Projet Periodic : Tâches périodiques
Système et programmation système

Éric MERLET
Université de Franche-Comté

Licence 2 Informatique
2019 – 2020

Le but de ce projet est de réaliser un clone simplifié de `cron(8)` et `at(8)`, permettant de définir des actions à effectuer à intervalle de temps régulier. Le projet est à implémenter en langage C.

Spécifications

Le projet se compose de trois programmes principaux :
- un lanceur de daemon générique : `launch_daemon` (pouvant éventuellement servir pour lancer d'autres daemons)
- le programme exécuté par le daemon : `period`
- un outil en ligne de commande : `periodic`.

L'outil `periodic` permet, entre autres, d'envoyer au programme `period` une commande à exécuter ainsi que sa date de départ et sa période. Une période de 0 indique que la commande ne doit être effectuée qu'une seule fois (à la manière de `at(8)`). La syntaxe est la suivante :

```
./periodic start period cmd [arg]...
```

où `start` est l'instant de départ, `period` est la période (en secondes) et `cmd` est la commande à exécuter avec ses éventuels arguments (`[arg]...`). Les trois premiers paramètres sont obligatoires. L'instant de départ peut être spécifié de différentes manières :
- en indiquant le nombre de secondes depuis Epoch (1er janvier 1970) ;
- en indiquant la chaîne de caractères `now`, pour indiquer l'instant présent ;
- en indiquant le caractère `+` suivi du nombre de secondes avant de lancer la commande.

Quelques exemples :

```
./periodic 1586679518 0 echo Bonjour
# On souhaite exécuter 'echo Bonjour' le dimanche 12 avril 2020 à 10:18:38
# La commande ne sera exécutée qu'une seule fois

./periodic now 5 ls
# On souhaite exécuter 'ls' toutes les 5 secondes à partir de maintenant
```

```
./periodic +10 30 uptime
# On souhaite exécuter 'uptime' toutes les 30 secondes en commençant dans 10 secondes

./periodic +15 2 date +%H:%M:%S
# On souhaite exécuter 'date +%H:%M:%S' toutes les 2 secondes en commençant dans 15 secondes
```

Pour communiquer avec `period`, l'outil `periodic` utilisera un tube nommé `/tmp/period.fifo` et des signaux. Plus précisément, l'outil `periodic` enverra `SIGUSR1` ou `SIGUSR2` de manière à réveiller `period`, puis communiquera via le tube nommé.

Le programme `period` est chargé de l'exécution des différentes commandes. Pour cela, sa tâche principale est de dormir jusqu'à ce qu'une commande doive être exécutée. Il reçoit les commandes à exécuter de l'outil `periodic`. L'ensemble des commandes est placé dans une structure de données dont la taille évolue dynamiquement. Si vous n'êtes pas à l'aise avec les structures de données dynamiques, vous pouvez utiliser un tableau de 512 commandes (vous serez alors évalué sur 17). Dans la suite de cet énoncé, cette struture de données est appelée la liste des commandes.

Pour obtenir un daemon, il faut lancer le programme `period` en utilisant `launch_daemon` :

```
./launch_daemon  /absolute_path_to_period
```

Outils complémentaires et bibliothèque

Exercice 1 : Préliminaires

Question 1.1 Écrire en C un programme `now` qui affiche le nombre de secondes depuis Epoch pour l'instant présent. Indice : `time(2)`.

Question 1.2 Écrire en C un programme `when` qui prend en paramètre un nombre de secondes depuis Epoch et qui affiche la date à laquelle ce nombre correspond. Indice : `ctime(3)`.

Exercice 2 : Communication via un tube nommé

Le but de cette partie est de produire une bibliothèque dynamique `libmessage.so` qui contiendra des fonctions utiles aux programmes `period` et `periodic`. Ces fonctions seront utilisées pour communiquer des chaînes de caractères via un tube nommé (en utilisant un descripteur de fichier pointant sur une de ses extrémités). Il faudra faire un fichier d'en-tête `message.h` contenant les prototypes et un fichier source `message.c` contenant les définitions.

