Systèmes d'exploitation centralisés

1ère Année Sciences du Numérique

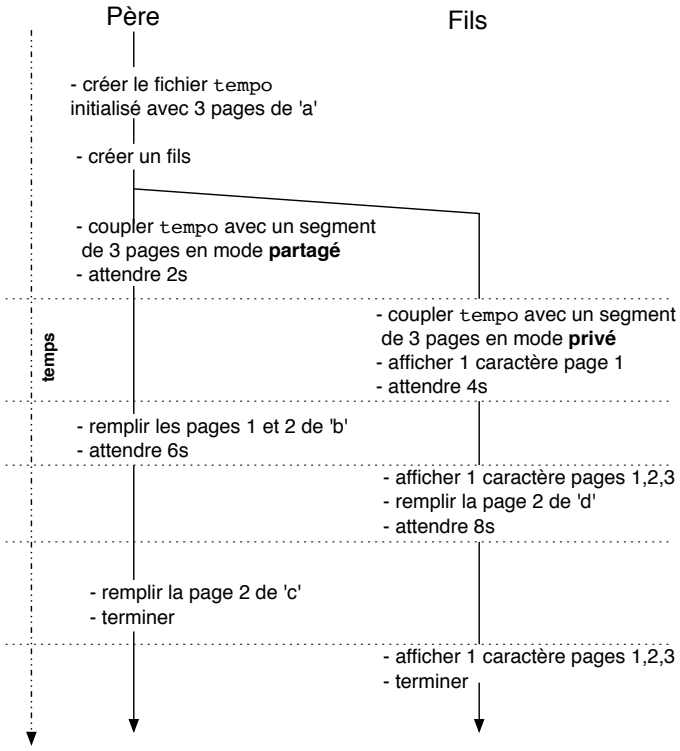
mai 2018

- Documents autorisés :
  2 feuilles de notes manuscrites originales (pas de photocopie) format A4, recto-verso.
- Tous appareils électroniques interdits
- Durée : 1 heure 30
- Barème : toutes les questions ont le même poids, à l'exception de la question 15.
  Les réponses non justifiées ne seront pas comptées.
- *Conseil* : il est préférable de **lire attentivement** l'énoncé avant de répondre.
- L'épreuve comporte **2 parties**, qui devront être **rendues séparément**.

1 Principes généraux (13 points)

1. Quelle est la différence entre la notion de processus et celle de programme ?
2. Pourquoi la notion de module est-elle particulièrement importante pour la conception des systèmes d'exploitation ? Donnez un exemple illustrant le recours à cette notion.
3. En matière de gestion mémoire, qu'est-ce que la fragmentation externe ? Quelle est sa cause ? Comment peut-on y remédier ?
4. Sous Unix, quelle est la différence entre les liens physiques (établis au moyen de la commande `ln`) et les liens symboliques (établis au moyen de la commande `ln -s`) ? Quel est l'intérêt des liens symboliques ?
5. Donnez deux stratégies générales de partage d'une ressource, et précisez comment elles se traduisent en matière d'allocation mémoire.
6. Comment les interruptions rendent-elles possible la multiprogrammation ?
7. Expliquez pourquoi il peut être intéressant de lancer deux instances d'un même système d'exploitation en utilisant la virtualisation.
8. Dans le domaine de l'allocation mémoire, en quoi consiste la politique LRU ?
9. Pourquoi estime-t-on que la politique LRU est une bonne approximation de la politique optimale ? LRU est-elle toujours la meilleure politique possible (Justifiez votre réponse) ?
10. On considère un processus sous Linux, qui crée un fichier `tempo` contenant trois pages de caractères 'a', ouvre ce fichier en lecture/écriture, et crée un processus fils.
    - Le processus père couple alors un segment de taille 3 pages à ce fichier en mode **partageable** et lecture/écriture, et attend 2 secondes. Ensuite, il remplit les pages 1 et 2 de caractères 'b', puis attend 6 secondes avant de remplir la page 2 de caractères 'c', et de terminer.
    - Le processus fils couple un segment de taille 3 pages au fichier `tempo` en mode **privé** et lecture/écriture, puis affiche le premier caractère de la page 1. Le processus fils attend alors 4 secondes, affiche le premier caractère de chacune des pages 1, 2 et 3, puis remplit la page 2 de caractères 'd', et attend 8 secondes avant d'afficher à nouveau le premier caractère de chacune des pages 1, 2 et 3, puis de terminer.

    Le schéma suivant présente une chronologie de l'exécution de ce programme.



Indiquez, en le justifiant, quel sera l'affichage produit par le fils.

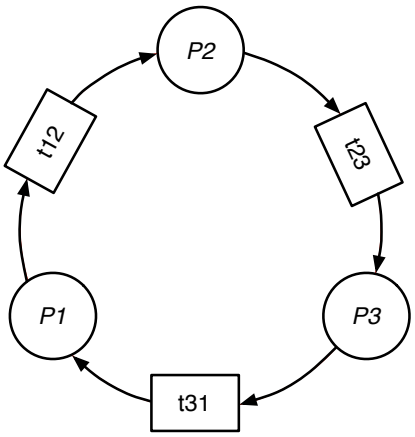
11. Lorsqu'un processus doit attendre la disponibilité d'une donnée (par exemple en sortie d'un tube (*pipe*)), il peut
    - répéter un test de disponibilité jusqu'à l'arrivée de la donnée (*scrutation*). Dans l'exemple du tube, cette option sera réalisée en répétant une lecture rendue non bloquante par appel à `fcntl(-)`.
    - demander à être suspendu, puis réveillé lorsque la donnée sera disponible. Cette option sera réalisée par l'appel d'une primitive système bloquante (lecture bloquante, ou `select(-)` dans le cas d'attente sur plusieurs descripteurs)

    Quels sont les avantages et les inconvénients de la scrutation par rapport à la suspension suivie du réveil ?
12. Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ? Justifiez votre réponse.
13. Du point de vue de l'application et du point de vue des mécanismes mis en jeu, quelles sont les différences entre la réception d'un signal et l'appel d'une procédure ?

