Master Informatique **M1** - Spécialité SAR
UE POSIX

TEMPS RÉEL

I – Ordonnancement dynamique
II – Signaux en temps réel
III – E/S asynchrones
IV – Gestion fine du temps

Système Temps Réel : Principes

Définition d'un système Temps Réel (TR)

Système dont le résultat dépend à la fois :
- de l'exactitude des calculs
- et du temps mis à produire les résultats

Notion d'échéance

Contrainte de temps bornant l'occurence d'un événement (production de résultat, envoi de signal, ...)

"*Introduction aux systèmes temps réel*", C. Bonnet & I. Demeure, Ed. Hermès/Lavoisier 1999

Problématiques Temps Réel

- Garantir qu'un événement se produit à une date donnée, ou avant une échéance donnée
  - Contraintes périodiques : le service doit être rendu selon un certain rythme
    eg. toutes les X ms
  - Contraintes ponctuelles : lorsque l'évt Y se produit, il doit être traité dans un tps limité
- Garantir qu'un évt A se produit avant un évt B
- Garantir qu'aucun évt externe à l'application TR ne retardera les pcs importants
- Garantir qu'un ordonnancement entre plusieurs tâches est effectivement possible

UNIX System V & Temps réel

Certaines caractéristiques de SysV empêchent une bonne gestion temps réel
- Gestion très basique des priorités
- Ordonnancement "round robin" pas toujours adapté
- Horloges et gestion du temps très sommaires
- E/S bloquantes

Il existe une spécif TR dans System V Release 4 (SVR4)
- Non portable
- Complexe
- Gestion du temps reste sommaire et les E/S synchrones

POSIX Temps Réel

POSIX.1b (POSIX.4) : POSIX Real-Time Scheduling Interfaces

- Gestion dynamique de l'ordonnancement
- Horloges et temporisateurs évolués
- Ajout d'E/S asynchrones
- Signaux TR
- Eléments déjà présentés dans ce cours
  threads, sémaphores, partage mémoire

Ordonnancement POSIX.1b (POSIX.4)

- **Politiques d'ordonnancement**
  - L'ordonnanceur est la partie du noyau qui décide quel processus prêt va être exécuté ensuite.
    - Il peut y avoir politiques différentes :
      - processus classiques, processus des applications à vocation temps-réel.
      - Une valeur de **priorité statique** *sched_priority* est assignée à chaque processus, et ne peut être modifiée que par l'intermédiaire d'appels systèmes.
        - Linux: 0 – 99

Ordonnancement POSIX.1b (POSIX.4)

`_POSIX_PRIORITY_SCHEDULING` doit être positionné dans `<unistd.h>`

Définitions fournies dans `<sched.h>`

```
struct sched param {
       int    sched_priority;
};

int    sched_get_priority_max(int);
int    sched_get_priority_min(int);
int    sched_getparam(pid_t, struct sched_param *);
int    sched_getscheduler(pid_t);
int    sched_rr_get_interval(pid_t, struct timespec *);
int    sched_setparam(pid_t, const struct sched_param *);
int    sched_setscheduler(pid_t, int, const struct sched_param *);
int    sched_yield(void)
```


Ordonnancement POSIX.1b (POSIX.4)

Politiques d'ordonnancement

Définissent le choix du processus à exécuter sur un processeur

Une seule politique en vigueur par processus

Politiques :

- `SCHED_FIFO`
- `SCHED_RR`
- `SCHED_OTHER`

- Les processus ordonnancés avec *SCHED_OTHER* doivent avoir une priorité statique de 0.
- Ceux ordonnancés par *SCHED_FIFO* ou *SCHED_RR* ont une priorité statique supérieur:
  - Exemple : Linux - dans l'intervalle 1 à 99.

Ordonnancement POSIX.4

Politiques d'ordonnancement - `SCHED_FIFO`

Une queue par niveau de priorité

Le processus exécuté est celui :
- dans la queue de plus haute priorité
- dont la date d'introduction est la + ancienne

Un processus rend le processeur :
- s'il est bloqué en attente d'E/S
- s'il le demande explicitement (cf. `sched_yield`)
- s'il est préempté par un processus de plus haute priorité
- s'il est terminé

8

Ordonnancement POSIX.4

- Politiques d'ordonnancement - `SCHED_FIFO '(cont.)`
  - **SCHED_FIFO** ne peut être utilisé qu'avec des priorités statiques supérieures à 0.
    - **SCHED_FIFO** est un ordonnancement simple sans tranches de temps.
    - Un processus **SCHED_FIFO** qui a été préempté par un autre processus de priorité supérieure restera en tête de sa liste et reprendra son exécution dès que tous les processus de priorités supérieures sont à nouveau bloqués.
    - Quand un processus **SCHED_FIFO** devient prêt, il est inséré à la fin de sa liste.
    - Un processus appelant **sched_yield( )** sera placé à la fin de sa liste.
      - Aucun autre événement ne modifiera l'ordre des listes de priorités statiques égales avec **SCHED_FIFO**.
      - Un processus **SCHED_FIFO** s'exécute jusqu'à ce qu'il soit bloqué par une opération d'entrée/sortie, qu'il soit préempté par un processus de priorité supérieure, ou qu'il appelle **sched_yield** ().

9

Ordonnancement POSIX.4

Politiques d'ordonnancement - `SCHED_RR`

FIFO augmentée d'un partage du temps processeur entre processus de même priorité
Addition d'un quantum de temps
- défini par le système
- pas nécessairement une constante
- récupérable pour chaque processus (cf. `sched_rr_get_interval`)

En + des autres contraintes, un processus doit rendre la main à la fin de son quantum
S'il n'est pas terminé, un processus retourne à la fin de sa queue en rendant la main

Tout ce qui est décrit pour **SCHED_FIFO** s'applique aussi à **SCHED_RR**, sauf que chaque processus ne dispose que d'une tranche temporelle limitée pour son exécution.

