

Programmation système

TP à rendre 3

Éléments de correction

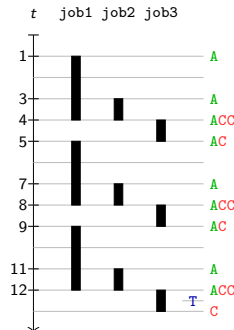
Rappels :

- ▶ Lancer des jobs périodiquement
par exemple :

sujet.table		
1:	job1:	sleep 3
2:	job2:	sleep 1
1:	job3:	sleep 1



- ▶ Entièrement piloté par des signaux :
SIGALRM : lancer un nouveau job ;
SIGCHLD : détecter la fin d'un job ;
SIGTERM : arrêter dès que possible ;
SIGUSR1 : activer/désactiver l'affichage ;
SIGUSR2 : afficher les jobs en cours.



1. Structure générale du programme

- ▶ Le programme doit avoir la structure générale suivante :

1. programmer l'alarme du premier job
2. tant qu'on n'a pas reçu SIGTERM
 - 2.1 attendre un signal (ou plusieurs)
 - 2.2 traiter les signaux reçus
3. attendre la fin des éventuels jobs encore en cours

- ▶ Pourquoi attendre ?

- ▶ Ce programme est *très peu* actif : en moyenne

$$n_{\text{job}} / \left(\sum_i \text{jobs}[i].\text{delay} \right)$$

lancement de job + attente *par seconde*

- ▶ Tous les événements sont « externes » au processus :
 - ▶ horloge (le système) pour SIGALRM
 - ▶ un fils (via le système) pour SIGCHLD
 - ▶ utilisateur (via le système) pour SIGUSR1/USR2/TERM
 - ▶ L'*ordre* des événements n'est pas prévisible : il dépend de la durée des jobs, de l'utilisateur... et de décisions du système
 - ▶ C'est un exemple de programme *réactif*
Autres exemples : GUI, serveurs, syst. distribués/temps-réel...

2. Code : communication gestionnaire → programme

- ▶ Un gestionnaire de signal s'exécute de façon *asynchrone* (n'importe quand, ou presque, *pendant* le programme)
- ▶ Quand le signal est « livré » : appel d'un gestionnaire → prototype contraint + pas de code « compliqué »
- ▶ Pas un appel normal : on ne peut pas passer des paramètres → variables globales

- ▶ Pour chaque signal à traiter :

```
volatile sig_atomic_t recu_XXX = 0;
void gestion_XXX (int sig)
{
    recu_XXX = 1;
}
```

et remise à 0 après traitement

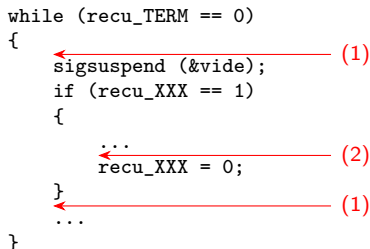
- ▶ Variante : un seul gestionnaire pour tous les signaux
 - ▶ `switch (sig) ... / if (sig == XXX) ... else ...`
 - ▶ tableau de `sig_atomic_t`

Il peut être interrompu par lui-même (pour un autre signal)

3. Attente et concurrence

- ▶ Une seule façon d'attendre un signal : `sigsuspend()`
= attente bloquante de l'exécution d'un gestionnaire
- ▶ Utilisation naïve (sans masque) toujours fausse :

```
while (recu_TERM == 0)
{
    sigsuspend(&vide);
    if (recu_XXX == 1)
    {
        ...
        recu_XXX = 0;
    }
    ...
}
```



- ▶ Problèmes :
 - (1) le signal arrive avant `sigsuspend()`, après le test
→ attente éternelle (par exemple : `SIGTERM/ALRM`)
 - (2) seconde arrivée pendant le traitement
→ une occurrence perdue (par exemple : `SIGUSR1/USR2`)
- ▶ Idem si `sigsuspend()` est dans un test
- ▶ Ça ne *peut pas* marcher ! (cf. cours)

4. Blocage des signaux

- ▶ Un signal peut-être bloqué (ou masqué) :
 - ▶ = mis en attente
 - ▶ le gestionnaire ne s'exécute pas. . .
 - ▶ jusqu'à ce que le signal soit débloqué
- ▶ Il y a 3 façons de changer le masque de blocage
 - ▶ par `sigprocmask()`
 - ▶ pendant `sigsuspend()`
 - ▶ pendant l'exécution du gestionnaire (`sa_mask`)
- ▶ Au changement de masque : exécution du gestionnaire
 - ▶ pour tous les signaux en attente ainsi débloqués
 - ▶ dans un ordre non défini
- ▶ Un appel à `sigsuspend(&msk)` :
 1. change le masque courant en `msk`
 2. attend l'arrivée d'un signal non bloqué dans `msk`
 3. rétablit le masque initial

Chaque étape peut provoquer l'exécution de gestionnaire(s), `sigsuspend()` retourne immédiatement si c'est le cas dans 1.

