

Programmation système

TP à rendre 2

Éléments de correction

Quelques règles tirées du sujet :

- ▶ Un processus par tâche, qui exécute un programme (donné) :
 - ▶ `fork()` pour créer le processus fils
 - ▶ `exec??()` pour exécuter le programme
- ▶ Les tâches s'arrêtent quand elles veulent, donc
 - ▶ `wait()` (n'importe quel fils), pas `waitpid()`
- ▶ Vérifier le bon déroulement des tâches, donc
 - ▶ `WIFEXITED()` et compagnie
- ▶ Contraintes sur l'ordonnanceur :
 - ▶ compter les tâches en cours d'exécution ($\leq nproc$)
 - ▶ connaître l'état du système (tâches prêtes, réussite/échec, etc.)

1. Rappel : API de gestion de graphes de tâches

- ▶ Charger un fichier .graph

```
struct graph * graph = graph_parse (stdin);
```

- ▶ Trouver une tâche prête :

```
struct task * ready = graph_find_ready (graph);  
if (ready == NULL) ... else ...
```

- ▶ Trouver une tâche d'après son *pid* :

```
pid_t pid = ...  
struct task * done = graph_find_byid (pid);
```

- ▶ Mettre à jour le graphe pour une tâche terminée avec succès

```
for (int s=0 ; s<done->nsucc ; s++)  
    done->succs[s]->npred -=1;
```

- ▶ Libérer la mémoire utilisée par le graphe :

```
graph_free (graph);
```


2. Rappel : structure de données pour une tâche

Attribut	Tâche data	Tâche tp12
name	"data"	"tp12"
cmd	{ "test", "-r", "notes.data", NULL }	{ "sh", "-c", "join ... tp12.data", NULL }
nsucc	3	1
succs	{ ptr vers tp1, ptr vers tp2, ptr vers tp3 }	{ ptr vers tp123 }
pid	-1	-1
npred	0	2

- ▶ l'attribut cmd a la forme d'un tableau argv ($\rightarrow v$)
- ▶ le programme n'est pas un chemin absolu ($\rightarrow p$)

But : le graphe contient l'état de l'exécution des tâches

3. Exemple d'exécution ($nproc \geq 3$)

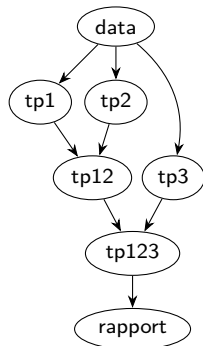
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée


 : tâche terminée

 : tâche prête



Tâche	npred
data	0
tp1	1
tp2	1
tp3	1
tp12	2
tp123	2
rapport	1

3. Exemple d'exécution ($nproc \geq 3$)

Légende :  : tâche en attente

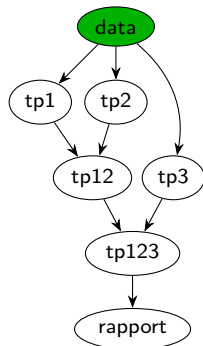
 : tâche débutée

 : tâche exécutée

 : tâche terminée


 : tâche prête

INIT data



Tâche	npred
data	0
tp1	1
tp2	1
tp3	1
tp12	2
tp123	2
rapport	1

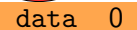
3. Exemple d'exécution ($n_{proc} \geq 3$)

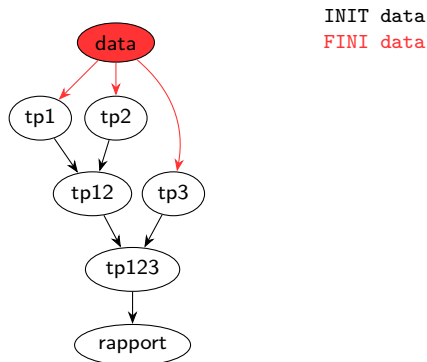
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée


 : tâche terminée

 : tâche prête



Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2
tp123	2
rapport	1

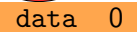
3. Exemple d'exécution ($n_{\text{proc}} \geq 3$)

