Communications inter-processus

Denis Conan, Michel Simatic et François Trahay

module CSC4508/M2

Avril 2018

Plan du document

1 Gestion des processus

1.1 Environnement des processus

- Environnement = liste de chaînes de caractères (couples `<nom>=<valeur>`)
- Variables exportées du shell
- Accessible à partir de `envp` ou de la variable externe `environ`

  `int main(int argc, char **argv, char **envp)`
  - Cette forme du `main` est aujourd'hui dépréciée
  - `envp` et `environ` possèdent la même valeur
- `getenv` récupère le contenu d'une variable d'environnement
  - `char *getenv(char *name)`
  - Valeur de retour = pointeur sur la chaîne `<valeur>` ou `NULL`

1.2 Informations concernant un processus

- Types d'informations :
  - `pid_t PID` : numéro du processus, unique et non modifiable
  - `pid_t PPID` : numéro du processus parent, unique et non modifiable
  - `gid_t PGID` : numéro du processus chef d'un groupe[a], unique et modifiable
  - `uid_t UID` : numéro de l'utilisateur réel, unique et non modifiable
  - `gid_t GID` : numéro du groupe de l'utilisateur réel, unique et non modifiable
  - `gid_t EUID` : numéro de l'utilisateur effectif, unique et modifiable
  - `gid_t EGID` : numéro du groupe de l'utilisateur effectif, unique et modifiable
- Primitives correspondantes :
  - `pid_t getpid()`, `pid_t getppid()`, `int getpgid()`, `int setpgid(pid_t pid, pid_t pgid)`
  - `uid_t getuid()`, `uid_t geteuid()`, `gid_t getgid()`, `gid_t getegid()`, `int setuid(uid_t id)`,`int setgid(gid_t id)`

a. Attention ! un groupe de processus repère un ensemble de processus dans une même session, ce n'est pas lié au groupe d'utilisateur.

1.3 Création de processus

- Héritage de presque tous les attributs du parent :
  - Segment `text` : copié ou partagé
  - Segment `data` : copié
  - `UID` / `GID` réels et effectifs : conservés
  - `PID` / `PPID` : modifiés
  - `PGRP` : conservé
  - Signaux interceptés : conservés
  - Descripteurs de fichiers / tubes : copiés
  - Temps U.C. : remis à 0
- Primitives correspondantes :

  `pid_t fork()`, `pid_t vfork()`

1.4 Terminaison de processus

- Terminaison :
  - Code retour du processus
  - Libération des segments de mémoire
  - Entrée dans la table des processus (cf. `/proc`) :
    - Gardée : stockage code retour, temps CPU...
    - Libérée par le processus parent (dans le `wait`)
- Attente de la terminaison de l'enfant :
  - Informations obtenues lors de la fin du processus enfant
    - Mode trace → octet poids fort = signal reçu, octet poids faible = `0xb1`
    - `exit` → octet poids fort = code retour enfant, octet poids faible = `0x00`
    - Signal → octet poids fort = `0xb1`, octet poids faible = signal reçu
  - On utilise plutôt les macros `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED` et `WTERMSIG`
- Primitives correspondantes :

  `void exit(int code_retour)`, `int atexit(void(*fonction_a_executer)())`

  `pid_t wait(int *code_retour)`, `pid_t waitpid(pid_t pid_enfant, int *code_retour, int options)`

1.5 Exécution d'un nouveau programme

- Modification de l'image (le segment de code) du processus courant
- Il n'est pas possible de revenir d'un `exec`
- Les instructions qui suivent un `exec` ne sont exécutées que si `exec` échoue
- Primitives correspondantes :

  `exec{l|v}[e|p]` → `man 3 exec`
  - `l` : liste d'arguments : fournir la liste (`char *arg0, char *arg1,..., 0`)
  - `v` : tableau d'arguments : fournir un pointeur sur le tableau (`char **argv` avec dernier élément = `'0'`)
  - `e` : passer un pointeur d'environnement (`char **envp`)
  - `p` : la variable d'environnement `PATH` est utilisée pour localiser le fichier (`char *nom_relatif`), sinon chemin absolu (`char *nom_absolu`)

2 Communications à l'aide de tubes

2.1 Principe des tubes

- Premier moyen d'échange de données entre processus sous UNIX
- Fichiers spéciaux gérés selon une méthodologie FIFO
- Taille limitée (typiquement, 64 Kio)
- Deux types de tubes :
  - Tubes ordinaires, non visibles dans l'arborescence, entre processus d'une même filiation
  - Tubes nommés, visibles dans l'arborescence, entre processus non forcément liés par filiation
- Création ou ouverture de tube retournent des descripteurs de fichiers ouverts gérés comme les autres fichiers