5. Code : masques pour chronos

- ▶ Principe : bloquer les signaux partout sauf en sigsuspend()

```
CHECK (sigemptyset (&masque));  
CHECK (sigaddset (&masque, SIGUSR1));  
... // + idem pour SIGUSR2, SIGALRM, SIGCHLD, SIGTERM  
CHECK (sigprocmask (SIG_BLOCK, &masque, &initial));  
...  
while (recu_TERM == 0)  
{  
    sigsuspend (&masque);  
    if (...) { ... }  
    ...  
}  
...  
CHECK (sigprocmask (SIG_SETMASK, &initial, NULL));
```

- gestionnaires exécutés « dans » sigsuspend() uniquement
- reste du code « protégé » des signaux (ex : sigaction())
 - ▶ Risque : arrivées multiples du même signal bloqué
 - ▶ tant pis, pas très grave (SIGUSR1/USR2)
 - ▶ logiquement impossible (SIGALRM/TERM)
 - ▶ géré autrement (SIGCHLD)

Et surtout : de toute façon inévitable !

6. Remarques sur le blocage

- ▶ Que faire des autres signaux (SIGHUP etc.) ?
Rien, on ne s'en occupe pas !
 - ▶ signaux = mécanisme coopératif, pas hostile
(+ soumis aux permissions habituelles)
 - ▶ SIGSTOP et SIGKILL intouchables
 - ▶ ne *pas* utiliser `sigfillset()`+`sigdelset()` !
 - ▶ Rappel : (`man sigprocmask`)
 - ▶ `fork()` : masque préservé dans le nouveau processus
 - ▶ `exec()` : masque préservé dans le nouveau programme
- il faut rétablir le masque original en lançant un job
- ▶ Des signaux peuvent être déjà masqués/ignorés à l'appel :
 - ▶ commande `nohup/disown` : ignore SIGHUP (perte de terminal)
 - ▶ on pourrait exécuter `chronos` en masquant SIGUSR1/USR2
(par exemple si il n'y a pas de terminal)
 - ▶ ...

7. Code : démarrage des jobs

- ▶ Alarmes :
 - ▶ au début de l'exécution pour le premier job
 - ▶ après lancement d'un job, pour le job suivant
- ▶ Avant et dans la boucle de traitement :

```
nextjob = 0;
alarm (table->jobs[nextjob].delay);
while (...)
{
    ...
    if (recu_ALRM == 1)
    {
        if (table->jobs[nextjob].pid != -1)
            chronos_log ("OMIT %s\n", table->jobs[nextjob].id);
        else
            chronos_exec (&table->jobs[nextjob], &initial);
        nextjob = (nextjob + 1) % table->njob;
        alarm (table->jobs[nextjob].delay);
        recu_ALRM = 0;
    }
    ...
}
```

avec `chronos_exec()` : idem TP2, + `sigprocmask()`

8. Code : réagir à SIGCHLD

- ▶ Rappel : signal SIGCHLD et wait()/waitpid() indépendants
- ▶ Rappel : nombre exact de SIGCHLD reçus inconnu
→ boucle d'appels, non-bloquant, nombre quelconque

```
void chronos_reap (struct table * table)
{
    pid_t pid; int status; int n = 0;
    while((pid=waitpid(-1, &status, WNOHANG)) > 0) {
        int index = table_find_pid (table, pid);
        int ok = WIFEXITED(status) &&
                (WEXITSTATUS(status)==EXIT_SUCCESS);
        chronos_log ("FINI %s %d %s\n", table->jobs[index].id,
                    pid, (ok ? "ok" : "ko"));
        table->jobs[index].pid = -1;
        ++ n;
    }
    if (n != 1) chronos_log ("REAPED %d\n", n); // par curiosité
}
```

- ▶ Cas tordu : un fils se termine *pendant* ce traitement
→ waitpid() renvoie son pid, deuxième itération
→ le signal (bloqué) sera traité plus tard
→ messages REAPED 2, puis REAPED 0 ; c'est la vie

9. Code : après SIGTERM

- ▶ Après la fin de la boucle, avant `sigprocmask()`

```
for (int i=0 ; i<table->njob ; i++)
{
    if (table->jobs [i].pid != -1)
    {
        int status;
        if (waitpid (pid, &status, 0) == -1)
            raler ("waitpid");
        chronos_log ("WAIT %s %d %s\n", ...);
        table->jobs [i].pid = -1;
    }
}
```

(rappel : les signaux sont toujours bloqués)

- ▶ Ici, `waitpid()` est bloquant, similaire à `wait()`, mais dans l'ordre des jobs (il faudra attendre qu'ils se terminent tous)
- ▶ Une attente non-bloquante (`WNOHANG`) serait soit une attente active, soit une erreur (selon la boucle utilisée)

10. Autres remarques

- ▶ La fonction `chronos_log()` :

```
#ifdef __GNUC__  
__attribute__((format (printf, 1, 2))) /* attribut de chronos_log */  
#endif  
void chronos_log (const char * fmt, ...) { /* etc. */ }
```

- ▶ si le compilateur est bien gcc (`#ifdef...#endif`)...
- ▶ alors considérer que les arguments sont les mêmes que ceux de `printf` (via un *attribut de fonction* destiné au compilateur)
- ▶ c'est-à-dire que le compilateur vérifie les types en fonction des marqueurs (`%s` → `char*`, `%d` → `int`, etc.)

C'est une extension non-standard de gcc

- ▶ `alarm()` est un peu ancien, précis à la seconde seulement
→ `setitimer()` (une seule alarme, périodique, à la μs)
→ `timer_create()` / `timer_settime()` / `timer_delete()`
- ▶ Inspiration pour ce projet : `cron` (standard Unix)
 - ▶ résolution entre minute et année (au lieu de seconde)
 - ▶ paramétrage par instant (au lieu de délai) : `man 5 crontab`