10

Ordonnancement POSIX.4

Politiques d'ordonnancement - `SCHED_OTHER`

- Défini par l'implémentation
  - Défaut : La politique standard de temps partagé « round-robin »
    - Pour tous les processus ne réclamant pas de fonctionnalités temps-réel
- La politique **SCHED_OTHER** ne peut être utilisée qu'avec des priorités statiques à 0.

-Le processus à exécuter est choisi dans la liste des processus de priorités statiques nulles, en utilisant une priorité dynamique qui ne s'applique que dans cette liste.

* La priorité dynamique est basée sur la valeur de « politesse » (nice) et est incrémentée à chaque time quantum où le processus est prêt mais non sélectionné par l'ordonnanceur. Ceci garantit une progression équitable de tous les processus **SCHED_OTHER**.

11

Ordonnancement – SCHED OTHER

Être gentil (nice)...

Pas de standard régissant l'ordonnancement des processus UNIX

Un seul recours pour l'utilisateur : indiquer des priorités d'exécution (change la priorité dynamique)

```
int nice(int incr);
```

Renvoie -1 en cas d'échec, 0 sinon
La priorité du processus est modifiée à *valeur_courante* + `incr`
La priorité par défaut d'un processus est de 0
Seul le super-utilisateur peut spécifier `incr` < 0

Ordonnancement POSIX.1b (POSIX.4)

Modification/Récupération de la politique d'ordonnancement

```
int sched_setscheduler(pid_t pcs, int pol, const struct sched_param *p);
```

Retourne -1 en cas d'échec, la valeur de la politique précédente sinon
– *pcs* processus concerné par la modification (0 => processus appelant)
– *p* paramètres d'ordonnancement à associer au processus dans la nouvelle politique
– *pol* nouvelle politique à mettre en place
(`SCHED_FIFO, SCHED_RR, SCHED_SPORADIC*, SCHED_OTHER`)

```
int sched_getscheduler(pid_t pcs);
```

Retourne la valeur de la politique en vigueur, -1 en cas d'échec
– *pcs* processus concerné (0 => processus appelant)

* SCHED_SPORADIC : La politique de base est la politique SCHED_FIFO, avec une quantité de temps alloué à un serveur dont la capacité est cette quantité

Ordonnancement POSIX.1b (POSIX.4)

Priorités de processus

La priorité statique par défaut d'un processus est à 0

En cas de `fork`, le processus fils hérite de la priorité de son père

Les priorités minimales et maximales dépendent de la politique en vigueur

```
int sched_get_priority_max(int pol);
int sched_get_priority_min(int pol);
```

– Retournent la valeur max/min en vigueur pour la politique concernée,
-1 en cas d'échec
– *pol* la politique d'ordonnancement dont on cherche à connaître les bornes de priorité

Exemple

```
#include <sched.h>

…

struct sched_param sp;
int politique;

…

if (politique = sched_getscheduler(getpid()) == -1)
   exit(1);
if (politique != SCHED_OTHER) {
   sp.sched_priority = 12 + sched_get_priority_min(SCHED_FIFO);
   if (sched_setscheduler(0, SCHED_FIFO, &sp) == -1)
      exit(2);
}

…
```

Ordonnancement POSIX.4

Priorités de processus (suite)

La priorité statique peut être modifiable pour chaque processus

```
int sched setparam(pid t pcs, const struct sched param *p);
```

– Retourne -1 en cas d'échec, 0 sinon
– *pcs* le processus dont on veut modifier la priorité
– *p* les paramètres contenant la nouvelle priorité

Pour connaître la priorité associée à un processus

```
int sched_getparam(pid_t pcs, struct sched_param *p);
```


Exemple

```
#include <sched.h>

…

struct sched_param sp;
int politique;

…

if (politique = sched_getscheduler(getpid()) == -1) exit(1);
if (politique == SCHED_RR) {
   if (sched_getparam(0, &sp) == -1) exit (2);
   sp.sched_priority += 12;
   sched_setparam(0, &sp);
}

…
```


Ordonnancement POSIX.4

Effet d'une modification sur l'ordonnancement

Changer la politique d'ordonnancement ou la priorité d'un processus place celui-ci automatiquement en fin de queue

"Préemption immédiate"
A change la priorité d'un autre processus B
La priorité de B devient + haute que celle de A
=> A est interrompu au profit de B avant même le retour de l'appel !

(Mauvais) Exemple

```
#include <signal.h>
#include <sched.h>
struct sched_param sp;

…

sched_getparams(0, &sp);
sp.sched_priority = 12 + sched_get_priority_min(SCHED_FIFO);
sched_setscheduler(0, SCHED_FIFO, &sp);

int pid  = fork();
if (pid) {
     sp.sched_priority++;
     sched_setparam(pid, &sp);
     kill (pid, sigint);
} else {
     printf(" CE PROGRAMME NE SE TERMINERA JAMAIS ! HA ! HA ! HA!\n");
     pause();
}

…
```

Ordonnancement POSIX.1b (POSIX.4)

Rendre explicitement (et gracieusement) la main

```
int sched_yield(void);
```

– Retourne -1 en cas d'échec, 0 sinon
– N'affecte que le processus appelant => retour en fin de queue

Exemple

```
int pid  = fork();
if (pid) {
     sched_yield();
     printf("Pere\n");
} else {
     printf("Fils\n");
}
```

Question : En FIFO, quel affichage aura lieu en premier ? Et si on retire le `sched_yield` ?

Important : `sched_yield` ne sert pas à synchroniser les processus entre eux ===> SÉMAPHORES

Signaux Temps Réel

Déficiences des signaux UNIX vis-à-vis du TR

Pas de priorités entre signaux

Information transmise est limitée à la valeur du signal

Pas de correspondance événement ⇔ notification
Réception de signal écrase tout signal pendant de même valeur

Manque de signaux réservés à l'utilisateur
Seulement 2 signaux : SIGUSR1 et SIGUSR2

Solution POSIX.1b (POSIX 4) : file de signaux

Signaux Temps Réel

Caractéristiques principales

Extension des signaux existants (`SIGRTMIN >> SIGRTMAX`)

Signaux TR peuvent être placés dans des files
Dépend de l'implémentation (flag `SA_SIGINFO`)
=> garantit l'ordre de délivrance

L'événement déclencheur est connu
(envoi par un pcs, échéance de temporisateur, fin d'aio, ...)

