

## Examen "Programmation Système"

### 1 Questions de cours

Un dessin et/ou deux lignes d'explications suffisent pour répondre aux questions suivantes.

**QC 1.** Gestion des fichiers :

- Quel est le rôle d'un catalogue ?
- Entre un verrou interne et un verrou externe, quel est le plus efficace ? Pourquoi ?

**QC 2.** Gestion des interruptions logicielles :

- Quels est le mécanisme qui permet à un processus Unix de savoir qu'il a reçu un signal ?
- Expliquez brièvement comment on fait pour ignorer un signal
- Expliquez brièvement comment on fait pour capturer un signal

**QC 3.** Ordonnancement des processus

- Quelle est la différence entre un système préemptif et un système non préemptif ?
- Soient 3 processus A,B et C

Processus	Temps d'exécution	Temps d'arrivée
A	7	0
B	4	1
C	2	4

- Donnez le chronogramme d'exécution des ces 3 processus avec les méthodes PAPS (ou FIFO) , SJF et RR avec un quantum de temps de 2
- Quel est, du point de vue de l'utilisateur, le meilleur système ? Justifier votre réponse

## 2 Questions de TP

### 2.1 Sujet

Supposons que l'on ait un exécutable codage qui transcrit des caractères. On peut l'appeler sous 2 formats :

1. codage : lit ses données sur l'entrée standard 0 et écrit ses résultats sur la sortie standard 1
2. codage fichier\_source fichier\_cible : lit ses données dans le fichier `fichier_source` et écrit ses résultats dans le fichier `fichier_cible`

On veut faire un système de type client/serveur c'est à dire qu'il y aura au moins 2 processus **indépendants**, sans relation de parenté.

- un processus `serveur.c` qui offrira le service de codage à un ou plusieurs client(s)
- un processus `client.c` qui utilisera le service de codage du serveur.

Ce système sera réalisé pour deux types de transferts de données :

- les données sont des flux de caractères (sans début ni fin a priori)
- les données sont des fichiers (paquet de données < 2048 octets)

#### QTP 1. Communication par flux, c'est à dire

- que le client saisira ses données au clavier (entrée standard 0) et les enverra au serveur (le choix de la méthode fait partie du sujet). En même temps, le client affichera au fur et à mesure les résultats du serveur sur l'écran (sortie standard 1)
- le serveur recevra un flux de données qu'il redirigera dans l'exécutable codage avec le premier format. Le serveur renverra les résultats de codage (donc par flux aussi) au client.

- (a) Ecrivez le code des programmes `client.c`, `serveur.c` pour le traitement d'un seul client, ainsi que tous les fichiers headers (`#include <... .h>`) que vous jugerez nécessaires
- (b) Avec cette architecture et si nécessaire avec des modifications que vous expliquerez, pourra-t-on traiter plusieurs clients en parallèle ? Pourquoi ?
- (c) Avec cette architecture et si nécessaire avec des modifications que vous expliquerez, pourra-t-on traiter plusieurs clients séquentiellement ? Pourquoi ?

**QTP 2.** Communication par paquet, c'est à dire que

- le client
    - enverra tout un fichier source au serveur (le choix de la méthode fait partie du sujet). On supposera que le nom du fichier source est contenu dans la constante FICHIER\_SOURCE
    - attendra que le serveur ait envoyé le résultat de son codage
    - enregistrera le résultat du serveur dans un fichier cible. On supposera que le nom du fichier cible est contenu dans la constante FICHIER\_CIBLE
  - le serveur
    - attend un fichier à décoder d'un client
    - effectue le codage de celui ci avec l'exécutable codage dans le deuxième format
    - envoie le résultat en un seul paquet/fichier au client
- 
- (a) Ecrivez le code des programmes `client.c`, et `serveur.c` pour le traitement de plusieurs clients, ainsi que tout les fichiers headers que vous jugerez nécessaires
  - (b) Avec cette architecture et si nécessaire avec des modifications que vous expliquerez, pourra-t-on traiter plusieurs clients en parallèle ? Pourquoi ?
  - (c) Avec cette architecture et si nécessaire avec des modifications que vous expliquerez, pourra-t-on traiter plusieurs clients séquentiellement ? Pourquoi ?

