UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Licence 3 Informatique
Info0601 – 2018/2019
***Resp. :* Cyril Rabat**

Devoir surveillé

Durée 2h

Aucun document, outil de calcul ou de communication autorisé. Sauf indications contraires, la gestion d'erreur ***ne doit pas*** *être réalisée et les inclusions ne doivent pas être spécifiées.*
Toute réponse doit être justifiée.

Exercice 1 (*Switch* applicatif - les tubes nommés)
Nous souhaitons réaliser une application, que nous appellerons *switch*, permettant à $n > 1$ applications indépendantes de communiquer entre elles. Le *switch* prend en argument le nombre n puis crée ensuite $2 \times n$ tubes nommés qui seront utilisés pour les communications avec les n applications. Chacune est identifiée par un numéro de 1 à n. Quand elle désire envoyer des données (quelconques, de tailles variables) à une autre application, elle les envoie au *switch* qui les transfert.

Première partie (4,5pts)

1. Aurait-il été possible d'utiliser des tubes anonymes ? Si oui, expliquer votre solution. Sinon, expliquez pourquoi.
2. Est-il possible qu'il y ait plus de n applications qui se "connectent" au *switch* ? Si oui, expliquez ce que cela entrainerait et proposez une solution pour l'éviter.
3. Aurait-il été possible d'utiliser seulement un tube par application pour les communications avec le *switch* ? Expliquez.
4. Indiquez quelles données (type, taille, taille maximum) sont envoyées dans les tubes.
5. Donnez la portion de code permettant de lire les données depuis un tube nommé supposé ouvert. Vous préciserez la déclaration des variables utilisées et/ou des variables supposées initialisées.
6. Une fois qu'il a créé tous les tubes, le *switch* peut-il tous les ouvrir en attendant que les applications soient exécutées ? Expliquez.

Deuxième partie (3,5 pts)

Nous supposons pour simplifier qu'une fois le *switch* exécuté, toutes les applications se "connectent" avant de commencer la moindre communication. Pour permettre les communications en parallèle, nous souhaitons que le *switch* crée des processus fils qui se focalisent chacun sur la réception des données d'une seule application. Le fils envoie ensuite les données vers la bonne application.

1. Expliquez tout ce que doit faire un fils du *switch*.
2. Quels problèmes de concurrence cela entraîne-t-il ? Comment les résoudre ?
3. Nous supposons que la fonction `fils` correspond à l'exécution d'un fils.
   (a) Donnez la signature de la fonction et expliquez les paramètres nécessaires.
   (b) Donnez le code du `main` du *switch* qui doit créer les tubes et les fils. Il attend ensuite la fin de chacun d'eux.

Exercice 2 (Application de log - les fichiers et les signaux - *9,5 pts*)
Nous souhaitons développer une application de log que nous appellerons le *logger*. Celui-ci se met en attente de la réception de signaux `SIGRTMIN` envoyés par d'autres applications (que nous appellerons les *clients*). Dès que le *logger* reçoit un tel signal, il crée un tube nommé dont le nom est `tube_PID.bin` où "`PID`" est le PID du *client*. Il se met alors en attente d'un message envoyé par le *client* (une chaîne de caractères de longueur

variable). Le *logger* stocke l'ensemble des messages reçus dans des fichiers binaires, un pour chaque *client*. Chaque fichier est nommé `PID.bin` où "`PID`" est le PID du client. Comme les messages peuvent être de tailles différentes, nous souhaitons utiliser une table de positions stockée au début du fichier. Cette table contient un nombre fixe d'entrées (nous utiliserons la constante `MAX_ENTREES`), chacune contenant l'horodatage (un *timestamp* Unix) et la position du message dans le fichier.

1°) Pourquoi est-il préférable que la table possède un nombre fixe d'entrées ? Expliquez comment faire pour stocker plus de `MAX_ENTREES` entrées dans le fichier.

2°) Proposez une structure en C pour les messages que vous nommerez `message_t`. Un message n'a pas de taille maximale.

3°) Donnez les structures en C de la table (nommée `table_t`) et d'une entrée (nommée `entree_t`).