Des données supplémentaires peuvent être jointes à l'envoi

Signaux Temps Réel

Ordre de délivrance

Pour des signaux de même valeur (si l'implém. le permet)
priorité = ordre de réception

Pour des signaux de valeurs différentes
priorité = valeur du signal
valeur la plus faible => priorité la plus forte
eg. `SIGRTMIN >> SIGRTMAX`

La spécif **n'impose pas d'ordre** entre les signaux TR et les signaux système
eg. `SIGINT <??> SIGRTMIN`

Signaux Temps Réel

Opérations associées aux signaux TR

Tous les appels associés aux signaux système (`kill`, `sigprocmask`, `alarm`, `sigsuspend`, ...) restent valables avec les signaux TR

Viennent s'ajouter des opérations avec une sémantique orientée TR :

```
int sigqueue(pid_t pid, int signo, const union sigval value);

int sigwait(const sigset_t *restrict set, int *restrict sig);

int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *restrict info);

int sigtimedwait(const sigset_t *restrict set,
                siginfo_t *restrict info,
                const struct timespec *restrict timeout);
```


Signaux Temps Réel

Envoi de signal TR

```
int sigqueue(pid_t pid, int signo, const union sigval value);
```

pid	pid du pcs destinataire (pas de "broadcast" similaire au `kill` !)
signo	numéro du signal à envoyer
value	donnée supplémentaire associée à l'événement

```
union sigval {
    int sival_int;
    void *sival_ptr;
};
```

Le signal est placé dans une file de signaux à trois conditions :

1. L'implém système autorise les files de signaux
2. `SIGRTMIN` ≤ `signo` ≤ `SIGRTMAX`
3. Le destinataire a validé l'insertion dans une file pour ce signal
   ie . signal associé à une `struct sigaction` avec `sa_flags = SA_SIGINFO` et une fonction définie pour le champs `sa_sigaction`

Renvoie -1 si échec, 0 sinon

Signaux Temps Réel

Réception de signal TR

```
int sigwait(const sigset_t *restrict set, int *restrict sig);
```

set	masque des signaux attendus
sig	numéro du signal délivré

Le processus appelant est bloqué en attente de signaux inclus dans `set`
(sauf si des signaux inclus dans `set` sont déjà pendants avant l'appel)

Le signal est retiré des signaux pendants, sauf si les trois conditions suivantes sont remplies :

1. L'implém système autorise les files de signaux
2. `SIGRTMIN` ≤ `sig` ≤ `SIGRTMAX`
3. D'autres instances du même signal (même valeur) n'ont pas encore été délivrées

Renvoie -1 si échec, 0 sinon

Signaux Temps Réel : Exemple

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
#define _POSIX_SOURCE 1

void interrupt_signal(int signo, siginfo_t *si, void *context){}

int main() {

    int mysig;
    union sigval val;
    sigset_t block_mask;
    struct sigaction action;

    action.sa_sigaction = interrupt_signal;
    action.sa_flags = SA_SIGINFO;
    sigfillset(&block_mask);
    action.sa_mask = block_mask;
    sigaction(SIGRTMIN, &action, 0);
    sigprocmask(SIG_SETMASK, &block_mask, 0);          /* ../.. */
```

Signaux Temps Réel : Exemple (suite)

```
    if (fork()) {    /* Pere */
        sigemptyset(&block_mask);
        sigaddset(&block_mask, SIGRTMIN);
        if (sigwait(&block_mask, &mysig) == -1) {
            perror("sigwait");
            exit(1);
        }
        printf("Valeur sig - %d\n", mysig);
    } else {    /* Fils */
        val.sival_int = 4;
        if (sigqueue(getppid(), SIGRTMIN, val) == -1) {
            perror("sigqueue");
            exit(1);
        }
        exit(0);
    }
    wait(0);
    printf("fin prog\n");
    return 0;
}
```


Signaux Temps Réel

Données associées à un signal TR

```
#include <sys/siginfo.h>

typedef struct {
    int si_signo;                   /* numero du signal */
    int si_code;                    /* source du signal */
    union sigval si_value;          /* donnee associee */
    int si_errno;                   /* errno (notif d'echec) */
    pid_t si_pid;                   /* pid de l'emetteur */
    uid_t si_uid;                   /* uid de l'emetteur */
    void *si_addr;                  /* @ de faute (notif d'echec) */
    int si_status;                  /* valeur de terminaison */
    int si_band;                    /* retour d'E/S ou poll */
} siginfo_t;
```

(en gras les champs pérennes)

Signaux Temps Réel : Exemple (re-suite)

```
/* ../.. */

void interrupt_signal(int signo, siginfo_t *si, void *context){

    printf("Valeur sig - %d\n", signo);

    if (si->si code = SI USER) {
        printf("Interruption utilisateur\n");
        printf("Valeur sig (le retour) - %d\n", si->si_signo);
        printf("PID emetteur - %d\n", si->si_pid);
        printf("Valeur associee - %d\n", si->si_value);
    }

}

/* ../.. */
```


Signaux Temps Réel

Réception de signal TR – opérations enrichies

```
int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *restrict info);
```

set masque des signaux attendus
info données associées au signal