Question 2.1 Écrire une fonction `send_string` qui envoie une chaîne de caractères via un descripteur de fichier. Pour cela, on enverra d'abord la longueur de la chaîne puis son contenu. La fonction a le prototype suivant :

```
int send_string(int fd, const char *str);
```

Question 2.2 Écrire une fonction `recv_string` qui reçoit une chaîne de caractères via un descripteur de fichier. Pour cela, on recevra d'abord la longueur de la chaîne, puis on allouera l'espace nécessaire (via `malloc(3)` ou `calloc(3)`, attention au caractère '\0' final), puis on lira la chaîne. La fonction a le prototype suivant :

```
char *recv_string(int fd);
```

Question 2.3 Écrire une fonction `send_argv` qui envoie un tableau de chaînes de caractères (le dernier pointeur de ce tableau vaut NULL). On enverra d'abord la taille du tableau puis chacune des chaînes de caractères. La fonction a le prototype suivant.

```
int send_argv(int fd, char *argv[]);
```

Question 2.4 Écrire une fonction `recv_argv` qui reçoit un tableau de chaînes de caractères. On recevra d'abord la taille du tableau puis on allouera l'espace nécessaire (Attention : le tableau doit se terminer par un NULL), ensuite on lira chacune des chaînes de caractères. La fonction a le prototype suivant :

```
char **recv_argv(int fd);
```

Question 2.5 Avant de passer à la suite, tester, avec valgrind, les 4 fonctions de la bibliothèque en créant 2 programmes qui communiquent via un tube nommé. Cette étape est **absolument fondamentale** : vous ne pourrez pas rendre un projet qui fonctionne si ces fonctions ne sont pas correctement implémentées.

L'outil `periodic`

L'outil `periodic` peut être appelé de deux façons :

```
./periodic start period cmd [arg]...
./periodic
```

La première forme permet d'envoyer à `period` une nouvelle commande à exécuter. La seconde forme (sans argument) permet de récupérer la liste courante des commandes, mémorisée dans `period`, pour l'afficher.

Exercice 3 : Vérification de l'existance d'un processus exécutant `period`

Lorsqu'il démarre, le programme `period` enregistre son PID dans un fichier **texte** nommé `/tmp/period.pid`. L'existance ou la non existance de ce fichier permet de savoir si un processus est en train d'exécuter `period`. Ce fichier texte, quand il existe, permet donc de connaître le PID du processus qui exécute `period`, ce qui permettra ensuite de le "contacter" en lui envoyant un signal par exemple.

Question 3.1 Écrire une fonction qui permet de tester l'existance d'un processus exécutant `period`, et de lire le cas échéant son PID.

Exercice 4 : Envoi d'une nouvelle commande à exécuter

Pour envoyer à `period` une nouvelle commande à exécuter, on va d'abord le prévenir qu'on va lui envoyer des données en lui envoyant un signal, puis on va lui transmettre les informations nécessaires.

Question 4.1 Déterminer la date de départ et la période en fonction des arguments. On affichera une aide si une erreur a eu lieu (mauvais nombre d'arguments, arguments erronés). On utilisera `strtol(3)` pour convertir en entier les chaînes numériques passées en argument et détecter s'il y a une erreur.

Exemple de détection d'erreur :

```
$ ./periodic +10az 5 echo bonjour
invalid start
usage : ./periodic start period cmd [arg]...
usage : ./periodic
```

Question 4.2 Envoyer le signal `SIGUSR1` à `period` pour lui indiquer qu'il va devoir lire des données.

Question 4.3 Envoyer les informations nécessaires (date de départ, période, commande avec ses arguments) via le tube nommé `/tmp/period.fifo` en se servant de la bibliothèque que vous avez écrite.

Exercice 5 : Récupération de la liste courante des commandes

L'outil `periodic`, appelé sans argument supplémentaire, permet de récupérer la liste courante des commandes, c'est-à-dire l'ensemble des actions actuellement programmées.

Question 5.1 Envoyer le signal `SIGUSR2` à `period` pour lui indiquer qu'il va devoir écrire des données.

