie 2

La norme POSIX 1003.1c

La norme POSIX (1)

• Objectif : définir une interface standard des services offerts par UNIX afin d'offrir une certaine portabilité des applications [VAH 96] & [JMR 93].

• Norme étudiée/publiée conjointement par l'ISO et l'ANSI.

• Problèmes

- Portabilité difficile car il existe beaucoup de différences entre les UNIX.

- Tout n'est pas (et ne peut pas) être normalisé.

- Divergence dans l'implantation des services POSIX (ex : threads sur Linux).

- Architecture de la norme.

• Exemple de systèmes POSIX : Lynx/OS, VxWorks, Solaris, Linux, QNX, ... etc

(presque tous les systèmes Unix temps réel).

La norme POSIX (2)

• Architecture de la norme : découpée en chapitres optionnels et obligatoires. Chaque chapitre contient des parties obligatoirement présentes, et d'autres optionnelles.

- Exemple de chapitres de la norme POSIX

Chapitres	Signification
POSIX 1003.1	Services de base (*fork*, *exec*, ...)
POSIX 1003.1a	Commandes shell (ex : sh)
POSIX 1003.1b (ex Posix.4 [GAL 95])	Temps réel
POSIX 1003.1c (ex Posix.4.a [RIF 95])	Threads
POSIX 1003.1d (ex Posix.4.b)	Autres extensions temps réel
POSIX 5	POSIX.1 en ADA

La norme POSIX (3)

• Cas du chapitre POSIX 1003.1b : presque tout les composants sont optionnels !!

Nom	Signification
_POSIX_PRIORITY_SCHEDULING	Ordonnancement à priorité fixe
_POSIX_REALTIME_SIGNALS	Signaux temps réel
_POSIX_ASYNCHRONOUS_IO	E/S asynchrones
_POSIX_TIMERS	Chien de garde
_POSIX_SEMAPHORES	Sémaphores
etc ...	

• Conséquence : que veut dire "être conforme POSIX 1003.1b" ? ... pas grand chose puisque la partie obligatoire n'est pas suffisante pour construire des applications temps réel.

La norme POSIX (4)

- Pour être portable, une application doit elle même déterminer si les éléments dont elle a besoin sont présents

1. Lors de la compilation : grâce à des macros prédéfinies dans *unistd.h*

```
#include <unistd.h>
#if _POSIX_VERSION < 199309L
#error POSIX absent
#else
#ifndef _POSIX_PRIORITY_SCHEDULING
#error POSIX pas d'ordonnancement HPF
#endif
#endif
```

2. A l'exécution, grâce à la fonction

```
#include <unistd.h>
long sysconf(int name);
```

qui permet de tester la présence de fonctionnalités.

Services POSIX 1003.1b et 1003.1c

2.1 Les threads POSIX.

2.2 Outils de synchronisation.

2.3 Les signaux temps réel.

2.4 La manipulation du temps.

2.5 Les entrées/sorties asynchrones.

2.6 Les files de messages.

2.7 Services d'ordonnancement.

2.8 La gestion mémoire.

Sous partie 2.1

Les threads POSIX

Les threads POSIX (1)

• Définis par le chapitre POSIX 1003.1c. Ce chapitre contient à la fois les threads et les outils de synchronisation liés aux threads (ex : mutex).