```
int sigtimedwait(const sigset_t *restrict set,
                 siginfo_t *restrict info,
                 const struct timespec *restrict timeout);
```

timeout échéance du temporisateur

Renvoient -1 si échec, 0 sinon

Signaux Temps Réel

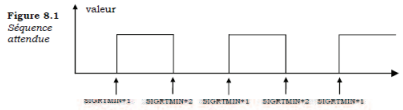

Figure 8.1 *Séquence attendue*

Malheureusement, si les signaux sont bloqués pendant un moment, ils ne seront pas délivrés dans l'ordre d'arrivée, mais en fonction de leur priorité. Toutes les impulsions SIGRTMIN+1 sont délivrées d'abord, puis toutes les impulsions SIGRTMIN+2.

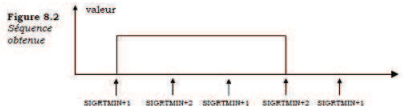

Figure 8.2 *Séquence obtenue*

Signaux Temps Réel – Exemple Prioirité

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int signaux_arrives [10];
int valeur_arrivee [10];
int nb_signaux = 0;

void
gestionnaire_signal_temps_reel (int numero,
                siginfo_t * info, void * inutile)
{
  signaux_arrives [nb_signaux] = numero - SIGRTMIN;
  valeur_arrivee [nb_signaux] = info -> si_value . sival_int;
  nb_signaux ++;
}


  void
envoie_signal_temps_reel (int numero, int valeur)
{
  union sigval valeur_sig;

  fprintf (stdout, "Envoi signal SIRTMIN+%d, valeur %d\n",
                    numero, valeur);
  valeur_sig . sival_int = valeur;
  if (sigqueue (getpid( ) , numero + SIGRTMIN, valeur_sig) < 0) {
     perror ("sigqueue");
     exit (1);
  }
}
```

C. Blaess

Signaux Temps Réel – Exemple priorité

```
   int
main (void)
{
   struct sigaction action;
   sigset_t ensemble;
   int i;

   fprintf (stdout, "Installation gestionnaires de signaux \n");
   action . sa_sigaction = gestionnaire_signaltemps réel;
   sigemptyset (& action . sa_mask);
   action . sa_flags = SA_SIGINFO;
   if ((sigaction (SIGRTMIN + 1, & action, NULL) < 0)
      || (sigaction (SIGRTMIN + 2, & action, NULL) < 0)
      || (sigaction (SIGRTMIN + 3, & action, NULL) < 0)) {
     perror ("sigaction");
     exit (1);
   }
   fprintf (stdout, "Blocage de tous les signaux \n");
   sigfillset(& ensemble);
   sigprocmask (SIG_BLOCK, & ensemble, NULL);
   envoie_signal_tempsreel (1, 0);
   envoie_signal_temps_reel (2, 1);
   envoie_signal_temps_reel (3, 2);
   envoie_signal_temps_reel (1, 3);
   envoie_signal_temps_reel (2, 4);
   envoie_signal_temps_reel (3, 5);
   envoie_signal_temps_reel (3, 6);
   envoie_signal_temps_reel (2, 7);
   envoie_signal_temps_reel (1, 8);
   envoie_signal_temps_reel (3, 9);

   fprintf (stdout, "Déblocage de tous les signaux \n");
   sigfillset(& ensemble);
   sigprocmask (SIG_UNBLOCK, & ensemble, NULL);

   fprintf (stdout, "Affichage des résultats \n");

   for (i = 0; i < nb_signaux; i++)
   fprintf (stdout, "Signal SIGRTMIN+%d, valeur %d\n",
            signaux_arrives [i], valeur_arrivee [i]);

   fprintf (stdout, "Fin du programme \n");
   return (0);
}
```

```
$ ./exemple_sigqueue_1

Installation gestionnaires de signaux
Blocage de tous les signaux
Envoi signal SIRTMIN+1, valeur 0
Envoi signal SIRTMIN+2, valeur 1
Envoi signal SIRTMIN+3, valeur 2
Envoi signal SIRTMIN+1, valeur 3
Envoi signal SIRTMIN+2, valeur 4
Envoi signal SIRTMIN+3, valeur 5
Envoi signal SIRTMIN+3, valeur 6
Envoi signal SIRTMIN+2, valeur 7
Envoi signal SIRTMIN+1, valeur 8
Envoi signal SIRTMIN+3, valeur 9
Déblocage de tous les signaux
Affichage des résultats
Signal SIGRTMIN+1, valeur 0
Signal SIGRTMIN+1, valeur 3
Signal SIGRTMIN+1, valeur 8
Signal SIGRTMIN+2, valeur 1
Signal SIGRTMIN+2, valeur 4
Signal SIGRTMIN+2, valeur 7
Signal SIGRTMIN+3, valeur 2
Signal SIGRTMIN+3, valeur 5
Signal SIGRTMIN+3, valeur 6
Signal SIGRTMIN+3, valeur 9
Fin du programme
$
```

C. Blaess

Signaux Temps Réel – Exemple sigwaitinfo

```
void
gestionnaire (int numero, struct siginfo * info, void * inutile)
{
   fprintf (stderr, "gestionnaire : %d reçu \n", numero);
}

   int
main (void)
{
   sigset_t ensemble;
   int numero; struct sigaction action;
   fprintf (stderr, "PID=%u\n", getpid( ));
   /* installation gestionnaire pour SIGRTMIN+1 */
   action . sa_sigaction = gestionnaire;
   action . sa_flags = SA_SIGINFO;
   sigemptyset (& action. sa_mask);
   sigaction (SIGRTMIN + 1, & action, NULL);
   /* Blocage de tous les signaux sauf SIGRTMIN+1 */
   sigfillset (& ensemble);
   sigdelset (& ensemble, SIGRTMIN + 1);
   sigprocmask (SIG_BLOCK, & ensemble, NULL);
   /* Attente de tous les signaux sauf RTMIN+1 et SIGKILL */
   sigfillset (& ensemble);
   sigdelset (& ensemble, SIGRTMIN + 1);
   sigdelset (& ensemble, SIGKILL);
   while (1) {
      if ((numero = sigwaitinfo (& ensemble, NULL)) < 0)
         perror ("sigwaitinfo");
      else
         fprintf (stderr, "sigwaitinfo : %d reçu \n", numero);
   }
   return (0);
}
```

pid = 1435

```
^                                  $ kill -TERM 1435
reçu sigwaitinfo : 15 reçu
                                   $ kill -33 1435
gestionnaire : 33 reçu
sigwaitinfo: Appel système interrompu
                                   $ kill -STOP 1435
sigwaitinfo : 19 reçu
```

SIGTRMIN+1 (33) n'est pas attendu

C. Blaess

E/S asynchrones

Déficiences des E/S UNIX vis-à-vis du TR

Synchrones
- Elles peuvent être bloquantes

Non synchronisées
- Phénomènes de bufferisation implicite
- *eg. le processus considère son E/S terminée alors que l'E/S sur disque est en cours*

Atomiques
- 1 E/S = 1 appel système !