2.2 Tubes ordinaires (ou locaux)

- Accès par création ou héritage
- Perte par un processus = perte d'accès définitive par ce processus
- Position courante entièrement déterminée par les lectures/écritures
  - Primitive `lseek` interdite
- Taille maximum déterminée par `PIPE_BUF` définie dans `<limits.h>`
- Opération de lecture par défaut bloquante
  - Bloquant jusqu'à lecture de la taille demandée ou disparition de tous les écrivains
  - Pour non bloquante : utiliser `fcntl` avec le drapeau `O_NONBLOCK`
- Opération d'écriture atomique (tout est écrit) si opération bloquante
  - Signal `SIGPIPE` si aucun lecteur
- Primitive correspondante :

  ```
  int pipe(int descfich[2])
  ```

  ▶ 0 pour la lecture et 1 pour l'écriture

  → Se mettre d'accord pour l'utilisation (celui qui écrit, ceux qui lisent)

2.3 Tubes nommés

- Création par la primitive `mknod` qui permet de créer les fichiers spéciaux ou par `mkfifo`
  - `mknod` était préalablement réservée au super-utilisateur
- Ouverture possible par les processus connaissant le nom du tube
  - Par défaut, ouverture bloquante : lecteurs et écrivains s'attendent → synchronisation
- Suppression du tube lorsque explicitement demandé et pas d'ouverture en cours
  - Si suppression alors qu'il existe des lecteurs et écrivains, fonctionnement comme un fichier ordinaire
- Primitives correspondantes :

```
int mknod(const char *nom_fich, mode_t mode, dev_t n_periph)
```

  - Mode = `S_IFIFO` (ou 0010000) plus les droits sur les trois derniers octets

```
int mkfifo(const char *nom_fich, mode_t mode)
```

  - Mode = droits

3 Communication à l'aide des IPC POSIX

3.1 Inter Process Communication POSIX

- Trois types d'objets système:
  - Files de messages : échange de flots de données formatées
  - Mémoire partagée
  - Sémaphores : outils de synchronisation
- Utilisation d'une clé pour identifier les objets système
  - Chaîne de caractère de la forme `/nom_de_la_cle`

3.2 IPC System V versus POSIX IPC

- System V = une des premières versions majeures d'Unix. A servi de base pour l'élaboration du standard POSIX
- POSIX = Portable Operating System Interface (défini par IEEE)
- System V et POSIX définissent les IPC de manières différentes
- Pendant longtemps, implantation incomplète des IPC POSIX sur certains systèmes
- Les IPC System V ont perduré pendant longtemps
- Aujourd'hui IPC POSIX implantées correctement dans Linux et la plupart des Unix

3.3 Files de messages

- Création / destruction :

```
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct
mq_attr *attr)
int mq_unlink(mqd_t mq)
```

- Ouverture / fermeture d'une file déjà créée :

```
mqd_t mq_open(const char *name, int oflag)
int mq_close(mqd_t mq)
```

- Émission / réception (bloquante si file pleine et `oflag` ne contient pas `O_NONBLOCK`) :

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
unsigned msg_prio)
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
unsigned *msg_prio)
```

3.4 Mémoire partagée

- Création :
  `int shm_open(const char *name, int oflag, mode_t mode)`
  - Création (`oflag` contient `O_CREAT`) d'un objet de taille nulle
  - Retour = descripteur de fichier
- Changer la taille de l'objet mémoire partagée :
  `int ftruncate(int fd, off_t length)`
- Attachement :
  `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - `addr` = 0 → choix de l'adresse par le système d'exploitation
  - Une même région peut être attachée plusieurs fois à des adresses différentes
  - `prot` : droit d'accès (lecture/écriture/exécution)
  - `flags` : `MAP_SHARED` pour partager la zone mémoire avec un autre processus
  - `fd` : descripteur de fichier retourné par `shm_open`
  - `offset` : emplacement de départ de la zone dans la mémoire partagée
  - Retour : adresse d'attachement effective ou `MAP_FAILED`
- Détachement :
  `int munmap(void *addr, size_t length)`

- Fermeture :

```
int close(int fd)
```

- Destruction :