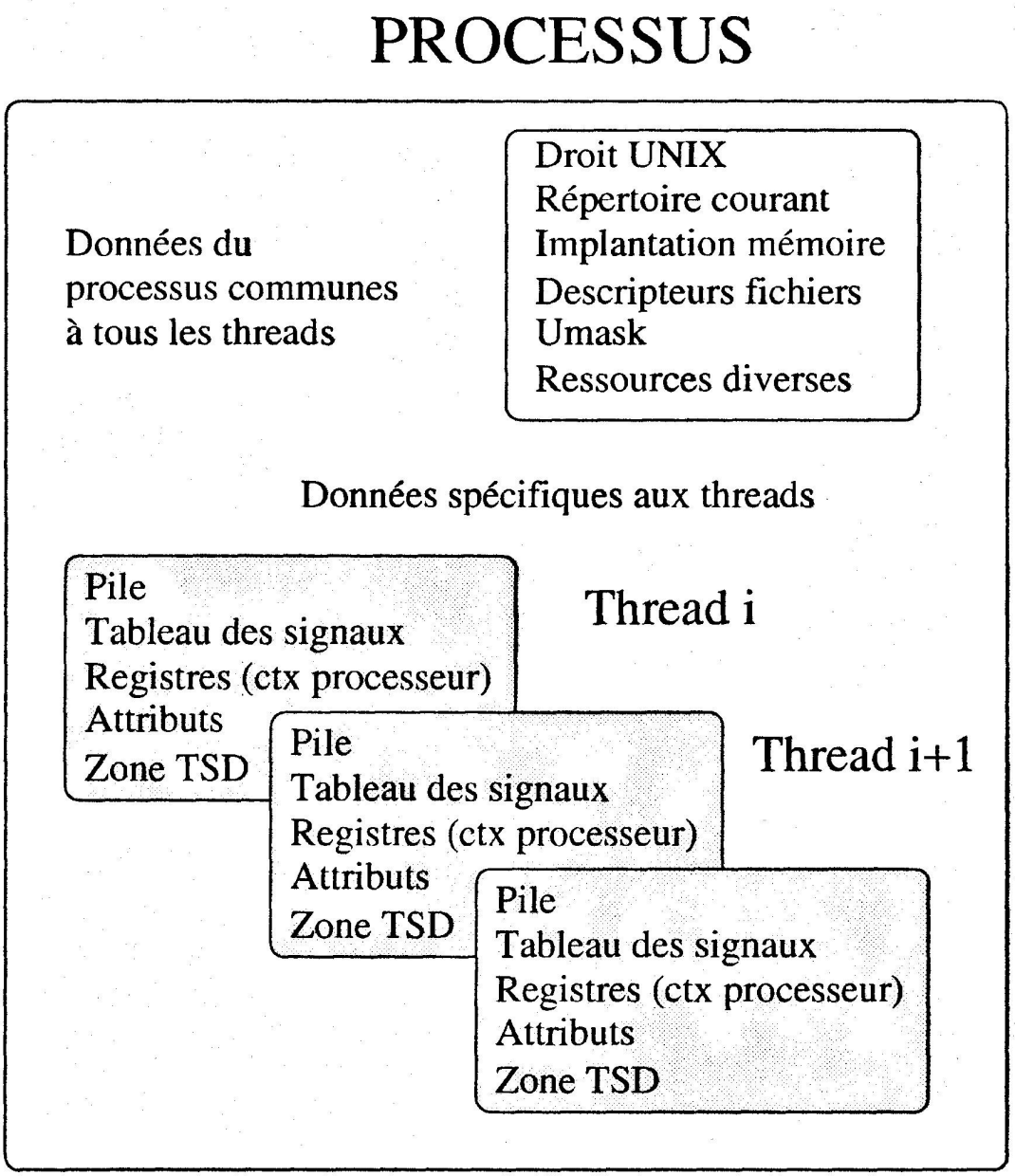
- Caractéristiques

- Un thread POSIX est défini par un identifiant local au processus. Il possède une pile, un contexte et un ensemble d'attributs qui définissent son comportement.

- Un thread POSIX peut être implanté, soit sous la forme utilisateur, soit sous la forme noyau => la norme ne définit que l'interface et non son implémentation.

Les threads POSIX (2)

- Description d'un processus/thread(s) POSIX



Les threads POSIX (3)

• Nécessite l'utilisation de code "réentrant" (ou "thread-safe") => code construit de façon à permettre une exécution concurrente sûre.

- Un code réentrant

- ne manipule pas de variable partagée.

- ou alors manipule les variables partagées en section critique.

• Tout doit être réentrant : le code utilisateur et les bibliothèques systèmes (ex : libc.so).

Les threads POSIX (4)

. Comment rendre "thread-safe"/réentrant du code?

- Technique 1 : compiler avec le symbole *_REENTRANT*.

```
#if defined( _REENTRANT)
extern int *___errno();
#define errno (*(___errno()))
#else extern int errno;
#endif
```

- Technique 2 : modification de l'implantation : mise en oeuvre de section critique sur les données partagées.

```
int fprintf(const char *format, /* args */ ...);
```

- Technique 3 : nouvelle signature : supprimer les données partagées

```
struct hostent *gethostbyname(const char *chaine);
struct hostent *gethostbyname_r(const char *ch, struct hostent *result);
```

Les threads POSIX (5)

pthread_create	Création d'un thread.
pthread_exit	Paramètres : code, attributs, arg. Terminaison d'un thread.
pthread_self	Paramètre : code retour. Renvoie l'identifiant d'un thread
pthread_cancel	Destruction d'un thread.
pthread_join	Paramètre : identifiant du thread. Suspend un thread => la terminaison d'un autre
pthread_detach	Suppression du lien de parenté entre thread.
pthread_kill	Emet un signal vers un thread.
pthread_sigmask	Modifie le masque de signal d'un thread.

Les threads POSIX (6)

• Dans un système POSIX 1003.1c, certains services ont une sémantique différente.

Exemple :

- Sémantique de *fork() :* crée un nouveau processus contenant un thread dont le code est la fonction *main()*.

- Sémantique de *exit() :* termine un processus et donc tous les threads contenus dans celui-ci.

Les threads POSIX (7)

- Création de thread et opération *join*

```
#include <pthread.h>
void* th(void* arg)
{
     printf(“Je suis le thread %d processus %d\n",
          pthread_self(),getpid());
     pthread_exit(NULL);
}
int main(int argc, char* argv[])
{
pthread_t id1 ,id2;
     pthread_create(&id1,NULL,th,NULL);
     pthread_create(&id2,NULL,th,NULL);
     pthread_join(id1,NULL);
     pthread_join(id2,NULL);
     printf("Fin du thread principal %d processus %d\n",
            pthread_self(),getpid());
     pthread_exit(NULL);
}
```

Les threads POSIX (8)

- Compilation et exécution 1. Sur Solaris :

$ gcc -D_REENTRANT create.c -o create -lpthread -lrt

$ create

Je suis le thread 4 processus 5539

Je suis le thread 5 processus 5539

Fin du thread principal 1 processus 5539

2. Sur Linux

$ gcc -D_REENTRANT creates -o create -lpthread

$ create

Je suis le thread 1026 processus 1253

Je suis le thread 2051 processus 1254

Fin du thread principal 1024 processus 1251

Les attributs de thread (1)

• Attributs d'un thread : caractéristiques d'un thread positionnées lors de sa création. Pas d'héritage entre thread père et thread fils.

• Si l'on ne précise pas la valeur d'un attribut lors de 1a création d'un thread, une valeur par défaut lui est affectée.

- Exemples d'attribut

Nom d'attribut Signification

detachstate	*pthread_join* possible ou non
policy	Politique d'ordonnancement
priority	Priorité
stacksize	Taille de la pile spécifiée par le programmeur ou non

Les attributs de thread (2)

• Lors de la création d'un thread, un objet de type *pthread_attr_t* peut être fourni si l'on souhaite spécifier explicitement les attributs du futur thread

