Master1 – POSIX **2004-2005** **Contrôle Continu**	- Durée : 2 heures - Toute documentation autorisée - Barème indicatif *Luciana Arantes, Bertil Folliot et Olivier Marin*

GESTION DES PROCESSUS

1. CRÉATION DE PROCESSUS (2 POINTS)

1.1

Écrivez le programme execute.c dont voici deux activations possibles :

```
execute "nomprog", execute "nomprog &"
```

Ce père génère un processus fils qui exécute le programme "nomprog" par substitution de code. Si le caractère "&" est absent, le père attend la fin du fils. S'il est présent en fin du second argument, le père n'attend pas la fin du fils. Dans les deux cas, le père affiche le PID du fils.

```
#define …
#include …

int main (int argc, char *argv[]) {
      int nowait=0, pid, len;
       if (argc != 2) {fprintf(stderr,"USAGE?\n"); exit(1);}
      len = strlen(argv[1])-1;  // -1 pour le \n
      if (argv[1][len] == '&') {
            nowait =1;
            argv[1][len-1] = '\0'; // -1 pour le ' '
      }
       if ((pid = fork()) == 0)
            if (execlp(argv[1], argv[1], 0) < 0) {perror("exec"); exit(1);}
       // PERE
       if (!nowait) wait(0);
       printf("[%d]\n", pid);
       exit(0);
}
```

2. SYNCHRONISATION DE PROCESSUS (8 POINTS)

2.1

On souhaite synchroniser N processus (N connu au démarrage du programme). Les N processus itèrent des calculs. Une itération correspond à une suite de deux phases. Une première phase en parallèle, puis une deuxième phase en séquentiel. C'est le processus initial qui contrôle les phases de calcul, ainsi que la terminaison de l'application. On considère que tous les signaux envoyés sont reçus.

Dans la suite de cet exercice vous expliquerez la structure de vos fonctions, des variables et des fonctions systèmes utilisées avant de les programmer.

Écrivez la fonction de création des N processus. Le père des N processus doit mémoriser les PID de tous ses fils. Chaque fils créé peut immédiatement commencer la phase de calcul en parallèle en appelant la fonction calcul_para(). Le temps de traitement de calcul_para() est variable suivant les fils.

Chaque fils prévient son père lorsqu'il a terminé, puis se met en attente de la réception d'un signal de son père (on utilisera le signal SIGUSR1). Écrivez la fonction d'attente des fils.

Le père attend d'avoir reçu le signal SIGUSR1 des N fils avant de commencer la phase de calcul séquentiel. Il prévient alors le premier fils. Celui-ci exécute le traitement calcul_synchro(), puis prévient le père lorsqu'il a terminé et se remet en attente d'un signal du père.

Le père prévient alors le deuxième fils, et ainsi de suite, jusqu'à ce que les N processus aient effectué le calcul synchronisé. Ecrivez les fonctions de synchronisation du père et des fils.

Lorsque les N fils ont terminé le calcul synchronisé, le père débloque les N fils, qui commencent alors la deuxième itération du calcul parallèle.