4°) Donnez le code de la fonction `int ouvrir_fichier(pid_t pid)`. Si le fichier nommé `X.bin` (où X est le PID) n'existe pas, il est créé et préparé : il contient une table d'entrées vide. Si le fichier existe déjà, il est simplement ouvert. Dans les deux cas, la fonction retourne le descripteur du fichier correspondant.

5°) Nous supposons l'existence de la fonction `off_t rechercher_vide(int fd)` qui retourne la position dans le fichier de la prochaine entrée de table vide. Expliquez le fonctionnement de cette fonction.

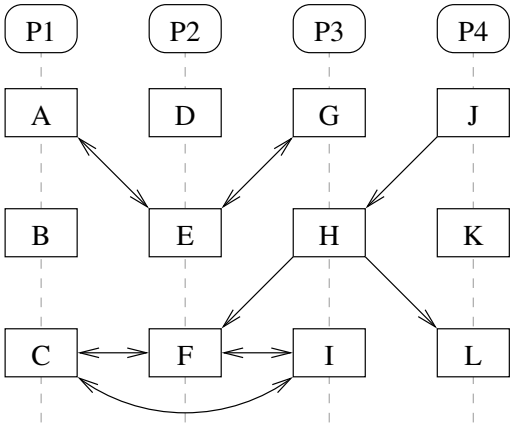
6°) Donnez le code de la procédure `void message_sauvegarder(message_t msg)` qui stocke le message passé en paramètre dans le fichier correspondant (utilisez les fonctions précédentes).

7°) Que se passe-t-il lors de l'ajout si le nombre de messages devient trop grand dans le fichier ?

8°) Proposez une solution pour corriger ce(s) problème(s).

9°) Discutez sur le choix d'avoir utilisé les noms de fichiers `PID.bin` où "`PID`" est le PID du client qui a envoyé le message.

Exercice 3 (Sémaphores de Dijkstra - *3,5 pts*)
Soit le diagramme de précédence suivant, représentant 4 processus P1, P2, P3 et P4 contenant chacun 3 blocs s'exécutant les uns après les autres :



1°) Proposer une solution à base de sémaphores pour résoudre ces contraintes (1 sémaphore pour chaque cas).

2°) Proposez une solution pour réduire le nombre de sémaphores.

3°) Expliquez pourquoi la solution proposée pour résoudre les contraintes entre A, E et G n'est pas satisfaisante.