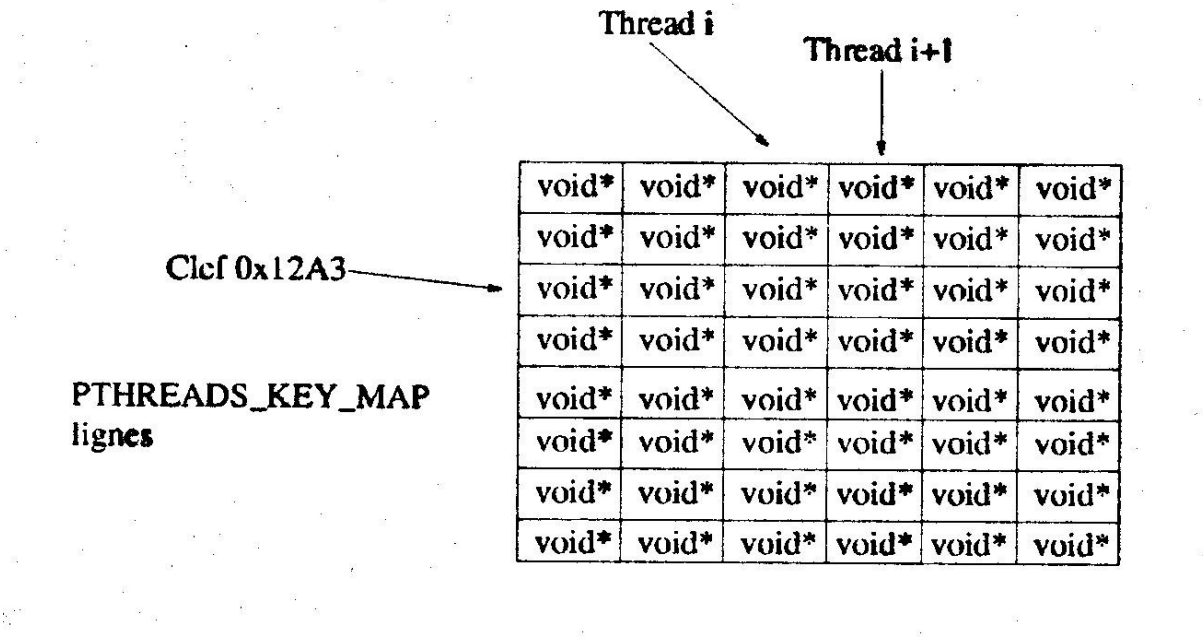
pthread_attr_init	Création d'un objet attribut.
pthread_attr_delete	Destruction d'un objet attribut.
pthread_attr_setATT	Positionne la valeur d'un attribut.
pthread_attr_getATT	Consulte la valeur d'un attribut

où *ATT* constitue le nom de l'attribut.

Les attributs de thread (3)

```
#include <pthread.h>
void* th(void* arg)
{
     printf("Je suis la thread %d\n", pthread_self ());
}
int main(int argc, char* argv[])
{
int i;
pthread_t id;
pthread_attr_t Attr;
struct sched_param Param;
    pthread_attr_init(&Attr);
    pthread_attr_setdetachstate(&Attr, PTHREAD_CREATE_DETACHED);
    pthread_attr_setschedpolicy(&Attr, SCHED_FIFO);
    Param.sched_priority=1;
    pthread_attr_setschedparam(&Attr , &Param);
    for(i=1;i<10;i++)
           pthread_create(&id, &Attr, th, NULL);
}
```

Les attributs de thread (4)



• La TSD (Thread Specific Data area) : zone mémoire permettant de stocker des informations spécifiques à chaque thread.

• Permet l'extension des attributs standards.

pthread_key_create	Création d'une clef.
pthread_key_delete	Destruction d'une clef.
pthread_getspecific	Consulte le pointeur associé à une clef du thread courant.
pthread_setsspecific	Initialise le pointeur associé à une clef du thread courant.

Les attributs de thread (5)

● Création d'un nouvel attribut

```
pthread_key_t cd_key;
int pthread_cd_init(void)
{
     return pthread_key_create(&cd_key, NULL);
}
char* pthread_get_cd(void)
{
     return (char*)pthread_getspecific(cd_key);
}
int pthread_set_cd(char* cd)
{
     char * mycd = (char*)malloc(sizeof(char)*100);
     strcpy(mycd,cd);
     return pthread_setspecific(cd_key,mycd);
}
int main(int argc, char* argv [])
{
     pthread_cd_init();
     pthread_set_cd("/rep1/rep2");
     printf("Mon repertoire courant est %s\n",pthread_get_cd());
}
```

Sous partie 2.2

Outils de synchronisation

- Principaux outils :

1. Les mutex.
2. Les variables conditionnelles.
3. Les sémaphores à compteur.

Les mutex (1)

• Sémaphores optimisés pour la mise en oeuvre de section critique.

• Gestion de la file d'attente : dépend de la politique d'ordonnancement utilisée (SCHED_FIFO, SCHED_OTHER, etc ...). Normalement, les threads sont réveillés par ordre décroissant de leur priorité.

• Comportement défini par un ensemble d'attributs dont les principaux sont les suivants :

Nom d'attribut	Signification
protocol	Protocole d'héritage à utiliser
pshared	Mutex inter-processus ou inter-thread
ceiling	Plafond de priorité

Les mutex (2)

pthread_mutex_init	Initialise un mutex.
pthread_ mutex_lock	Prise éventuellement bloquante du verrou.
pthread_mutex_trylock	*Tentative non bloquante de prise du verrou.*
pthread_ mutex_unlock	Libération du verrou.
pthread_mutex_destroy	*Destruction du verrou.*
pthread_mutexattr_init	Initialisation d'une structure attribut.
pthread_mutexattr_setATT	Positionne un attribut.
pthread_mutexattr_getATT	*Consulte un attribut.*

où *ATT* est un attribut du mutex.

Les variables conditionnelles (1)

• Paradigme de programmation permettant de bloquer un thread en attente d'une condition à remplir.

• Principes :

- Basé sur le couplage d'un mutex et d'une variable dite "conditionnelle".

- Le réveil est sans mémoire : si aucun thread n'attend la condition, l'événement de réveil est perdu (≠ aux sémaphores à compteur).

- Gestion de la file d'attente : dépend de la politique d'ordonnancement utilisée *(SCHED_FIFO, SCHED_OTHER,* etc ...). Normalement, les threads sont réveillés par ordre décroissant de leur priorité.

Les variables conditionnelles (2)

pthread_cond_init	Initialise une variable.
pthread_cond_destroy	Détruit une variable.
pthread_cond_wait	Attend sur une condition.
pthread_cond_signal	Signale l'arrivée de la condition à un thread.
pthread_cond_broadcast	Signale l'arrivée de la condition à tous les threads.

Les variables conditionnelles (3)

• Programme type d'un thread modifiant la variable : on utilise un mutex *(var_mutex)* et une variable conditionnelle *(var_cond)*.

```
pthread_mutex_lock(&var_mutex);
/* Modification de la variable en section critique */
var = ..... ;
/* Si la condition est remplie, on avertit le thread bloqué */
if (condition(var))
        pthread_cond_signal(&var_cond);
pthread_mutex_unlock(&var_mutex);
```

Les variables conditionnelles (4)

• Programme type d'un thread qui attend la modification d'une variable :

```
pthread_mutex_lock(&var_mutex);
while (!condition(var)) {
        /* Si la condition n'est pas remplie, attendre */
          pthread_cond_wait(&var_cond, &var_mutex);
}
/* Exploiter la variable en section critique */
pthread_mutex_unlock(&var_mutex);
```

• Attention *pthread_cond_wait* effectue une libération/ acquisition implicite du mutex.