Question 5.2 En utilisant la bibliothèque, recevoir l'ensemble des commandes, puis les afficher.

Le programme `period`

Exercice 6 : Initialisations diverses

Question 6.1 Écrire le PID du processus dans le fichier **texte** `/tmp/period.pid`. Comme il ne faut pas qu'il y ait plus d'un processus qui exécute ce programme, si ce fichier existe déjà, le programme doit se terminer.

Question 6.2 Dans la version finale de ce programme, établir une redirection de la sortie standard vers le fichier `/tmp/period.out` et de l'erreur standard vers `/tmp/period.err`. Pour faciliter les tests, nous vous conseillons d'effectuer ses redirections qu'une fois que les programmes `period` et `periodic` sont au point.

Question 6.3 S'il n'existe pas, créer le tube nommé `/tmp/period.fifo`.

Question 6.4 S'il n'existe pas, créer le répertoire `/tmp/period`.

Exercice 7 : Définition des strutures de données

Question 7.1 Définir une structure de données pour chaque entrée (ou commande) de la liste.

Question 7.2 Définir une structure pour la "liste" (qui, pour rappel, peut être un simple tableau de 512 commandes) ainsi que les opérations associées : ajout, etc.

Exercice 8 : Ajout d'une commande dans la liste

Question 8.1 Quand le signal `SIGUSR1` est reçu, lire la nouvelle commande envoyée par l'outil `periodic` à travers le tube nommé, et faire les traitements nécessaires (ajout dans la liste, etc.).

Exercice 9 : Envoi de la liste courante

Question 9.1 Quand le signal `SIGUSR2` est reçu, envoyer l'ensemble des entrées de la liste à l'outil `periodic` à travers le tube nommé.

Exercice 10 : Gestion des commandes à exécuter

Question 10.1 Parcourir la liste pour déterminer la durée à attendre avant la prochaine action à effectuer, et prévoir une alarme.

Question 10.2 A la réception du signal `SIGALRM`, vérifier qu'il y a bien une (ou plusieurs) action(s) à effectuer et les lancer. Pour chaque action, on fera les redirections nécessaires : l'entrée standard sera redirigée vers `/dev/null` ; la sortie standard et l'erreur standard seront redirigées respectivement vers les fichiers `/tmp/period/X.out` et `/tmp/period/X.err` où `X` est le numéro d'enregistrement de la commande (1 pour la première commande enregistrée, 2 pour la seconde, etc.). Les deux fichiers `/tmp/period/X.out` et `/tmp/period/X.err` seront ouverts en ajout. Attention à bien être sûr que des actions n'ont pas été oubliées.

Exercice 11 : Gestion des fin de processus

Question 11.1 Gérer le signal `SIGCHLD`, qui est notamment envoyé à un processus quand un de ces processus fils est terminé, de manière à éliminer les zombis (bien vérifier que tous les processus zombies sont éliminés). Le processus père devra afficher, sur son erreur standard, comment ses fils se sont terminés.

Exercice 12 : Gestion de la fin du programme `period`

Question 12.1 Gérer les signaux `SIGINT`, `SIGQUIT` et `SIGTERM` afin que l'envoi de l'un de ces signaux termine "proprement" le processus qui exécute `period` :

- supprimer le fichier `/tmp/period.pid`
- libérer l'espace mémoire alloué dynamiquement
- éventuellement terminer et éliminer tous les processus créés restant

Exercice 13 : Gestion des signaux

Afin de réduire au maximum la taille du code exécuté dans les handlers de signaux, de ne pas utiliser de fonctions qui ne sont pas «async-signal-safe» dans un handler, et de ne pas devoir créer des structures de données globales (comme une liste chaînée) accessibles (notamment en modification) à la fois dans un handler et dans le programme principal, nous vous conseillons d'adopter l'architecture suivante :

```
/* déclaration de variables globales servant de drapeaux */
volatile sig_atomic_t usr1_receive = 0;
...

/* handler du signal SIGUSR1 */
void handSIGUSR1(int sig) {
  usr1_receive = 1;
}
...

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

  /* initialisations diverses */

  /* installation des handlers */

  while(1){
    // attente de réception d'un signal
    ...
    if (usr1_receive){
      // faire ce qu'il faut quand
      // le processus reçoit SIGUSR1
      usr1_receive = 0;
    }
    ...
  }

  ...
}
```