E/S asynchrones

Synchronisation mémoire/support stable

2 remèdes POSIX.1 déjà rencontrés
- ⇒ flag `O_SYNC` ou fonction `fsync`

Solution POSIX.4 : 3 flags distincts
- `O_DSYNC` mise à jour du disque à chaque écriture
- `O_SYNC` `O_DSYNC` + mise à jour de l'inode à chaque écriture
- `O_RSYNC` mise à jour de l'inode à chaque lecture

E/S asynchrones

Synchronie des E/S

Remède POSIX.1 déjà rencontré
- ⇒ flag `O_NONBLOCK`

Solution POSIX.4 : Fonctions d'E/S asynchrones (**AIO**)

```
#include <aio.h>
int aio_read(struct aiocb *);
int aio_write(struct aiocb *);
ssize_t aio_return(struct aiocb *);
int aio_error(const struct aiocb *);
int lio_listio(int, struct aiocb *const[], int,
                  struct sigevent *);
int aio_suspend (const struct aiocb *const[], int,
                  const struct timespec *);
int aio_fsync(int, struct aiocb *);
int aio_cancel(int, struct aiocb *);
```


E/S asynchrones

Bloc de contrôle pour E/S asynchrones

```
struct aiocb {
     int             aio_fildes          /* file descriptor */
     off_t           aio_offset          /* file offset */
     volatile void*  aio_buf             /* location of buffer */
     size_t          aio_nbytes          /* length of transfer */
     int             aio_reqprio         /* request priority */
     struct sigevent aio_sigevent        /* signal number and value or thread
     */
     int             aio_lio_opcode      /* operation to be performed:
                                            LIO_READ, LIO_WRITE, LIO_NOP */
};
```

Regroupe l'ensemble des paramètres pour une E/S classique :
- un descripteur de fichier (`aio_fildes`), un pointeur vers un buffer (`aio_buf`)
- et un nombre d'octets à transférer (`aio_nbytes`)

Rajoute des éléments importants :
- un positionnement de curseur explicite pour chaque E/S (`aio_offset`)
- une gestion de priorité entre E/S (`aio_reqprio`)
- la possibilité de spécifier un événement à déclencher en fin d'E/S (`aio_sigevent`)
- la possibilité de lister plusieurs E/S dans un même appel (`aio_lio_opcode`)

E/S Asynchrone

Structure de notification

façon dont un processus sera averti d'un évènement (par exemple la fin d'une requête asynchrone

```
union sigval {
      int sival_int;  void *sival_ptr;
};

struct sigevent {
    int sigev_notify;     // SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD)
    int sigev_signo;            //Signal number
    union sigval sigev_value;    //Notif. data
    void (*)(union sigval) sigev_notify_function   //Thread function
    void *sigev_notify_attributes; /* Thread function attributes */
};
```


E/S Asynchrone

Le champ *sigev_notify* indique comment les notifications seront effectuées. Ce champ peut prendre une des valeurs suivantes :

- **SIGEV_NONE** Une notification « vide » : ne fait rien quand l'évènement se produit.
- **SIGEV_SIGNAL** Notifie le processus en envoyant le signal indiqué en *sigev_signo*.
- **SIGEV_THREAD** Notifie le processus par l'appel de *sigev_notify_function* « comme s'il » s'agissait de la fonction de démarrage d'un nouveau thread

Les mêmes renseignements sont aussi disponibles si le signal permet d'utiliser **sigwaitinfo**.

Exemple

```
#include <aio.h>

…

char c = 'X';
int fd = open("toto", O_WRONLY, 0600);
struct aiocb a;
a.aio_filedes = fd;
a.aoi_buf = &c;
a.aio_nbytes = 1;
a.aio_offset = 0;
a.aio_reqprio = 0;
a.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
a.aio_sigevent.sigev_signo = SIGRTMIN;

aio_write(&a);
/* a.aio_lio_opcode sera ignoré puisqu'on connait le type d'opération */

…
```


E/S asynchrones

Retour de fonction AIO

Toute fonction AIO renvoie 0 si l'appel est accepté par le système, -1 sinon
⇒ Mais on ne sait rien du résultat de l'E/S

Conséquence : fonctions de consultation du résultat

`int aio_error(const struct aiocb *);`
Renvoie 0 si l'opération s'est terminée avec succès
`EINPROGRESS` si l'opération est en cours
un code d'erreur sinon

`ssize_t aio_return(struct aiocb *);`
Renvoie cf. `read` & `write`
Important : une fois appelée, `aio_return` libère les ressources relatives à l'AIO
=> il faut toujours vérifier que l'opération est bien terminée avec `aio_error`

E/S asynchrones

Lancer une combinaison d'AIOs

On peut lancer une suite d'AIOs en un seul appel système
⇒ Il faut définir tous les `aiocb` d'opérations à lancer

```
int lio_listio(int mode, struct aiocb *const list[], int nent,
               struct sigevent *sig);
```

- *mode* synchronie de l'appel
  - `LIO_WAIT` attendre la fin de **toutes** les opérations
  - `LIO_NOWAIT` no comment...
- *list* liste des opérations à lancer
- *nent* nombre d'opérations à lancer
- *sig* événement à déclancher à la fin de **toutes** les opérations

Renvoie 0 si toutes les opérations se sont terminées avec succès (`LIO_WAIT`) ou si le système accepte l'appel (`LIO_NOWAIT`)
`-1` sinon

E/S asynchrones

Attendre la terminaison d'une AIO

Même si elle est asynchrone, on peut avoir besoin d'attendre la fin d'une E/S

```
int aio_suspend(const struct aiocb * const list[], int nent,
                const struct timespec *timeout);
```