Les variables conditionnelles (5)

• Exemple :

```
int y=2, x=0;
pthread_mutex_t mut;
pthread_cond_t cond;
void* th(void* arg)
{
int cont=1;
    while(cont)  {
          pthread_mutex_lock(&mut); x++;
          printf("x++\n");
          if (x > y)  {
           pthread_cond_signal(&cond);
           cont=0;
          }
          pthread_mutex_unlock(&mut);
    }
}
```

Les variables conditionnelles (6)

```
int main(int argc, char* argv)
{
pthread_t id;
     pthread_mutex_init(&mut,NULL);
     pthread_cond_init(&cond,NULL);
     pthread_create(&id,NULL,th,NULL);
     pthread_mutex_lock(&mut);
     while (x <= y) pthread_cond_wait(&cond, &mut);
     printf("x>y est vrai\n");
     pthread_mutex_unlock(&mut);
}
```

- Exécution :

$ cond &

x++

x++

x++

x>y est vrai

Sémaphore à compteur (1)

- Sémaphore classique : file d'attente + compteur.
- Pas de gestion des problèmes d'inversion de priorité : phénomène à régler au niveau utilisateur.
- Utilisation pour la synchronisation et la mise en oeuvre d'exclusion mutuelle inter-processus et inter-thread.
- Gestion de la file d'attente : dépend de la politique d'ordonnancement utilisée *(SCHED_FIFO, SCHED_OTHER, …).* Normalement, les threads sont réveillés par ordre décroissant de leur priorité.
- Deux types de sémaphores :
  1. sémaphores nommés
  2. et non nommés.

Sémaphore à compteur (2)

sem_open	Connexion à un sémaphore nommé.
sem_close	Déconnexion d'un sémaphore nommé.
sem_unlink	Destruction d'un sémaphore nommé.
sem_init	Initialisation d'un sémaphore non nommé.
sem_destroy	Destruction d'un sémaphore non nommé.
sem_post	Libération d'un sémaphore.
sem_wait	Acquisition d'un sémaphore.
sem_trywait	Acquisition non bloquante d'un sémaphore.

Sémaphore à compteur (3)

• Exemple :

```
#include <pthread.h>
#include <semaphore.h>
sem_t sem;
int main (int argc, char* argv[])
{
pthread_t id;
struct timespec delai;
    sem_init(&sem,0,0);
    pthread_create(&id,NULL,th,NULL);
    delai.tv_sec=4;
    delai.tv_nsec=0;
    nanosleep(&delai,NULL);
    printf("thread principal %d : ", pthread_self());
    printf("liberation de l'autre thread \n",
    sem_post(&sem);
    pthread_exit(NULL);
}
```

Sémaphore à compteur (4)

- Exemple (suite) :

```
void* th(void* arg)
{
    printf("thread %d en attente\n",pthread_self());
    sem_wait(&sem);
    printf("thread %d debloqué\n",pthread_self());
}
```

- Exécution :

$sem &

thread 4 en attente

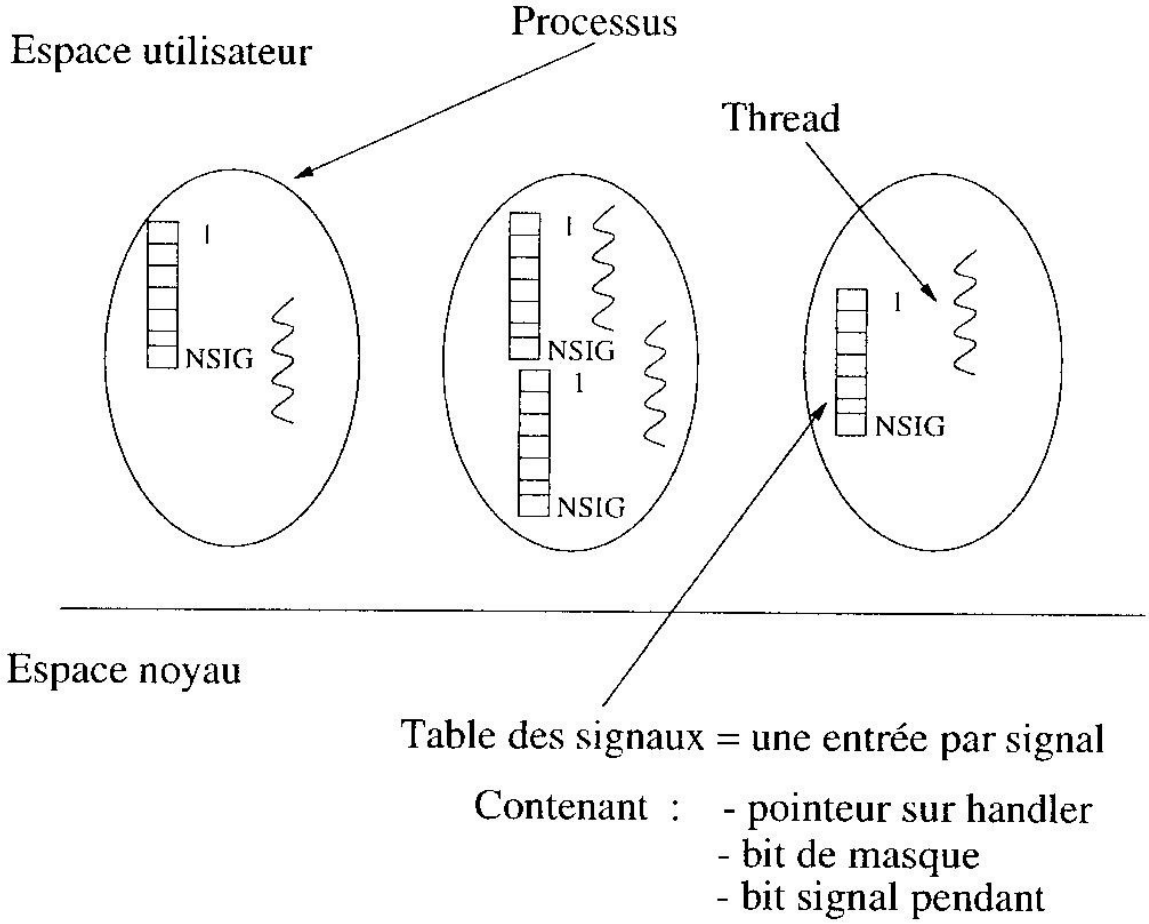
thread principal 1 : liberation de l'autre thread

thread 4 debloqué

Sous partie 2.3

Les signaux temps réel

Les signaux temps réel (1)



- Signal = événement délivré de façon asynchrone à un processus/thread = interruption logicielle.