Annexes : quelques en-têtes de fonctions C

```
void           _exit(int code);
unsigned int   alarm(unsigned int nb_secondes);
int            atexit(void (*procedure)(void));
void           clearerr(FILE *flux);
int            close(int fd);
int            closedir(DIR *repertoire);
int            creat(const char *chemin, mode_t mode);
int            dup(int ancien_fd);
int            dup2(int ancien_fd, nouveau_fd);
int            execve(const char *fichier, char *const argv[], char *const envp[]);
void           exit(int statut);
int            fcntl(int fd, int commande, ...);
int            fclose(FILE *flux);
FILE          *fdopen(int fd, const char *mode);
int            feof(FILE *flux);
int            ferror(FILE *flux);
int            fflush(FILE *flux);
int            fileno(FILE *flux);
FILE          *fopen(const char *chemin, const char *mode);
pid_t          fork();
size_t         fread(void *tampon, size_t taille, size_t nombre_elements, FILE *flux);
FILE          *freopen(const char *chemin, const char *mode, FILE *flux);
int            fseek(FILE *flux, long offset, int point_depart);
int            fstat(int fd, struct stat *tampon);
int            ftell(FILE *flux);
size_t         fwrite(const void *tampon, size_t taille, size_t nombre_elements, FILE *flux);
pid_t          getpid();
pid_t          getppid();
int            kill(pid_t pid, int numero_signal);
off_t          lseek(int fd, off_t offset, int point_depart);
int            lstat(const char *chemin, struct stat *tampon);
void          *memcpy(void *destination, const void *source, size_t taille);
void          *memset(void *tampon, int caractere, size_t nombre_elements);
int            mkfifo(const char *nomFichier, mode_t mode);
int            open(const char *chemin, int options);
int            open(const char *chemin, int options, mode_t mode);
DIR           *opendir(const char *chemin);
int            pause(void);
int            pipe(int tube[2]);
ssize_t        read(int fd, void *tampon, size_t taille);
struct dirent *readdir(DIR *repertoire);
void           rewind(FILE *flux);
void           rewinddir(DIR *repertoire);
void         (*signal(int numero_signal, void (*handler)(int)))(int)
int            sigaction(int numero_signal, const struct sigaction *action,
                         struct sigaction *ancienne_action);
int            sigaddset(sigset_t *ensemble, int numero_signal);
int            sigdelset(sigset_t *ensemble, int numero_signal);
int            sigemptyset(sigset_t *ensemble);
int            sigfillset(sigset_t *ensemble);
int            sigismember(sigset_t *ensemble, int numero_signal);
int            sigpending(sigset_t *ensemble);
int            sigprocmask(int comment, const sigset_t *ensemble, sigset_t *ancien);
int            sigqueue(pid_t pid, int numero_signal, const union sigval valeur);
int            sigsuspend(const sigset_t *ensemble);
int            sigwaitinfo(const sigset_t *ensemble, siginfo_t *info);
int            sigtimedwait(const sigset_t *ensemble, siginfo_t *info,
                            const struct timespec *timeout);
unsigned int   sleep(unsigned int nombre_secondes);
int            stat(const char *chemin, struct stat *tampon);
```

```
time_t          time(time_t *temps);
int             truncate(const char *chemin, off_t offset);
int             ftruncate(int fd, off_t offset);
int             truncate64(const char *chemin, off64_t offset);
int             ftruncate64(int fd, off64_t offset);
int             unlink(const char *nomFichier);
pid_t           wait(int *statut);
pid_t           waitpid(pid_t pid, int *statut, int options);
ssize_t         write(int fd, const void *tampon, size_t taille);
```

Annexes : constantes associées à des paramètres ou des champs de structures

options (open, fcntl) : O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_TRUNC, O_APPEND
mode (creat, open) : S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH
commande (fcntl) : F_GETFL, F_SETFL
mode (fopen, fdopen, freopen) : "r", "r+", "w", "w+", "a", "a+"
point_depart (lseek, fseek) : SEEK_SET, SEEK_CUR, SEEK_END
mode (mkfifo) : O_RDONLY, O_WRONLY, O_CREAT, O_EXCL, S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH
handler (signal, champ sa_handler de struct sigaction) : SIG_IGN, SIG_DFL
sa_flags (champ sa_flags de struct sigaction) : SA_ONESHOT, SA_NOMASK, SA_SIGINFO
comment (sigprocmask) : SIG_BLOCK, SIG_UNBLOCK, SIG_SET
options (waitpid) : WNOHANG, WUNTRACED

Annexes : macros

Sur **statut** de wait, waitpid : WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG
Sur le champ **st_mode** de struct stat : S_ISREG, S_ISDIR, S_ISFIFO

Annexes : quelques structures C

```
struct stat {
  dev_t      st_dev;
  ino_t      st_ino;
  mode_t     st_mode;
  nlink_t    st_nlink;
  uid_t      st_uid;
  gid_t      st_gid;
  dev_t      st_rdev;
  off_t      st_size;
  blksize_t  st_blksize;
  blkcnt_t   st_blocks;
  time_t     st_atime;
  time_t     st_mtime;
  time_t     st_ctime;
};
```

```
struct siginfo_t {
  int      si_signo;
  int      si_code;
  sigval_t si_value;
  pid_t    si_pid;
  ...
};
```

```
struct dirent {
  char d_name[256];
};
```

```
struct sigaction {
  void     (*sa_handler) (int);
  void     (*sa_sigaction) (int, siginfo_t*, void*);
  sigset_t sa_mask;
  int      sa_flags;
};
```

```
union sigval_t {
  int   sival_int;
  void *sival_ptr;
};
```

```
struct timespec {
  long tv_sec;
  long tv_nsec;
};
```