- *list* liste des opérations dont la fin est attendue
  L'attente est rompue à la **première** fin d'opération
- *nent* nombre d'opérations dans la liste
- *timeout* temps maximal d'attente

Renvoie 0 si une des opérations s'est terminée
`-1` sinon
`errno == EINTR` si un signal a interrompu l'attente ou si le *timeout* est atteint

E/S asynchrones – Exemple 1

```
#define SIGNAL_IO (SIGRTMIN + 3)

void
gestionnaire (int signum, siginfo_t * info, void * vide)
{
   struct aiocb * cb;
   ssize_t nb_octets;
   if (info -> si_code == SI_ASYNCIO)
      cb = info -> si_value . sival_ptr;
      if (aio_error (cb) EINPROGRESS)
      return;
      nb_octets = aio_return (cb);
      fprintf (stdout, "Lecture 1 : %d octets lus \n", nb_octet
   }
}

void
thread (sigval_t valeur)
{
   struct aiocb * cb;
   ssize_t nb_octets;
   cb = valeur . sival_ptr;
   if (aio_error (cb) == EINPROGRESS)
      return;
   nb_octets = aio_return (cb);
   fprintf (stdout, "Lecture 2 : %d octets lus \n", nb_octets);
```

```
int
main (int argc, char * argv [])
{
   int fd;
   struct aiocb cb [3];
   char buffer [256] [3];
   struct sigaction action;
   int nb_octets;

   if (argc != 2) {
      fprintf (stderr, "Syntaxe : %s fichier \n", argv [0]);
      exit (1);
   }

if ((fd = open (argv [1], O_RDONLY)) < 0) {
   perror ("open");
   exit (1);
}
action . sa_sigaction = gestionnaire;
action . sa_flags = SA_SIGINFO;
sigemptyset (& action . sa_mask);
if (sigaction (SIGNAL_IO, & action, NULL) < 0) {
   perror ("sigaction");
  exit (1);
}
```

C. Blaess

E/S asynchrones – Exemple 1

aio_read

```
/* Lecture 0 : Pas de notification */
cb [0] . aio_fildes = fd;
cb [0] . aio_offset = 0;
cb [0] . aio_buf = buffer [0];
cb [0] . aio_nbytes = 256;
cb [0] . aio_reqprio = 0;
cb [0] . aio_sigevent . sigev_notify = SIGEV_NONE;
/* Lecture 1 : Notification par signal */
cb [1] . aio_fildes = fd;
cb [1] . aio_offset = 0;
cb [1] . aio_buf = buffer [1];
cb [1] . aio_nbytes = 256;
cb [1] . aio_reqprio 0;
cb [1] . aio_sigevent sigev_notify = SIGEV_SIGNAL;
cb [1] . aio_sigevent sigev_signo = SIGNAL_IO;
cb [1] . aio_sigevent sigev_value . sival_ptr = & cb [1];
/* Lecture 2 : Notification par thread   */
cb [2] . aio_fildes = fd;
cb [2] . aio_offset = 0
cb [2] . aio_buf buffer [2];
cb [2] . aio_nbytes = 256;
cb [2] . aio_reqprio = 0;
cb [2] . aio_sigevent sigev_notify = SIGEV_THREAD;
cb [2] . aio_sigevent sigev_notify_function = thread;
cb [2] . aio_sigevent sigev_notify_attributes = NULL;
cb [2] . aio_sigevent sigev_value . sival_ptr = & cb [2];
```

```
/* Lancement deš lectureš */
if ((aio_read (& cb [0]) < 0)
   ||(aio_read (& cb [1]) < 0)
   ||(aio_read (& cb [2]) < 0)) {
   perror ("aio_read");
   exit (1);
}
fprintf (stdout, "Lectures lancées \n");
while ((aio_error (& cb [0]) == EINPROGRESS)
   || (aio_error (& cb [1]) == EINPROGRESS)
   || (aio_error (& cb [2]) == EINPROGRESS))
      sleep (1);
   nb_octets = aio_return (& cb [0]);
   fprintf (stdout, "Lecture 0 : %d octets lus \n", nb_octets);
   return (0);
}
```

lio_read

Dans chaque structure `aiocb`

- `LIO_READ` : on veut faire une lecture.

```
   struct sigevent lio_sigev;
   struct aiocb * lio [3];

   /* Lancement des lectures */
      lio [0] = & cb [0];
      lio [1] = & cb [1];
      lio [2] = & cb [2];
      liosigev . sigev_notify = SIGEV_NONE;
      if (lio_listio (LIO_NOWAIT, lio, 3, & lio_sigev) < 0) {
         perror ("lio_listio");
         exit (1);
      }
```

E/S asynchrones – Exemple 2

```
#define NB_OP10

int
main (int argc, char * argv [])
{
   int fd;
   int i;
   struct aiocb cb [NB_OP];
   char buffer [256] [NB_OP];

   struct sigevent lio_sigev;
   struct aiocb * lio [NB_OP];

   if (argc != 2) {
      fprintf (stderr, "Syntaxe %s fichier \n", argv [0]);
      exit (1);
   }
   if ((fd = open (argv [1], O_RDONLY)) < 0) {
      perror ("open");
      exit (1);
   }
   for (i = 0; i < NB_OP; i ++) {
      cb [i] . aio_fildes = fd;
      cb [i] . aio_offset = 0;
      cb [i] . aio_buf = buffer fi];
      cb [i] . aio_nbytes = 256;
      cb [i] . aio_reqprio = NB_OP - i;
      cb [i] . aio_lio_opcode = LIO_READ;
      cb [i] . aio_sigevent . sigev_notify = SIGEV_NONE;
       lio [i] = & cb[i];
   }
   lio_sigev . sigev_notify = SIGEV_NONE;
   if (lio_listio (LIO_NOWAIT, lio, NB_OP, & lio_sigev) < 0) {
      perror ("lio_listio");
      exit (1);
   }
   fprintf (stdout, "Lectures lancées \n");
   while (1) {
      /* Reste-t-il des opérations en cours */
      for (i = 0; i < NB_OP; i ++)
         if (lio [i] != NULL)
            break;
      if (i == NB_OP)
         /* Toutes les opérations sont finies */
         break;
      if (aio_suspend (lio, NB_OP, NULL) _= 0) {
         for (i = 0; i < NB_OP; i ++)
            if (lio [i] != NULL)
               if (aio_error (lio [i]) != EINPROGRESS) {
                  fprintf (stdout, "Lecture %d : %d octets \n",
                           i, aio_return (lio [i]));
                  /* fini... */
                  lio [i] = NULL;
               }
      }
   }
   return (0);
}
```