Le lanceur de daemon `launch_daemon`

Exercice 14 : Implémentation du lanceur de daemon

Question 14.1 Implémenter la méthode du double `fork(2)` pour lancer le daemon. Pour rappel, voici les actions à effectuer :

- Le processus principal forke et attends son fils
- Le fils devient leader de session via `setsid(2)`
- Le fils forke et se termine, rendant le petit-fils orphelin
- Le petit-fils n'est pas leader de session, ce qui l'empêche d'être rattaché à un terminal
- Le processus principal termine, laissant le petit-fils être le daemon

Explications supplémentaires :

- Le shell qui lance le processus principal crée un nouveau groupe de processus, dont le leader est le processus principal. Or pour créer une nouvelle session, un processus ne doit pas être leader de son groupe, car la création de la session passe par une étape de constitution d'un nouveau groupe prenant l'identifiant du processus appelant. Et un processus leader d'un groupe ne peut pas créer un nouveau groupe qui aurait comme identifiant son PID, car par définition, il existe déjà un groupe de processus ayant son PID pour identifiant.
- à partir du moment où le fils crée une nouvelle session, tous ses descendants (qui ne créent pas eux mêmes une nouvelle session) appartiennent à cette session.
- Tous les processus d'une session partagent (éventuellement) un même terminal de contrôle. Une session sans terminal de contrôle en acquiert un lorsque le leader de session ouvre un terminal pour la première fois. Ici, le leader de session se termine sans ouvrir de terminal. Donc la session, à laquelle appartient le petit fils, ne pourra plus jamais acquérir un terminal de contrôle.

Question 14.2 Changer la configuration du daemon.

- Changer le répertoire courant à `/` via `chdir(2)`
- Changer le umask à 0 via `umask(2)`
- Fermer tous les descripteurs de fichiers standard

Question 14.3 Charger le nouveau programme à exécuter : `period`

Bonus

Exercice 15 : Retrait d'une commande

Question 15.1 Ajouter la possibilité de retirer une commande de la liste.

Annexe : les tubes nommés

Un tube est un mécanisme de type **FIFO** (First In First Out) qui permet à deux processus d'échanger des informations de manière **unidirectionnelle** en mode **flot**. "En mode flot" signifie que les envois successifs de données dans un tube apparaîssent du point de vue de leur extraction comme une une seule et même émission.

Le schéma (fig. 1) illustre la possibilité de réaliser des opérations de lecture dans un tube sans relation avec les opérations d'écriture.

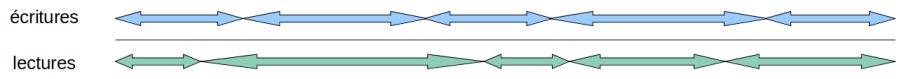

FIGURE 1 – Echange d'informations en mode flot

Ce mode de communication est à opposer à la communication en mode structuré (ou datagramme) dans lequel une lecture correspond exactement à une écriture.

Un tube **anonyme** est un tube qui n'existe que pendant la durée des processus qui l'utilisent, et disparaît ensuite. Il ne permet la communication qu'entre processus **ayant un lien de parenté**. On le crée en shell avec un `|`.

Les tubes **nommés** ont les mêmes caractéristiques que les tubes anonymes et ont la propriété supplémentaire d'être **référencés** dans le système de fichiers. C'est cette propriété qui permet à tout processus connaissant une référence d'un tube nommé d'obtenir en appelant `open(2)` un descripteur en lecture ou en écriture connecté à ce tube, pour communiquer avec d'autres processus par son intermédiaire. Ils permettent donc la communication entre processus **sans lien de parenté**. Un tube nommé est un tube qui persiste tant qu'il n'est pas explicitement détruit. On le crée en shell avec la commande `mkfifo(1)`.