- Signaux bloqués (ou masqués), pendants ou délivrés.

- Comportement standard modifiable par l'utilisateur.

- Table de signaux par thread/processus.

Les signaux temps réel (2)

• Abstraction déjà présente dans POSIX 1003.l mais posant les problèmes suivants :

1. Implantation de la table des signaux pendants => livraison non fiable (perte possible).

2. Ordre d'émission non respecté lors de la livraison.

3. Ne véhicule pas d'information et peu de signaux disponibles pour l'utilisateur : peu adapté à la mise en œuvre d'IPC.

4. Faible performance (lent : latence importante).

Les signaux temps réel (3)

• Interface POSIX 1003.1 :

kill	Emission d'un signal.
sigaction	Connexion d'un handler à un signal.
sigemptyset	Initialise un masque vide.
sigfillset	Initialise un masque avec tous les signaux.
sigaddset	Ajoute un signal dans un masque.
sigdelset	Supprime un signal d'un masque.
sigismember	Teste la présence d'un signal dans un masque.
sigsuspend	Bloque un processus jusqu'à la réception d'un signal.
sigprocmask	Installe un masque.

Les signaux temps réel (4)

• Exemple : signaux non temps réel

```
#include <stdio.h>
#include <signal.h>
void handler(int sig)
{
     printf("Signal %d recu\n",sig);
}
int main(int argc, char * argv[])
{
struct sigaction sig;
    sig.sa_flags=SA_RESTART;
    sig.sa_handler=handler;
    sigemptyset(&sig.sa_mask);
    sigaction(SIGUSR1,&sig,NULL);
    while(1); /* le programme travaille !! */
}
```

- Exécution :

[phf@christmas ex]$ ./sig1&

[1] 2491

[phf@christmas ex]$ kill -USR1 2491

[phf@christmas ex]$ Signal 10 recu

Les signaux temps réel (5)

- Extension POSIX 1003.1b :

- Plage de nouveaux signaux numérotés de SIGRTMIN à SIGRTMAX (au moins *RTSIG_MAX* signaux).

- Possibilité de signaux valués.

- Plus de perte de signaux : utilisation d'une file d'attente pour les signaux pendants.

- Livraison ordonnée : respect de la politique d'ordonnancement des processus bloqués + priorité liée au signal => SIGRTMIN est le plus prioritaire.

- Emission par k*ill*, *sigqueue*, par *timer* ou E/S asynchrone.

- Interface complémentaire de POSIX 1003.1b :

sigqueue	Emission d'un signal temps réel.
sigwaitinfo	Attente d'un signal sans lancement du handler
sigtimedwait	Idem ci-dessus + timeout sur le temps de blocage.

Les signaux temps réel (6)

- Exemple : avec des signaux temps réel

```
int main(int argc, char * argv[])
{
struct sigaction sig;
union sigval val;
int cpt;
     sig.sa_flags=SA_SIGINFO;
     sig.sa_sigaction=handler;
     sigemptyset(&sig.sa_mask);
     if(sigaction(SIGRTMIN,&sig,NULL)<0) perror("sigaction");
     for(cpt=0;cpt<5;cpt++) {
          struct timespec delai;
          delai.tv_sec=1;
          delai.tv_nsec=0;
          val.sival_int=cpt;
          sigqueue(getpid(),SIGRTMIN,val);
          nanosleep(&delai,NULL);
     }
}
```

Les signaux temps réel (7)

```
void handler (int sig, siginfo_t *sip, void *uap)
{
    printf("Reception signal %d, val = '%d \n",
           sig, sip->si_value.sival_int);
}
```

- Exécution:

[phf@christmas ex]$ ./sig2

Reception signal 35, val = 0

Reception signal 35, val = 1

Reception signal 35, val = 2

Reception signal 35, val = 3

Reception signal 35, val = 4

Sous partie 2.4

La manipulation du temps

La manipulation du temps (1)

• Services liés au temps :

1. Quelle heure est il?

2. Bloquer un thread/processus pendant une durée donnée.

3. Réveiller un thread/processus régulièrement *(timer)* => tâches périodiques.

• Précision de ces services :

- Liée au matériel présent (circuit d'horloge), à ses caractéristiques (période d'interruption) et au logiciel qui l'utilise (handler d'interruption)

- Ex : Linux intel : circuit activé périodiquement (10 ms) + registre RDTSC du Pentium. Résultat = mesure en micro-seconde *(gettimeofday)* mais temps de réveil autour de 10/12 ms => si *nanosleep <= 12* ms et *SCHED_RR ou SCHED_FIFO* = attente active avec précision 1 ou 2 ms.

La manipulation du temps (2)

• Service de temps déja existant sur les UNIX POSIX 1003.l, BSD ou SVR4. Ex : *gettimeofday*, timer BSD *setitimer*.

• Extensions POSIX 1003.1b :

- Support de plusieurs horloges possible => plusieurs horloges physiques, profiling.

- Impose la présence d'au moins une horloge : *CLOCK_REALTIME* (précision d'au moins *20* ms).

- Structure *timespec* précision "théorique" jusqu'à la micro-seconde ....

- Services disponibles :

  - Consultation et modification des horloges.
  - Mise en sommeil d'une tâche.
  - Timer périodique couplé avec les signaux UNIX ; éventuellement avec les signaux temps réel. Avec ou sans réarmement automatique.