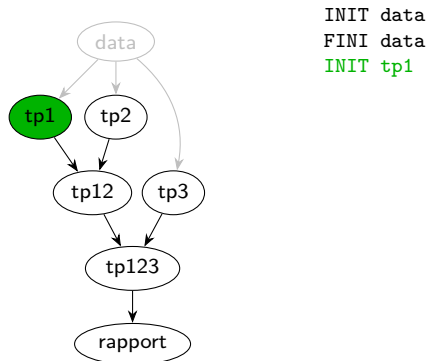
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée


 : tâche terminée

 : tâche prête



Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2
tp123	2
rapport	1

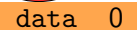
3. Exemple d'exécution ($n_{\text{proc}} \geq 3$)

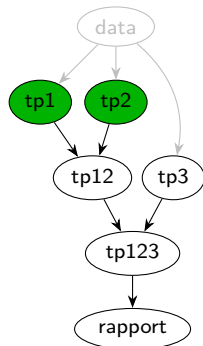
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête



INIT data


FINI data

INIT tp1

INIT tp2

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2
tp123	2
rapport	1

3. Exemple d'exécution ($n_{\text{proc}} \geq 3$)

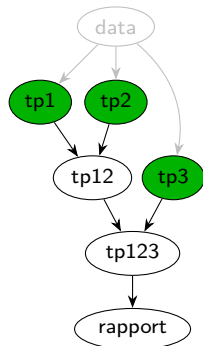
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête



INIT data

FINI data


INIT tp1

INIT tp2

INIT tp3

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2
tp123	2
rapport	1

3. Exemple d'exécution ($n_{\text{proc}} \geq 3$)

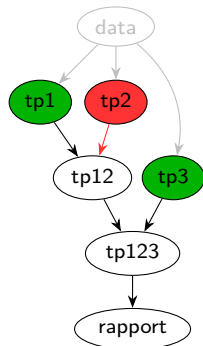
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête



INIT data

FINI data

INIT tp1


INIT tp2

INIT tp3

FINI tp2

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1
tp123	2
rapport	1

3. Exemple d'exécution ($nproc \geq 3$)

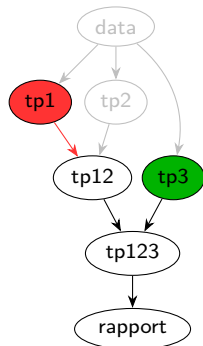
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête



INIT data

FINI data

INIT tp1

INIT tp2


INIT tp3

FINI tp2

FINI tp1

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2
rapport	1

3. Exemple d'exécution ($nproc \geq 3$)

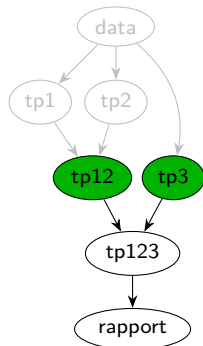
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête



INIT data

FINI data

INIT tp1

INIT tp2

INIT tp3


FINI tp2

FINI tp1

INIT tp12

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2
rapport	1

3. Exemple d'exécution ($nproc \geq 3$)

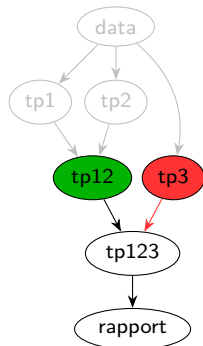
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée


 : tâche prête



INIT data
FINI data
INIT tp1
INIT tp2
INIT tp3
FINI tp2
FINI tp1
INIT tp12
FINI tp3

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2 1
rapport	1

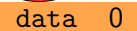
3. Exemple d'exécution ($nproc \geq 3$)

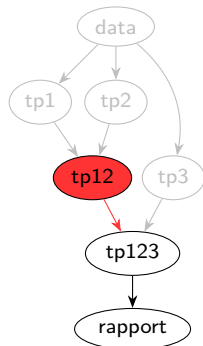
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête




```

INIT data
FINI data
INIT tp1
INIT tp2
INIT tp3
FINI tp2
FINI tp1
INIT tp12
FINI tp3
FINI tp12
  
```

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2 1 0
rapport	1

3. Exemple d'exécution ($nproc \geq 3$)

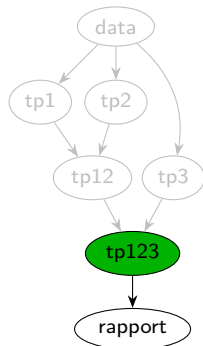
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée


 : tâche prête



INIT data
FINI data
INIT tp1
INIT tp2
INIT tp3
FINI tp2
FINI tp1
INIT tp12
FINI tp3
FINI tp12
INIT tp123

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2 1 0
rapport	1

3. Exemple d'exécution ($nproc \geq 3$)

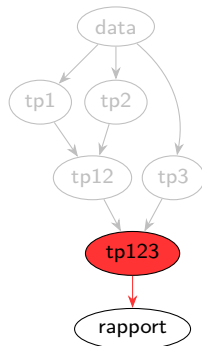
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête




```

INIT data
FINI data
INIT tp1
INIT tp2
INIT tp3
FINI tp2
FINI tp1
INIT tp12
FINI tp3
FINI tp12
INIT tp123
FINI tp123
  
```

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2 1 0
rapport	1 0

3. Exemple d'exécution ($nproc \geq 3$)

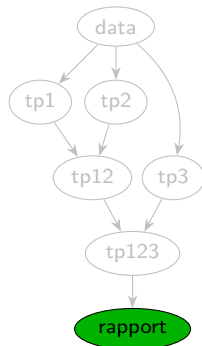
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête




```

INIT data
FINI data
INIT tp1
INIT tp2
INIT tp3
FINI tp2
FINI tp1
INIT tp12
FINI tp3
FINI tp12
INIT tp123
FINI tp123
INIT rapport
  
```

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2 1 0
rapport	1 0

3. Exemple d'exécution ($nproc \geq 3$)

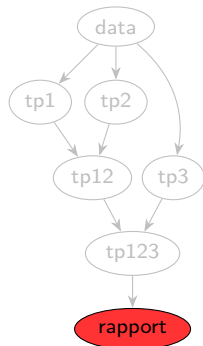
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête




```

INIT data
FINI data
INIT tp1
INIT tp2
INIT tp3
FINI tp2
FINI tp1
INIT tp12
FINI tp3
FINI tp12
INIT tp123
FINI tp123
INIT rapport
FINI rapport
  
```

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2 1 0
rapport	1 0

3. Exemple d'exécution ($nproc \geq 3$)

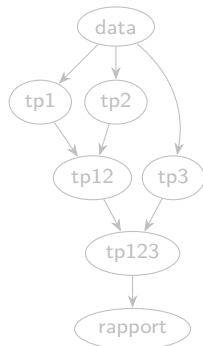
Légende :  : tâche en attente

 : tâche débutée

 : tâche exécutée

 : tâche terminée

 : tâche prête



```
INIT data
FINI data
INIT tp1
INIT tp2
INIT tp3
FINI tp2
FINI tp1
INIT tp12
FINI tp3
FINI tp12
INIT tp123
FINI tp123
INIT rapport
FINI rapport
```

Tâche	npred
data	0
tp1	1 0
tp2	1 0
tp3	1 0
tp12	2 1 0
tp123	2 1 0
rapport	1 0

Aucun échec, l'ordonnanceur s'arrête avec le code EXIT_SUCCESS

4. Code : lancer un processus fils

Rappel :

- ▶ `fork()` : nouveau processus, même programme
- ▶ `exec??()` : même processus, nouveau programme

```
int task_execute (struct task * task)
{
    pid_t pid = fork ();
    switch (pid) {
        case 0: execvp (task->cmd[0], task->cmd);
                perror ("execvp");
                exit (127); /* convention (?) */
                break;
        case -1: perror ("fork");
                ... /* Voir plus bas */
                break;
        default: task->pid = pid;
                fprintf (stderr, "INIT %s %d\n", task->name, task->pid);
                break;
    }
    return (pid != -1);
}
```

Le fils (la tâche) s'exécute à son rythme à partir de là,
le père (l'ordonnanceur) continue son job (lancer/attendre)

5. Code : attendre la fin d'un fils

```
int graph_standby (struct graph * graph, ...)
{
    int success; int result; pid_t pid;
    if ((pid=wait (&result)) == -1) raler ("wait"); /* Voir plus bas */
    struct task * task = graph_find_bypid (graph, pid); /* FIXME: */
    if (WIFEXITED (result) && WEXITSTATUS (result) == EXIT_SUCCESS) {
        for (int i=0 ; i<task->nsucc ; i++)
            -- task->succs [i]->npred;
        fprintf (stderr, "FINI %s %d\n", task->name, task->pid);
        success = 1;
    } else {
        fprintf (stderr, "DEAD %s %d ", task->name, task->pid);
        if (WIFEXITED (result))
            fprintf (stderr, "EXIT %d\n", WEXITSTATUS (result));
        else
            fprintf (stderr, "SIGNAL %d\n", WTERMSIG (result));
        success = 0;
    }
    ...
    return success;
}
```

L'ordonnanceur n'attend pas un fils particulier (\rightarrow wait())
Il manque la mesure du temps (voir plus bas).