L'application se termine lorsque le père reçoit un signal SIGINT. Il prévient alors tous les fils (avec SIGINT) et affiche le nombre d'itérations effectuées. Complétez les fonctions précédentes pour le père et les fils.

```
#define _POSIX_SOURCE 1

#include …

int n;
int *tab_N;
int cpt, nb_iter, attente = 1;
int pid;

//PERE & FILS

void fct_null() { // SIGUSR1 (inutile pour le fils)
  cpt++;
  if (cpt==n) { // fin de l'attente des N SIGUSR1
    attente = 0;
    cpt=0;
  }
}

//FILS

void calcul_synchro() {
```

```
  // TRAVAIL_SYNCHRO()
  if (kill(getppid(), SIGUSR1) < 0) {perror("kill"); exit(1);}
  pause();
  calcul_para();
}

void calcul_para(void) {
  // TRAVAIL_PARA()
  if (kill(getppid(), SIGUSR1) < 0) {perror("kill"); exit(1);}
  pause();
  calcul_synchro();
}

//PERE

void fct_stop() { // SIGINT pour tuer proprement les fils
  int i;
  for (i=0; i< n; i++) {
    if (kill(tab_N[i], SIGINT) < 0) {perror("kill fils"); exit(1);}
  }
  printf("Nb. iter %d\n", nb_iter);
  exit(0);
}

void creer_N (void) {
  if ((tab_N = (int *) malloc(n * sizeof(int))) == 0) {perror("malloc"); exit(1);}
  for (pid = 0; pid < n; pid++) {
    if ((tab_N[pid] = fork()) < 0) {perror("fork"); exit(1);}
    if (tab_N[pid] == 0) calcul_para();
  }
}

void lancer_N(void) {
  int i;
  for (i=0; i< n; i++)
    if (kill(tab_N[i], SIGUSR1) < 0) {perror("kill"); exit(1);}
  attente_N();
}

void attente_N (void) {
  int i;
  while (attente) pause();
  // calcul synchro
```

```
  for (i=0; i< n; i++) {
    if (kill(tab_N[i], SIGUSR1) < 0) {perror("kill"); exit(1);}
    pause();
  }
  nb_iter++;
  cpt=0;
  // calcul para
  lancer_N();
}

// MAIN
int main(int argc, char *argv[]) {
    struct sigaction act;

    if (argc != 2) exit(1);
    if ((n = atoi(argv[1])) <= 0) exit(1);

    act.sa_handler = fct_null;
    act.sa_flags = 0;
    sigaction(SIGUSR1, &act, 0); // pour les synchros

    creer_N();

    act.sa_handler = fct_stop;
    sigaction(SIGINT, &act, 0); // pour terminer l'appli.

    attente_N();

    exit(0); // jamais atteint
}
```

3. EXECUTION DES PROCESSUS (3 POINTS)

Considérez le code du programme ci-dessous :

```
1: int main(int argc, char **argv) {
2:  pid_t pid;
3:  int  j=0; int i = 0;

4:  while (i < 2) {
5:    i ++;

6:    if ( (pid = fork ()) == -1) {
7:       perror ("fork");
8:       exit (1);
9:    }
10:     else if (pid == 0) j=i;
11:  }
```

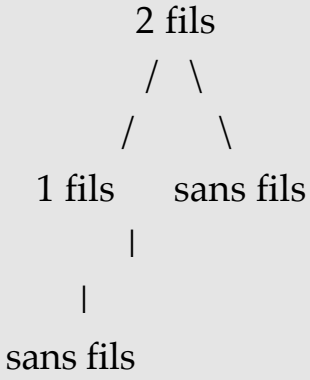
```
12:  if (j == 2) {
13:          sleep (2);
14:          printf ("sans fils \n");
15:  }
16:  else {

17:          printf ("%d fils \n », (i-j) );
18:          while (j < i) {
19:                  j++;
20:                  wait (NULL);
21:          }
22:  }
23:  return EXIT_SUCCESS;
24: }
```

3.1

Combien de processus sont-ils créés ? Dessinez l'arborescence des processus en montrant les affichages de chaque processus.

4 processus :



3.2

Au lieu d'appeler la fonction *sleep* dans la ligne 13, nous voulons appeler la commande *sleep* (e.g. sleep 2), qui se trouve dans le répertoire /bin/sleep. Remplacez la ligne 13 par l'appel de la commande sleep on utilisant *execl*. Même question en utilisant *execv*.

Le nombre de processus créés change-t-il ? Et l'affichage ?

```
if (execl ("/bin/sleep", "sleep", "2",NULL) == -1)
    perror ("execl");


char *argv[NMAX];
argv[0] = "sleep"; argv[1] = "2"; argv[2] = NULL;
if (execv("/bin/sleep", argv) == -1)
    perror ("execl");
```

Le nombre de processus ne change pas, mais l'affichage oui. On a plus l'affichage « sans fils » des 2 processus sans fils. Les autres restent.