La manipulation du temps (3)

clock_gettime	Consulte la valeur d'une horloge.
clock_settime	Modifie la valeur d'une horloge.
clock_getres	Obtention de la précision d'une horloge.
timer_create	Crée un timer.
timer_delete	Détruit un timer.
timer_getoverrun	Donne le nombre de signaux non traités.
timer_settime	Active un timer.
timer_gettime	Consulte le temps restant avant terminaison du timer.
nanosleep	Bloque un processus/thread pendant une durée donnée.

La manipulation du temps (4)

• Exemple de timer avec SIGALRM :

```
int main(int argc, char * argv[])
{
timer_t monTimer;
struct sigaction sig;
struct itimerspec ti;
     timer_create(CLOCK_REALTIME,NULL,&monTimer);
     sig.sa_flags=SA_RESTART;
     sig.sa handler=trop_tard;
     sigemptyset(&sig.sa_mask);
     sigaction(SIGALRM,&sig,NULL);
     ti.it_value.tv_sec=capacite;
     ti.it_value.tv_nsec=0;
     ti.it_interval.tv_sec=0;    /* ici le timer n'est pas */
     ti.it_interval.tv_nsec=0;   /* automatiquement réarmé */
     timer_settime(monTimer,0,&ti,NULL);
     printf("Debut capacite\n");
     while(go>0)  printf("Je travaille ... \n");
     printf("Debloquee par timer : echeance rates ..\n");
}
```

La manipulation du temps (5) `

• Exemple (suite) :

```
int go=1;
void trop_tard(int sig)
{
     printf("Signal %d recu\n",sig);
     go=0;
}
```

• Exécution :

$timer

Debut capacite

Je travaille ...

Je travaille ...

Je travaille ...

Je travaille ...

Je travaille ... Signal 14 recu

Debloque par timer : echeance ratee ..

La manipulation du temps (6)

• Exemple de timer avec SIGRTMIN :

```
timer_t monTimer;
struct sigevent event;
struct sigaction sig;
struct itimerspec ti;
     event.sigev_notify=SIGEV_SIGNAL;
     event.sigev_value.sival_int=capacite;
     event.sigev_signo=SIGRTMIN;
     timer_create(CLOCK_REALTIME,&event,&monTimer);
     sig.sa_flags=SA_SIGINFO;
     sig.sa_sigaction=trop_tard;
     sigemptyset(&sig.sa_mask);
     sigaction(SIGRTMIN,&sig,NULL);
     ti.it_value.tv_sec=capacite;
     ti.it_value.tv_nsec=0;
     ti.it_interval.tv_sec=0;  /* timer non réarmé */
     ti.it_interval.tv_nsec=0;
     timer_settime(monTimer,0,&ti,NULL);
     printf("Debut capacite\n");
     while(go>0) printf("Je travaille ...\n");
     printf("Debloquee par timer : echeance ratee ..\n");
```

La manipulation du temps (7)

• Exemple (suite) :

```
int go=1;
void trop_tard(int sig, siginfo_t *info, void *uap)
{
printf("Reception signal %d, capacite = %d depassee \n",
          sig,info->si_value.sival_int);
go=0;
}
```

- Exécution :

$timer-rt

Debut capacite

Je travaille ...

Je travaille ...

Je travaille ...

Je travaille ...

Je travaille ...

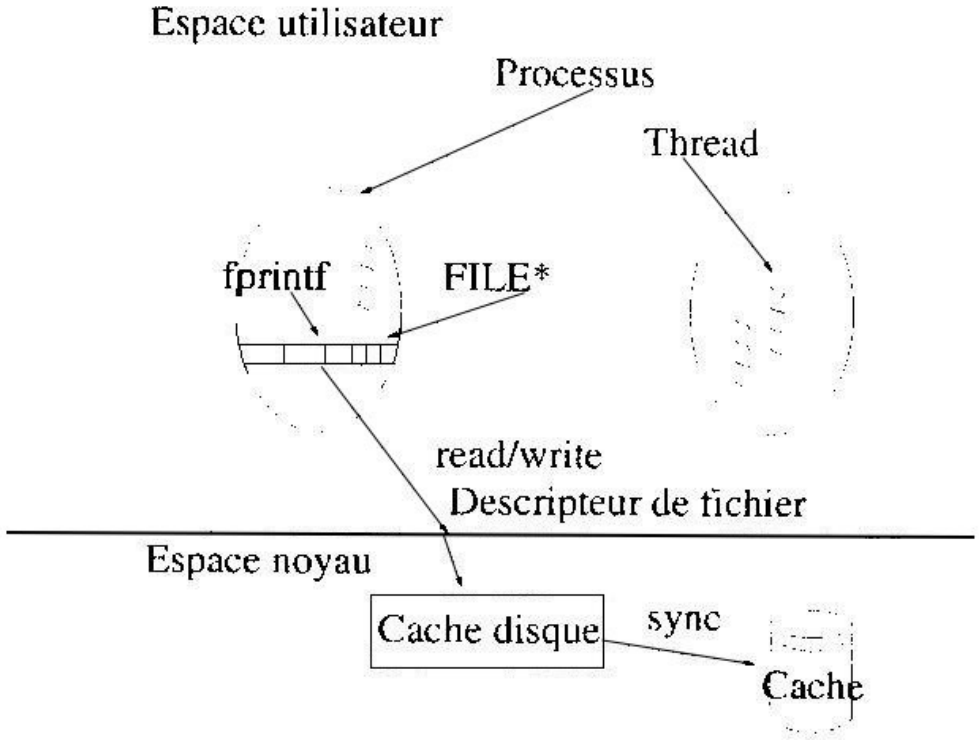
Reception signal 35, capacite = 1 depassee

Debloquee par timer : echeance ratee ..

Sous partie 2.5

E/S pour le temps réel

E/S pour le temps réel (1)



• Problèmes liés aux E/S sur UNIX :

- Les E/S ne sont pas synchrones : cache disque.

- Efficace mais non déterminisme temporel. Ordre d'exécution des E/S non spécifié => généralement FIFO.

- E/S non bloquant par *ioctl :* non standard et cher => recouvrement du temps de blocage.

- Ressources cachées : agencement des données sur le disque? algorithme d'accès (EDF, SCAN, SSTF) ? préallocation de zones contiguës ?

E/S pour le temps réel (2)

• Solutions POSIX 1003.1b :

- Mapping des périphériques en mémoire => sécurité, portabilité.