C. Blaess

Gestion du temps

Mesure du temps

La période minimale dépend de la puissance (cadence) du processeur

- Décomposition du temps en ticks horloge
- Constante HZ dans <sys/param.h>

Fréquence de tick in HZ Hertz et une période de 1/HZ seconds

Exemple :

#define HZ 1000

- interruption horloge avec une fréquence de 1000HZ (1000 fois par second).

Gestion du temps

Horloges

Font partie intégrante d'UNIX depuis le début

3 horloges, dont une principale :

`ITIMER_REAL:` horloge globale

ITIMER_VIRTUAL : temps virtuel du processus seul (temps CPU).

ITIMER_PROF: temps virtuel complet du processus (temps CPU + des appels systèmes).

2 fonctions principales : `time` & `gettimeofday`

Le monde selon UNIX est né le 1er janvier 1970 à 00:00am (*Epoch*)

Temporisateurs

2 types : ponctuel, périodique

UNIX possède un timer ponctuel de base : la fonction `sleep`

Gestion du temps

Fonctionnalités POSIX manquantes dans UNIX

Définir un nombre d'horloges supérieur à 3

Établir une mesure de temps inférieure à la microseconde

La majorité des processeurs actuels peut compter en nanosecondes

Déterminer le nombre de débordements d'un temporisateur

ie. le temps écoulé depuis la dernière échéance **traitée**

Choisir le signal indiquant l'expiration du temporisateur

UNIX par défaut : `SIGALRM`

Gestion du temps

Horloges POSIX

Autant d'horloges que définies dans <time.h>

Nombre d'horloges et leur précision dépend de l'implémentation
Un identifiant par horloge (type `clockid t`)
Une horloge POSIX **doit** être fournie par l'implém : `CLOCK_REALTIME`

Structure de comptabilisation du temps à granularité en nanosecondes

```
struct timespec {
      time_t tv_sec;  /* secondes dans l'intervalle */
      time_t tv_nsec; /* NANOsecondes dans l'intervalle */
};
```

Fonctions d'utilisation

```
include <time.h>
int clock_settime(clockid_t, const struct timespec *);
int clock_gettime(clockid_t, struct timespec *);
int clock_getres(clockid_t, struct timespec *);
```


Gestion du temps

Horloges POSIX - fonctions d'utilisation

Renvoient -1 en cas d'échec, 0 sinon

Récupération de la précision (*eng : resolution*)

```
int clock_getres(clockid_t cid, struct timespec *res);
```

- *cid* identifiant de l'horloge dont on cherche la précision
- *res* résultat : précision de l'horloge

Récupération de l'heure courante

```
int clock_gettime(clockid_t cid, struct timespec *cur_time);
```

- *cid* identifiant de l'horloge dont on veut obtenir l'heure
- *cur_time* résultat : heure courante

Changement de l'heure courante

```
int clock_settime(clockid_t cid, struct timespec *new_time);
```

- *cid* identifiant de l'horloge dont on veut changer l'heure
- *new_time* nouvelle heure courante après mise à jour

Gestion de Temps

- L'argument *cid* est l'identifiant d'une horloge particulière sur laquelle agir.

  - Une horloge peut être globale au système, et par conséquent visible de tous les processus, ou propre à un processus, si elle mesure le temps uniquement pour celui-ci.

- Toutes les implémentations supportent l'horloge temps réel globale, laquelle est identifiée par **CLOCK_REALTIME**.
  - Son temps représente le nombre de secondes et nanosecondes écoulées depuis le début de l'Ére Unix (01-01-1970 GMT 00:00).
  - Lorsque son temps est modifié, les horloges mesurant un intervalle de temps ne sont pas affectées alors que celles indiquant une date (heure) absolue le sont.

Exemple

```
#include <time.h>

…

struct timespec what_time_is_it;

if (clock_gettime(CLOCK_REALTIME, &what_time_is_it) == -1) {
    perror("clock_gettime");
    exit(1);
}
printf("temps écoulé depuis Epoch : %d nanosecondes",
         what_time_is_it.tv_sec*1E9 + what_time_is_it.tv_nsec);

…
```

Gestion du temps

Temporisateurs POSIX

POSIX autorise N temporisateurs par processus
minimum 32, maximum `TIMER_MAX` (`<limits.h>`)
Chaque temporisateur est un élément distinct dans le système
- identifiant unique
- événement spécifique déclenché à l'échéance
- ressources associées (à libérer après utilisation, donc...)

Structure de description de temporisateur

```
struct itimerspec {
     struct timespec it_value;    /* première échéance */
     struct timespec it_interval; /* échéances suivantes */
};
```

`it_interval` équivalent à 0 nanosecondes => temporisateur ponctuel

Gestion du temps

Temporisateurs POSIX - Nanosleep

```
int clock_nanosleep(clockid_t cid, int flags,
        const struct timespec *rqtp, struct timespec *rmtp);
```

- *cid* identifiant de l'horloge régissant le temps
- *flags* mode de temporisation
  - `TIMER_ABSTIME` Temps absolu (ie. date précise, construite avec `mktime`)
  - `0` Temps relatif (ie. à partir de l'appel)
- *rqtp* échéance du réveil
- *rmtp* temps restant jusqu'à l'échéance si un signal a interrompu le sommeil

Renvoie 0 si le temps requis est écoulé, un code d'erreur sinon

Gestion du temps

Temporisateurs POSIX - Création & destruction

Création de temporisateur

```
int timer_create(clockid_t cid, struct sigevent *evp, timer_t *tid);
```