Merci de répondre à cette partie sur une copie séparée.

2 API Unix (7 points)

On considère une application réalisée par trois processus communicant au moyen de tubes (pipes) selon une configuration en anneau illustrée par la figure suivante.



Cette structure est créée à partir du processus père *P1*. Celui-ci crée le processus *P2*, lequel crée à son tour le processus *P3*. Chacun des processus suit le même comportement qui consiste à lire un entier (jeton) sur son tube entrant, et à écrire cet entier sur son tube sortant. En outre, le père incrémente l'entier reçu, avant de le retransmettre. La circulation de ce jeton est initiée par le père, qui écrit la valeur initiale du jeton (zéro) sur son tube sortant, puis suit le comportement régulier décrit précédemment. La valeur courante du jeton représente donc le nombre de tours qu'il a accomplis.

14. Quel processus devra créer le tube *t31* ? Justifiez votre réponse.
15. **(3 points)** Programmer en C la création des processus et des tubes selon les indications précisées pour obtenir la structure d'anneau. La solution ne sera considérée comme correcte que si tous les descripteurs inutiles sont fermés et si les seuls processus créés sont *P1*, *P2*, *P3*.
16. Donnez la suite des commandes shell à saisir (éventuellement sur différents terminaux) afin de lancer ce programme, puis d'envoyer un signal `SIGUSR1` au processus *P2*.
17. Quel sera le comportement des différents processus après réception du signal `SIGUSR1` par *P2* ? Votre réponse doit être précise et complète.
18. Complétez ce programme afin qu'à la réception du signal `SIGUSR1`, le processus P1 – et seulement le processus P1 (les autres processus conservant le comportement par défaut) – affiche (sur la sortie standard) le nombre de tours déjà effectués par le jeton.

Annexes

Prototypes utiles (mais pas forcément nécessaires...)

Tous définis dans le fichier d'en-tête `unistd.h`. Par ordre alphabétique :

```
- int close(int desc);
- int dup2(int desc, int nv_desc);
- void exit(int n);
- pid_t fork();
- long lseek(int desc, long offset, long origine);
- int open (char *nomf, int option, int mode);
- int pipe(int desc[2]);
- int read(int desc, char *buf, int nb_oct);
- pid_t wait(int *term);
- int write(int desc, char *buf, int nb_oct);
```

Note : `printf` est définie dans `stdio.h`.

Page man de la primitive signal

NAME

signal — simplified software signal facilities

LIBRARY

Standard C Library (libc, –lc)

SYNOPSIS

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

or in the equivalent but easier to read typedef'd version:

```
typedef void (*sig_t) (int);

sig_t
signal(int sig, sig_t func);
```

DESCRIPTION

This **signal**() facility is a simplified interface to the more general sigaction(2) facility.

Signals allow the manipulation of a process from outside its domain, as well as allowing the process to manipulate itself or copies of itself (children). There are two general types of signals: those that cause termination of a process and those that do not. Signals which cause termination of a program might result from an irrecoverable error or might be the result of a user at a terminal typing the 'interrupt' character. Signals are used when a process is stopped because it wishes to access its control terminal while in the background (see tty(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals result in the termination of the process receiving them, if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the **signal**() function allows for a signal to be caught, to be ignored, or to generate an interrupt. These signals are defined in the file <signal.h>:

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))

24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

The *sig* argument specifies which signal was received. The *func* procedure allows a user to choose the action upon receipt of a signal. To set the default action of the signal to occur as listed above, *func* should be SIG_DFL. A SIG_DFL resets the default action. To ignore the signal, *func* should be SIG_IGN. This will cause subsequent instances of the signal to be ignored and pending instances to be discarded. If SIG_IGN is not used, further occurrences of the signal are automatically blocked and *func* is called.

The handled signal is unblocked when the function returns and the process continues from where it left off when the signal occurred. **Unlike previous signal facilities, the handler func() remains installed after a signal has been delivered.**

For some system calls, if a signal is caught while the call is executing and the call is prematurely terminated, the call is automatically restarted. Any handler installed with signal(3) will have the SA_RESTART flag set, meaning that any restartable system call will not return on receipt of a signal. The affected system calls include read(2), write(2), sendto(2), recvfrom(2), sendmsg(2), and recvmsg(2) on a communications channel or a low speed device and during a ioctl(2) or wait(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count). These semantics could be changed with siginterrupt(3).

When a process which has installed signal handlers forks, the child process inherits the signals. All caught signals may be reset to their default action by a call to the execve(2) function; ignored signals remain ignored.

If a process explicitly specifies SIG_IGN as the action for the signal SIGCHLD, the system will not create zombie processes when children of the calling process exit. As a consequence, the system will discard the exit status from the child processes. If the calling process subsequently issues a call to wait(2) or equivalent, it will block until all of the calling process's children terminate, and then return a value of –1 with *errno* set to ECHILD.

See sigaction(2) for a list of functions that are considered safe for use in signal handlers.

RETURN VALUES

The previous action is returned on a successful call. Otherwise, SIG_ERR is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **signal**() function will fail and no action will take place if one of the following occur:

[EINVAL] The *sig* argument is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

kill(1), kill(2), ptrace(2), sigaction(2), sigaltstack(2), sigprocmask(2), sigsuspend(2), wait(2), fpsetmask(3), setjmp(3), siginterrupt(3), tty(4)