GESTION DES SIGNAUX

4. MASQUAGE DE SIGNAUX (3 POINTS)

4.1

Quel est l'effet sur le masque de signaux lorsque la fonction *sigprocmask* est appelée en passant SIG_BLOCK comme premier paramètre et l'adresse d'un ensemble vide comme deuxième paramètre ? Quel peut être l'intérêt d'un tel appel ?

Aucun. Appel intéressant pour obtenir le masque courant en troisième paramètre. (Aide à la prochaine question.)

4.2

Nous voulons offrir la fonction *unsigned int sigismasked (int sig)* qui renvoie 1 si le signal *sig* se trouve masqué et 0 en cas contraire. Programmez une telle fonction.

```
int sigismasked (int sig) {
sigset_t  sig_rec ;          /* liste des signaux */
sigset_t  sig_masque;    /* liste des signaux masques */

 sigemptyset (&sig_rec);
    /* Obtenir le masque de signaux courants */
    sigprocmask (SIG_BLOCK, &sig_rec, &sig_masque);
   if (sigismember (&sig_masque,sig))
      return 1;
   else
     return 0;
}
```

5. FONCTION SLEEP (4 POINTS)

5.1

La fonction *unsigned int sleep (int sec)* endort le processus courant pendant *sec* secondes ou jusqu'à ce qu'un signal soit capturé ; le traitement du signal ne termine pas le processus. La fonction renvoie 0 si le temps demandé est écoulé, ou le nombre de secondes restantes en cas de capture d'un signal.

Programmez la fonction *sleep* en utilisant la fonction *alarm* et le signal *SIGALRM*. Comme recommandé dans le « man » de la fonction *sleep*, nous considérons que lorsque l'usager utilise la fonction *sleep* dans son programme, il n'y utilise pas la fonction *alarm* ni le signal *SIGALRM.*

Observations :

1. Si votre fonction masque des signaux ou remplace le traitement par défaut d'un signal il faut penser à restaurer le masque et/ou le traitement originaux.

2. La description de la fonction *alarm* est donnée en annexe.

```
int flag_alrm =0;
void sig_hand(int sig){
 if ( sig == SIGALRM )
   flag_alrm = 1;
}
unsigned int sleep (unsigned int sec) {
 sigset_t  sig_bloc, anc_mask ;          /* liste des signaux bloques */
 struct sigaction action, anc_action;

 sigemptyset (&sig_bloc);

 /* masquer SIGALRM */
 sigaddset (&sig_bloc, SIGALRM);
 sigprocmask (SIG_BLOCK, &sig_bloc, &anc_mask);

 sig_bloc = anc_mask;
 sigdelset (&sig_bloc, SIGALRM);

 /* remplacer le traitement defaut de SIGALRM */
 action.sa_mask=sig_bloc;
 action.sa_flags=0;
 action.sa_handler = sig_hand;
 sigaction(SIGALRM, &action, &anc_action);

 alarm (sec);

 /* attendre signaux */
 sigsuspend (&sig_bloc);

 /* restaurer le masque ancien */
 sigprocmask (SIG_SETMASK, &anc_mask, NULL);

 /* restaurer le traitement defaut */
 sigaction(SIGALRM, &anc_action, NULL);

  if (flag_alrm == 1)
    return 0;
  else
   return (alarm (0));
}
```

ANNEXE – FONCTION ALARM [RIFFLET]

```
unsigned int alarm (unsigned int nsec)
```

Correspond à une requête au système d'envoyer au processus le signal SIGALRM dans (environ) *nsec* secondes.

Les demandes ne sont pas empilées : une demande annule la demande antérieure du processus.

La valeur renvoyée par un appel est le temps restant avant l'envoi du signal SIGALRM suite à la demande antérieure. En particulier, un appel avec la valeur 0 en paramètre annule la demande en cours sans en formuler de nouvelle.

Rappelons que le comportement par défaut pour le signal SIGALRM provoque la terminaison du processus.