6. Algorithmique : principe de l'ordonnancement

- ▶ Le graphe est quelconque, la durée des tâches est imprévisible
→ pas d'hypothèse possible a priori
- ▶ Objectifs de l'ordonnanceur :
 - ▶ profiter au maximum du parallélisme
→ lancer les tâches dès/autant que possible
 - ▶ respecter les contraintes du graphe
→ ne lancer que des tâches prêtes
 - + ne jamais lancer plus de `nproc` processus
→ il faut connaître le nombre de tâches « en vol »
→ une variable `running` (entre 0 et `nproc` inclus)
- ▶ État instantané de l'exécution : le graphe + `running`
Permet de répondre aux questions :
 - ▶ l'ordonnanceur peut-il lancer une nouvelle tâche ?
si oui, on la lance sans perdre de temps
 - ▶ doit-il attendre la fin d'une tâche en cours ?
oui, si il y en a une et si il ne peut pas lancer une tâche
(but : décrémenter `running`, peut-être libérer d'autres tâches)
 - ▶ l'exécution est-elle terminée ?

7. Algorithmique : lancer autant de tâches que possible

- ▶ Il y a deux conditions à considérer :
 - ▶ y a-t-il une tâche libre? ($r = \text{graph_find_ready}(g)$)
 - ▶ peut-on en lancer une nouvelle? ($\text{running} < \text{nproc}$)

Il faut distinguer 6 cas :

	$r \neq \text{NULL}$	$r == \text{NULL}$
$\text{running} == 0$	<code>execute (r)</code>	(fini)
$0 < \text{running} < \text{nproc}$	<code>execute (r)</code>	<code>standby ()</code>
$\text{running} == \text{nproc}$	<code>standby ()</code>	<code>standby ()</code>

- ▶ On peut « paver » cette table avec deux tests consécutifs :
 - si $r \neq \text{NULL}$ et $\text{running} < \text{nproc}$ alors
exécuter la nouvelle tâche r
 - sinon si $\text{running} > 0$ alors
attendre la fin d'une tâche
 - sinon
c'est fini
- ▶ On répète jusqu'à atteindre le troisième cas

8. Code : exécution d'un graphe

```
int graph_exec (struct graph * graph, unsigned nproc)
{
    int running = 0;
    int goon = 1;
    int status = EXIT_SUCCESS; /* code de retour de l'ordonnanceur */
    ...
    while (goon)
    {
        struct task * ready = graph_find_ready (graph);
        if (ready != NULL && running < nproc) {
            if (task_execute (ready))
                ++ running;
            else
                status = EXIT_FAILURE;
        } else if (running > 0) {
            if (graph_standby (graph, ...) == 0)
                status = EXIT_FAILURE;
            -- running;
        } else {
            goon = 0; /* ou break si vous insistez */
        }
    }
    ...
    return status;
}
```


9. Algorithmique : en cas de problème système

- ▶ si `fork()` échoue? (2 cas : `ENOMEM` ou `EAGAIN`).

Version simple : dans `task_execute()`, la tâche échoue

case -1:

```
perror ("fork");  
task->pid = 0; /* On ne relancera pas cette tâche */  
fprintf (stderr, "INIT %s %d\n", task->name, task->pid);  
fprintf (stderr, "DEAD %s %d EXIT 256\n", task->name, task->pid);  
break;                               /* ~~~~~ convention discutable... */
```

Variante :

- ▶ ne plus essayer de lancer de tâches *momentanément*
- ▶ attendre quelques `wait()` (si `running > 0`)
- ▶ arrêter l'ordonnanceur (si `running == 0`)
- ▶ si `exec??()` échoue? (sûrement `ENOENT` ou `EACCESS`)
 - ▶ la tâche (processus fils) échoue (convention : `exit(127)`)
 - ▶ l'ordonnanceur poursuit « normalement »
- ▶ si `wait()` échoue?
 - ▶ soit un bug du programme, soit un désastre du système (avec une très faible probabilité pour le second cas)
 - ▶ on arrête tout immédiatement