- Opération de vidage de cache ; séparation entre les données et le bloc de contrôle : primitives *fsync* et *fdatasync*.

- Configuration à l'ouverture des fichiers : options *O_DSYNC* et *O_SYNC*.

- E/S asynchrones => parallélisme sur les lectures/écritures => threads ? ?

E/S pour le temps réel (3)

• Les fonctions :

aio_write	Emission d'une écriture asynchrone.
aio_read	Emission d'une lecture asynchrone.
aio_error	Consultation du résultat d'une E/S.
aio_return	Retourne le nombre de caractères lus ou écrits.
aio_cancel	Annulation d'une E/S.
aio_suspend	Bloque le processus jusqu'à terminaison d'une E/S.
aio_ fsync	Bloque jusqu'à terminaison des E/S en cours.
lio_listio	Emission d'un ensemble d'E/S.

E/S pour le temps réel (4)

• Exemple :

```
int main(int argc, char * argv[])
{
char msg[100];
int fd;
struct sigaction sig;
struct aiocb io;
     fd=open("fic.txt",O_SYNC | O_WRONLY);
     sig.sa_flags=SA_SIGINFO;
     sig.sa_sigaction=io_terminee;
     sigemptyset(&sig.sa_mask);
     sigaction(SIGRTMIN,&sig,NULL);
     strcpy(msg,"Hello world !");
     io.aio_buf=msg;
     io.aio_offset=0;
     io.aio_fildes=fd;
     io.aio_nbytes=strlen(msg);
     io.aio_reqprio=0;
     io.aio_sigevent.sigev_notify=SIGEV_SIGNAL;
     io.aio_sigevent.sigev_value.sival_ptr=&io;
     io.aio_sigevent.sigev_signo=SIGRTMIN;
     aio_write(&io);
     printf("Emission de l'ecriture\n");
     while(1);
}
```

E/S pour le temps réel (5)

• Exemple (suite)

```
void io_terminee(int sig, siginfo_t *info, void *uap)
{
     struct aiocb* io=(struct aiocb*)info->si_value.sival_ptr;
     printf("E/S terminee sur le descripteur %d\n", io->aio_fildes);
     if(aio_error(io)!=EINPROGRESS)
          printf("Nb caracteres ecrits %d\n", aio_return(io));
     exit(0);
}
```

- Exécution :

$aio

Emission de l'ecriture

E/S terminee sur le descripteur 3

Nb caracteres ecrits : 12

$cat fic.txt

Hello world!

Sous partie 2.6

Les files de messages

Les files de messages (1)

• Dans UNIX, le mécanisme de communication classique est le pipe (nommé ou non) => peu adapté au temps réel car trop abstrait.

• Les files de messages POSIX 1003.1b ont les mêmes fonctionnalités mais :

- Priorité à l'émission/réception.

- Fonctionnement éventuellement non bloquant.

- Préallocation éventuelle des ressources.

- Communication inter-processus et inter-thread.

- Hélas, pas de prise en compte des phénomènes d'inversion de priorités.

• Une file de messages est caractérisée par un nom symbolique et éventuellement configurée grâce à une structure de type *mq_attr*.

Les files de messages (2)

• Les fonctions :

mq_open	Création ou connexion à une file.
mq_unlink	Destruction d'une file.
mq_receive	Réception du message le plus ancien et le plus prioritaire.
mq_send	Emission d'un message avec une priorité donnée.
mq_close	Déconnexion d'une file.
mq_notify	Notification de la réception d'un message.
mq_setattr	Positionne les attributs.
mq_getattr	Consulte les attributs.

Les files de messages (3)

• Exemple :

```
#include <pthread.h>
#include <mqueue.h>
int main(int argc, char* argv[])
{
pthread_t tid; mqd_t id;
char buff [100]; struct mq_attr attr;
     attr.mq_maxmsg=100;
     attr.mq_flags=0;
     attr.mq_msgsize=100;
     id=mq_open("/mafile",O_CREAT|O_WRONLY,444,&attr);
     strcpy(buff,"coucou");
     /* Emission avec la priorite 1 (de 0 a MQ_PRIO_MAX) */
     mq_send(id,buff,100,1);
     pthread_create(&tid,NULL,consommateur,NULL);
     pthread_exit(NULL);
}
```

Les files de messages (4)

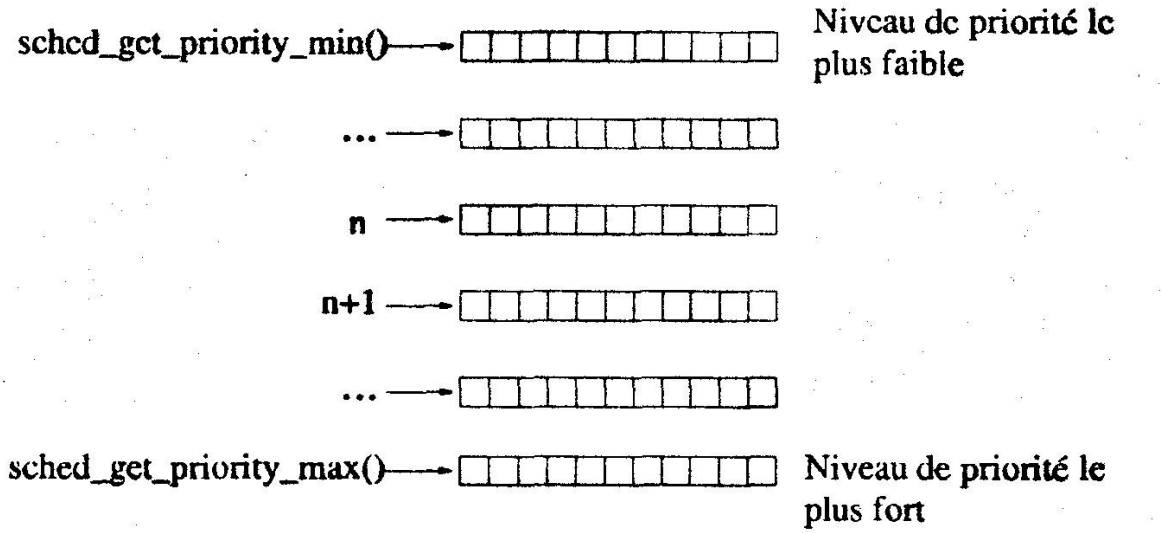
• Exemple (suite) :

```
void* consommateur(void* arg) {
mqd_t id;
char buff [100] ;
     id=mq_open("/mafile",O_RDONLY);
     mq_receive(id,buff,100,NULL);
     printf("msg = %s\n",buff);
     mq_unlink("/mafile");
     pthread_exit(NULL);
}
```

Sous partie 2.7

Services d'ordonnancement.