## 2.2 Remarques

**Rem 1.** n'écrivez pas les bouts de codes relatifs aux traitements des erreurs

**Rem 2.** un rappel des syntaxes des principales primitives vues en cours/TD/TP est donné en § 3 page 4.

### 3 Rappel des primitives

#### 3.1 Gestion des tubes

Création d'un tube standard :

```
int pipe(int tube[2]);
```

Duplication d'un descripteur dans la 1<sup>ière</sup> entrée libre de la table des descripteurs locaux :

```
int dup(int descr);
```

Création d'un tube nommé :

```
int mknod(const char * nom_fichier, S_IFIFO, 0666);
```

#### 3.2 Gestion des BAL

Création/identification d'une file de messages :

```
int msgget(key_t cle, /* Cle de file de messages */
           int option); /* Option de creation */
```

Dépôt d'un message :

```
int msgsnd(int fileid, /* Identifiant de file */
           const void *p_mess, /* Pt sur message a envoyer */
           size_t lg, /* Longueur du message */
           int option); /* Option d'emission */
```

Retrait d'un message :

```
ssize_t msgrcv(int fileid, /* Identifiant de la file */
                void *p_mess, /* Pt sur zone reception */
                size_t lgmax, /* Longueur max prevue */
                long int type, /* Type de message attendu */
                int option); /* Option de reception */
```

Contrôle sur une file de messages :

```
int msgctl(int fileid, /* Identifiant de la file */
            int cmd, /* Operation a effectuer */
            struct msqid_ds *p_buf); /* Arguments de l'operation */
```

### 3.3 Gestion des fichiers

Ouverture

```
int open(const char *path, int oflag, 0666);  
avec oflag = O_RDONLY, O_WRONLY, O_RDWR
```

Fermeture, lecture et écriture

```
int close(int fildes);  
ssize_t read(int fildes, void *buf, size_t nbyte);  
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

### 3.4 Gestion des processus

Création dynamique d'un processus :

```
pid_t fork(void);
```

Synchronisation de processus :

```
pid_t wait(int *cr);
```

Recouvrements de processus :

```
int execl(const char *path, const char *arg0, ...,  
const char *argn, char * /*NULL*/);  
  
int execv(const char *path, char *const argv[]);  
  
int execle(const char *path, const char *arg0, ...,  
const char *argn, char * /*NULL*/, char *const envp[]);  
  
int execve(const char *path, char *const argv[],  
char *const envp[]);  
  
int execlp(const char *file, const char *arg0, ...,  
const char *argn, char * /*NULL*/);  
  
int execvp(const char *file, char *const argv[]);
```

### 3.5 Gestion des signaux

Envoi d'un signal :

```
int kill( pid_t pid, /* identificateur du processus */
          int sig); /* numero du signal */
```

Capture d'un signal :

```
void (*signal( int sig, /* numero du signal */
               void (*hand)(int) /* handler a lui associer */
             ))(int);

int sigaction( int sig,
               const struct sigaction * p_action,
               struct sigaction * p_action_ancienne );
```

avec

```
struct sigaction
{
    int sa_flags; /* options pour prise en compte */
    void (*_handler)(); /* handler de signal */
    sigset_t sa_mask; /* signaux a masquer a la prise en compte */
};
```

Manipulation des ensembles de masques :

sigmask(n)	Donne le masque du signal n
sigemptyset(S)	$S \leftarrow \emptyset$
sigaddset(S,n)	$S \leftarrow S \cup n$
sigdelset(S,n)	$S \leftarrow S - n$
sigismember(S,n)	Vrai ssi $n \in S$
sigorset(S1,S2)	$S1 \leftarrow S1 \cup S2$
sigandset(S1,S2)	$S1 \leftarrow S1 \cap S2$
sigdiffset(S1,S2)	$S1 \leftarrow S1 - S2$

avec  $S_i$  un ensemble de signaux et  $n$  un n° de signal