Pour plus d'informations : `man 7 fifo`.

→ Exemple de création et d'utilisation d'un tube nommé en shell

Synopsis de la commande `mkfifo(1)` :

```
mkfifo [OPTION]... name...
```

Cette commande permet de créer un tube nommé dont le nom est `name`.

Premier exemple :

```
$ mkfifo foo
$ ls -l foo
prw-r--r-- 1 eric eric 0 avril 12 17:36 foo
$ rm foo
$
```

Les tubes nommés sont identifiables dans les résultats fournis par la commande `ls -l` par le caractère `p` correspondant à leur type.

Deuxième exemple :

```
$ mkfifo baz
$ gzip -9 -c < baz > out.gz
```

On commence par créer un tube nommé `baz` avec la commande `mkfifo(1)`. Ensuite le shell crée un nouveau processus pour exécuter la commande `gzip -9 -c`. Avant que le fils ne se recouvre pour exécuter la commande `gzip`, le shell doit rediriger l'entrée standard du processus fils vers la sortie du tube nommé, et sa sortie standard vers le fichier `out.gz`. Pour cela, il doit ouvrir en lecture seule le tube nommé : cette ouverture est **bloquante** jusqu'à ce qu'un autre processus réalise une ouverture en écriture du même tube nommé (cf. le paragraphe suivant).

On exécute alors dans un autre terminal :

```
$ cat file > baz
```

Pour exécuter cette commande, un autre shell (celui du second terminal) crée un nouveau processus pour exécuter la commande `cat file`. Avant que le fils ne se recouvre pour exécuter la commande `cat`, le shell doit rediriger la sortie standard du processus fils vers l'entrée du tube nommé `baz`. Pour cela, il ouvre ce tube nommé en écriture seule, ce qui a pour effet de débloquer le premier shell.
Les 2 shells finissent alors leur travail de redirection des descripteurs standard des 2 processus fils, et ces 2 processus fils se recouvrent et chargent en mémoire, pour un la commande `gzip`, et pour l'autre la commande `cat`.
Les 2 processus fils s'exécutent alors en concurrence. Quand le processus qui exécute `cat` se termine, le processus qui exécute `gzip` détecte une fin de fichier sur la sortie du tube nommé (son entrée standard) une fois qu'il est vide, et se termine alors à son tour.
Le fichier `out.gz` contient alors le résultat de la compression du fichier `file`.

→ Création d'un tube nommé en C

```
int mkfifo(const char *pathname, mode_t mode);
```

`mkfifo(3)` crée le tube nommé dont le nom est `pathname` avec les permissions `mode`.

→ Ouverture d'un tube nommé en C

- Une fois créé, le tube nommé peut être ouvert avec `open(2)` comme n'importe quel fichier.
- Pour être utilisé, un tube nommé doit être ouvert par deux processus, un en lecture et l'autre en écriture.
- L'ouverture d'un tube nommé **peut être bloquante** :
  - une demande d'ouverture en lecture est bloquante en l'absence d'écrivain sur le tube et de processus bloqué sur une ouverture en écriture
  - une demande d'ouverture en écriture est bloquante si il n'y a aucun lecteur sur le tube et aucun processus bloqué sur une ouverture en lecture

→ Exemple de création et d'utilisation d'un tube nommé en C

```
int main (void)
{
  if (mkfifo("baz", 0644)) {
    perror("mkfifo");
    exit(EXIT_FAILURE);
  }
  fd = open("baz", O_RDONLY);
  char buf;
  while (read(fd, &buf, 1) > 0){
      ...
  }
  close(fd);
  unlink("baz");
  return 0;
}
```

Ce premier programme commence par créer un tube nommé de nom `baz`. Ensuite, il réalise une demande d'ouverture en lecture seule de ce tube nommé. Cette demande d'ouverture est bloquante car, à l'instant où elle est réalisée, il n'y a ni écrivain dans ce tube nommé, ni aucun processus bloqué sur une demande d'ouverture en écriture.

```
int main (void)
{
  int fd = open("baz", O_WRONLY);
  ...

  const char *str = "hello world!";
  size_t lo = strlen(str);
```

```
  ssize_t nb = write(fd, str, lo);
  close(fd);
  return 0;
}
```