- *cid* identifiant de l'horloge régissant le temporisateur
- *evp* événement déclenché lorsque le temporisateur arrive à échéance
- *tid* identifiant du temporisateur créé

Destruction de temporisateur (implicite à la terminaison du processus propriétaire)

```
int timer_delete(timer_t tid);
```

- *tid* identifiant du temporisateur à détruire

Renvoient -1 en cas d'échec, 0 sinon

Gestion du temps – timer_create ( )

Cree une nouvelle minuterie pour une processus. Le parametre clockid indique l'horloge que la nouvelle minuterie utilise pour mesurer le temps. Il peut prendre une des valeurs suivantes :

- **CLOCK_REALTIME:** Une horloge temps reel .
- **CLOCK_MONOTONIC:** Une horloge non configurable, toujours croissante qui mesure le temps depuis un instant non spécifie dans le passe et qui ne change pas après le démarrage du système.
- **CLOCK_PROCESS_CPUTIME_ID** :Une horloge qui mesure le temps CPU (utilisateur et système) consomme par le processus appelant (et tous ses threads).
- **CLOCK_THREAD_CPUTIME_ID**: Une horloge qui mesure le temps CPU (utilisateur et système) consommé par le processus appelant.

Gestion du temps

Temporisateurs POSIX - Manipulation

Renvoient -1 en cas d'échec, 0 sinon

Armement de temporisateur

```
int timer_settime(timer_t timerid, int flags,
        const struct itimerspec *value, struct itimerspec *ovalue);
```

- *timerid* identifiant du temporisateur à armer
- *flags* mode de temporisation
  - `TIMER_ABSTIME` Temps absolu (ie. date précise, construite avec `mktime`)
  - `0` Temps relatif (ie. à partir de l'armement)
- *value* échéances (première et suivantes) du temporisateur
- *ovalue* temps restant jusqu'à la prochaine échéance **courante** (ie. avant réarmement)

Renvoie -1 en cas d'échec, 0 sinon

Gestion du temps

Temporisateurs POSIX - Manipulation (suite)

Consultation de temporisateur

```
int timer_gettime(timer_t timerid, struct itimerspec *t_remaining);
```

- *timerid* identifiant du temporisateur consulté
- *t_remaining* temps restant jusqu'à la prochaine échéance

Renvoie -1 en cas d'échec, 0 sinon

Détermination du débordement

```
int timer_getoverrun(timer_t timerid);
```

- *timerid* identifiant du temporisateur consulté

Renvoie le nombre de déclenchements non traités, -1 sinon

A chaque traitement d'événement déclenché par une échéance, ce nombre est remis à 0

Permet de pallier l'impossibilité de comptabiliser le nombre de signaux reçus

Exemple

```
#include <time.h>
#include <signal.h>

…

timer_t tmr_expl;
struct sigevent signal_spec;
sigexpl.sigev_notify = SIGEV_SIGNAL;
signal_spec.sigev_signo = SIGRTMIN; /* signal utilisateur POSIX.4 */
timer_create(CLOCK_REALTIME, &signal_spec, &tmr_expl);

struct itimerspec new_tmr, old_tmr;
new_tmr.it_value.tv_sec = 1;
new_tmr.it_value.tv_nsec = 0;
new_tmr.it_interval.tv_sec = 0;      /* temporisateur ponctuel */
new_tmr.it_interval.tv_nsec = 0;     /* temporisateur ponctuel */
timer_settime(tmr_expl, 0, &new_tmr, &old_tmr);

pause();

…
```


Exemple – timer/signal

```
static void handler(int sig, siginfo_t *si, void *uc)
{  printf("Caught signal %d\n", sig);
   printf(" sival_ptr = %p; ", si->si_value.sival_ptr);
}

 main(int argc, char *argv[]) {
 timer_t timerid; struct sigevent sev; struct itimerspec its;
 long long freq_nanosecs; sigset_t mask;
  struct sigaction sa; sa.sa_flags = SA_SIGINFO;
 sa.sa_sigaction = handler;
 sigemptyset(&sa.sa_mask);
 if (sigaction(SIGRTMIN, &sa, NULL) == -1)
    perror("sigaction");

 sigemptyset(&mask); sigaddset(&mask, SIGRTMIN);
 if (sigprocmask(SIG_SETMASK, &mask, NULL) == -1)
 perror("sigprocmask");
```

```
 sev.sigev_notify = SIGEV_SIGNAL;
 sev.sigev_signo = SIGRTMIN;
 sev.sigev_value.sival_ptr = &timerid;
 if (timer_create(CLOCK_REALTIME,
 &sev, &timerid) == -1)
    perror ("timer_create");

 freq_nanosecs = atoll(argv[2]);
 its.it_value.tv_sec = freq_nanosecs /
 1000000000;
 its.it_value.tv_nsec = freq_nanosecs %
 1000000000;
 its.it_interval.tv_sec =its.it_value.tv_sec;
 its.it_interval.tv_nsec =its.it_value.tv_nsec;
 if (timer_settime(timerid, 0, &its, NULL)
 == -1)
      perror("timer_settime");
```

Exemple – timer/signal

```
/* Sleep for a while; meanwhile, the timer may expire multiple times */
  printf("Sleeping for %d seconds\n", atoi(argv[1]));
        sleep(atoi(argv[1]));

  /* Unlock the timer signal, so that timer notification can be delivered */
   printf("Unblocking signal %d\n", SIGRTMIN);
   if (sigprocmask(SIG_UNBLOCK, &mask, NULL) == -1)
              perror("sigprocmask");

  exit(EXIT_SUCCESS);
}
```


Conclusion

POSIX.1b = Outils de construction de systèmes temps réel

- Processus légers
- Sémaphores
- Partage et verrouillage mémoire
- Ordonnancement
- Gestion du temps
- E/S asynchrones
- Signaux TR

! Ce cours omet volontairement des éléments POSIX.1b !

- Verrouillage mémoire : `mlock`, `mlockall`, `munlock`