Services d'ordonnancement (1)



• Caractéristiques :

- Application au niveau des threads et des processus.

- Priorités fixes, préemptif => RM facile. Doit offrir au minimum 32 niveaux de priorité.

- Une file d'attente par priorité + politiques de gestion de la file *(SCHED_FIFO, SCHED_RR, SCHED_OTHERS).*

- Services accessibles à l'utilisateur privilégié.

• La norme précise que la politiques d'ordonnancement doit s'appliquer partout ou un choix de processus/thread s'effectue (ex : choix d'un processus/thread à la libération d'un sémaphore).

Services d'ordonnancement (2)

• Politiques POSIX 1003.1b :

#define SCHED_OTHER 0

#define SCHED_FIFO 1

#define SCHED_RR 2

• Paramètre(s) : extensible pour les politiques à venir

```
struct sched_param
{
        int sched_priority;
        ….
} ;
```

• Modification des paramètres :

1. Thread : à la création du thread à l'aide d'un attribut ou modification en cours de vie du thread.

2. Par héritage lors d'un *fork() ou* modification en cours de vie du processus.

Services d'ordonnancement (3)

• Les fonctions :

sched_get_priority_max	Consulte la valeur de la priorité maximale.
sched_get_priority_min	Consulte la valeur de la priorité minimale.
sched_rr_ get_interval	Consulte la valeur du quantum.
sched_ yield	Libère le processeur.
sched_ setscheduler	Positionne la politique d'ordonnancement.
sched_getscheduler	Consulte la politique d'ordonnancement.
sched_setparam	Positionne la priorité.
sched_getparam	Consulte la priorité.
pthread_setschedparam	Positionne la priorité.
pthread_getschedparam	Consulte la priorité.

• *NOTE :* Les deux dernières fonctions s'appliquent sur un thread, les autres sur un processus et/ou thread.

Services d'ordonnancement (4)

• Exemple : modification des paramètres de 2 processus

```
struct sched_param parm;
int res=-1;
….
      /* Tache T1 ; P1=10 */
      parm.sched_priority=l5;
      res=sched_setscheduler(pid_T1,SCHED_FIFO,&parm);
      if  (res<0)  perror("sched_setscheduler tache T1");
      /* Tache T2 ; P2=30 */
      parm.sched_priority=l0;
      res=sched_setscheduler(pid_T2,SCHED_FIFO,&parm);
      if (res<0) perror("sched_setscheduler tache T2");
```

Sous partie 2.8

La gestion mémoire

La gestion mémoire

• Dans un système temps partagé : indéterminisme temporel possible car :

- Allocation mémoire dynamique.

- Pagination : *swapin/swapout.*

=> Solution : limiter les allocations dynamiques et verrouiller les pages en mémoire centrale.

• Interface POSIX 1003.1b

- *mlockall()/munlockall() :* verrouille/déverrouille toutes les pages d'un processus.

- *mlock()/munlock() :* verrouille/déverrouille une plage d'adresses.

! Attention : *mlock()* est peu portable (car il n'y a pas de norme sur le modèle mémoire dans POSIX).

Partie 3

Résumé

Résumé (1)

• Caractéristiques :

- Cible les applications ayant des contraintes temporelles non strictes.

- Cohabitation avec applications temps partagé et interaction avec services "évolués" (BD, IHM, etc).

• Abstractions identiques à celles d'un système temps réel embarqué mais différences sur :

- Les garanties offertes (niveau de préemptivité, inversion de priorité possible, précision sur les services qui manipulent le temps, etc).

- L'allocation des ressources qui est parfois cachée.

- Services très souvent sans timeout.

- Plus flexible.

Résumé (2)

• Deux approches :

1. Extensions POSIX. Présent à la fois sur des UNIX temps partagé et sur les systèmes temps réel embarqués.

2. Extensions UNIX proposées par SVR4 (plus limitées et moins portables).

• POSIX 1003.1b et POSIX 1003.1c :

- Chapitres optionnels de la norme POSIX. Objectif : définir des extensions temps réel portables d'un UNIX à un autre.
- Composition : threads et outils de synchronisation/communications, ordonnancement, timers, E/S et signaux adaptées.
- Portabilitée limitée mais abstractions riches.

Partie 4

Références

[AND 92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy.

« Scheduler Activations : Effective Kernel Support for the User-Level Management of Parallelism ». *ACM Transactions on Computer Systems,* 10(1) :53-79, February 1992.

[DEM 94] I. Demeure and J. Farhat.

« Systèmes de processus légers : concepts et exemples ». *Technique et Science Informatiques,* 13(6) :765-795, décembre 1994.

[DEN 66] B. Dennis and E. C. Van Horn.

« Programming Semantics for Multiprogrammed Computations ». *Communications of the ACM,* 9(3) :143-155, March 1966.

[GAL 95] B. O. Gallmcister. *POSIX 4 : Programming for the Real World* . O'Reilly and Associates, January 1995.

[GHO 94] K. Ghosh, B. Mukherjee, and K. Schwan.

« A survey of Real Time Operating Systems ». Technical Report, College of Computing. Georgia Institute of Technology. Report GIT-CC-93/18, February 1994.

[GOV 91] R. Govindan and D. P. Anderson.

« Scheduling and IPC Mechanisms for Continuous Media ». pages 68-80. 13th ACM Symposium on Operating Systems Principles, October 1991.

[JMR 93] J. M. Rifflet. *La programmtion sous UNIX.* Addison-Wesley, 3rd edition, 1993.

[RIF 95] J. M. Rifflet. *La communication sous UNIX : applications réparties.* Ediscience International, 2rd edition, 1995.

[VAH 96] U. Vahalia. *UNIX Internals : the new frontiers.* Prentice Hall, 1996.