10. Algorithmique : tâches bloquées et code de retour

- ▶ Si une tâche échoue :
 - ▶ ses successeurs (si elle en a) ne seront jamais prêts
 - ▶ le code de retour de l'ordonnanceur sera `EXIT_FAILURE`
- ▶ Inversement, si aucune tâche n'échoue :
 - ▶ toute tâche sera prête, tôt ou tard
(parce que le graphe est acyclique)
- ▶ Mais « code de retour `EXIT_FAILURE` » n'est *pas* équivalent à « il existe des tâches bloquées » (*stalled*)
 - ▶ parce qu'une tâche peut ne pas avoir de successeur
(en cas d'échec : `EXIT_FAILURE`, mais pas de tâche bloquée)
- ▶ Le code de retour est fixé pendant l'ordonnancement
(variable `status`, transparent 8)
- ▶ Le seul code nécessaire à la fin de `graph_exec` est :

```
for (struct task * t=graph->head ; t!=NULL ; t=t->next)
    if (t->pid == -1)
        fprintf (stderr, "STALLED %s\n", t->name);
```

11. Système : temps pris par un processus

- ▶ Deux notions : temps *utilisateur* et *système* (cf. cours ASE L3)
- ▶ Temps de l'ordonnanceur, ou temps des tâches ?
(et quand une tâche lance plusieurs processus ?)
- ▶ `getrusage` (`RUSAGE_CHILDREN`, ...) fournit
 - ▶ le temp *cumulé* des fils...
 - ▶ pour lesquels `wait()` a *déjà* été appelé
 - ▶ accumulation père-fils au moment de `exit()`

→ ce n'est pas une mesure instantanée

- ▶ Il faut donc faire des différences *entre* les appels à `wait()`
 - `avant` \leftarrow `getrusage ()`
 - `pid` \leftarrow `wait ()`
 - `apres` \leftarrow `getrusage ()`

Temps utilisé par `pid` = `apres` - `avant`

- ▶ Le temps est en secondes + μ secondes (`struct timeval`)
→ `timersub()`, `timerclear()`
(on ne peut *pas* faire la différence terme à terme...)

12. Code : mesurer le temps d'une tâche

```
int graph_exec (struct graph * graph, unsigned nproc)
{
    struct timeval ut_sofar = {0,0}; /* Temps utilisateur cumulé */
    struct timeval st_sofar = {0,0}; /* Temps système cumulé */
    ... (graph_standby (graph, &ut_sofar, &st_sofar) == 0) ...
}

int graph_standby (struct graph * graph, struct timeval * ut_sofar,
                  struct timeval * st_sofar)
{
    ... /* après wait (), task désigne la tâche terminée */
    struct rusage ru;
    if (getrusage (RUSAGE_CHILDREN, &ru) == -1) { /* pas si grave que ça */
        task->utime.tv_sec = task->stime.tv_sec = 0;
    } else {
        timersub (&ru.ru_utime, ut_sofar, &task->utime);
        timersub (&ru.ru_stime, st_sofar, &task->stime);
        *ut_sofar = ru.ru_utime;
        *st_sofar = ru.ru_stime;
    }
    fprintf (stderr, " %ld.%06ld %ld.%06ld\n", /* fin de ligne FINI/DEAD */
            task->utime.tv_sec, task->utime.tv_usec,
            task->stime.tv_sec, task->stime.tv_usec);
    return success;
}
```

13. Annexe : transformer un argument en nombre

1. Vite fait... mais fragile :

```
int nproc = atoi (argv[1]);
```

Aucune test d'erreur...

2. Simple, et pratique :

```
if (sscanf (argv[1], "%d", &nproc) != 1) {!.}
```

Vérifier des contraintes simples :

```
if (sscanf (argv[1], "%d", &nproc) != 1 ||  
    nproc < 1) {!.}
```

Permet aussi les lectures selon un format simple :

```
if (sscanf (str, "%d:%d:%d", &h, &m, &s) != 3) {!.}
```

3. L'arme absolue (utilisée par les deux précédentes) :

```
val = strtol (str, &endptr, base);
```

Mais le test d'erreur est déraisonnable (cf. exemple du manuel)