L'exécution de ce second programme est alors lancée. Ici, l'appel de `open(2)` retourne immédiatement car le premier programme est bloqué sur une demande d'ouverture en lecture du même tube nommé. L'appel de `open(2)` dans le premier retourne alors à son tour. On est donc ici en présence d'une **synchronisation** entre les 2 programmes : le premier programme qui fait une demande d'ouverture attend que l'autre fasse une demande d'ouverture dans l'autre sens (lecture / écriture) pour poursuivre son exécution.

Ensuite, les opérations de lecture de d'écriture dans le tube nommé fonctionnent comme dans un tube anonyme : le noyau assure une synchronisation des processus qui écrivent et lisent dans le tube :

- si le tube est plein, il bloque le processus écrivain
- si le tube est vide, et qu'il reste au moins un écrivain, il bloque le processus lecteur
- si le tube est vide, et qu'il y a plus d'écrivain, le lecteur détecte une fin de fichier (i.e. `read(2)` renvoie 0). C'est pourquoi, le processus lecteur ne sort de la boucle de lecture qu'après la fermeture du tube dans l'écrivain.

L'appel de `unlink(2)` correspond à une demande de suppression du lien physique `baz`.

Travailler avec Git et GitLab

Le projet est à réaliser en binôme, et comme pour le projet de LW, nous vous demandons de travailler avec Git et GitLab pour faciliter le travail à distance en binôme. NB : il peut être interessant de versionner son travail avec Git même quand on travaille seul !

Ceux qui ne suivent pas l'UE LW trouveront des informations sur l'utilisation de Git et GitLab sur Moodle :

`http://moodle.univ-fcomte.fr/pluginfile.php/1216749/mod_resource/content/4/gitv2.html`

→ Demander à Git d'ignorer des fichiers avec `.gitignore`

Les fichiers binaires (exécutables, *.o et *.so) ne doivent pas être versionnés : il ne faut pas enregistrer une nouvelle version de ces fichiers à chaque fois qu'ils sont modifiés suite à une construction du projet. On ne souhaite pas que la commande `git status` affiche que certains de ces fichiers ont été modifiés dès que vous exécutez la commande `make`. Seuls les fichiers source doivent être versionnés.

Pour ignorer un fichier dans Git (i.e. pour qu'il ne soit pas versionné), il faut indiquer le nom de ce fichier, dans un fichier `.gitignore` placé dans votre `working directory`. Il faut, dans ce fichier `.gitignore`, entrer un nom de fichier par ligne, comme ceci :

```
periodic
period
*.o
bin/
```

Le fichier `.gitignore` précédent demande à Git de ne pas versionner les exécutables `periodic` et `period`, ainsi que tous les fichiers ayant l'extension `.o` et tous les fichiers contenus dans le répertoire `bin/`.

Nous vous laissons compléter/modifier ce fichier `.gitignore` en fonction de vos besoins.

Évaluation

La note du projet tiendra compte des points suivants (liste non-exhaustive) :
— la séparation du projet en plusieurs unités de compilation et bibliothèque(s) ;
— la présence d'un Makefile fonctionnel ;
— les commentaires dans le code source (chaque fonction doit être précédée d'un bloc de commentaires indiquant notamment son rôle et les paramètres attendus) ;
— le contrôle des valeurs de retour des appels système et fonctions de la bibliothèque standard du langage C ;
— l'absence de fuites mémoire.

Comme pour le projet de LW, il n'y aura pas de rendu sur Moodle pour ce projet : à une date qui vous sera précisée ultérieurement, nous récupérerons tous vos projets depuis GitLab pour évaluation.

Votre rendu devra contenir un fichier README avec :
— une conclusion/bilan sur le produit final (ce qui est fait, testé, non fait)
— une présentation des améliorations possibles mais non réalisées
— un bilan par rapport au travail en binôme