```
int shm_unlink(const char *name)
```

3.5 Sémaphores

3.5.1 Introduction aux sémaphores

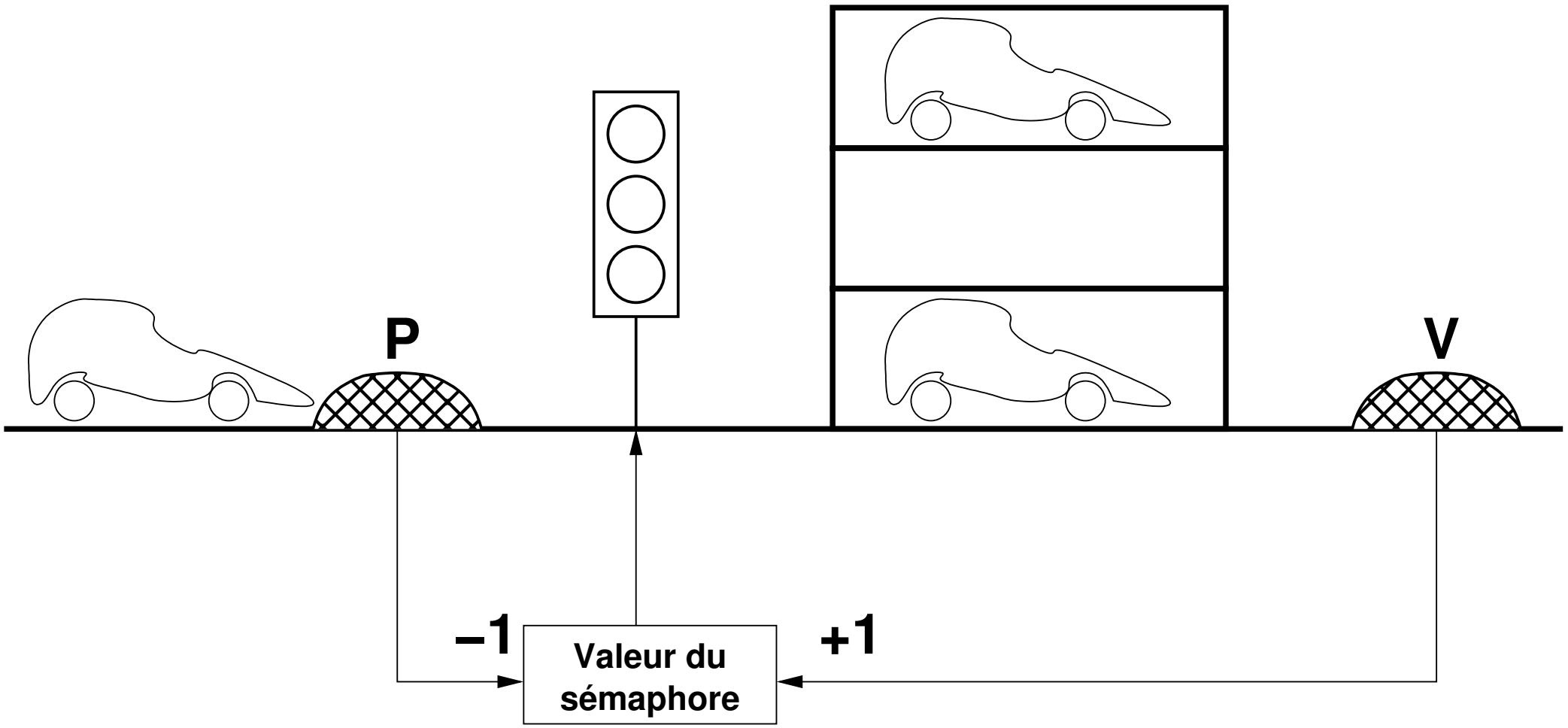
- Sémaphore = objet composé :
  - D'une variable (sa valeur)
  - D'une file d'attente (les processus bloqués)
- Primitives associées :
  - Initialisation (avec une valeur positive ou nulle)
  - Manipulation :
    - Prise (P ou Wait) = demande d'autorisation
    - Validation (V ou Signal) = fin d'utilisation
- Principe : sémaphore associé à une ressource
  - Prise = demande d'autorisation (Puis-je ?)

    si $valeur > 0$ accord, sinon blocage
  - Validation = restitution d'autorisation (Vas-y)

    si $valeur < 0$ déblocage d'un processus

3.5.2 Introduction aux sémaphores : analogie

- Parking de N places contrôlé par un feu



3.5.3 Introduction aux sémaphores : algorithmes P ET V

- Initialisation(sémaphore,n)

```
valeur[sémaphore] = n
```

- P(sémaphore)

```
valeur[sémaphore] = valeur[sémaphore] - 1
si (valeur[sémaphore] < 0) alors
   étatProcessus = Bloqué
   mettre processus en file d'attente
finSi
invoquer l'ordonnanceur
```

- V(sémaphore)

```
valeur[sémaphore] = valeur[sémaphore] + 1
si (valeur[sémaphore] <= 0) alors
   extraire processus de file d'attente
   étatProcessus = Prêt
finSi
invoquer l'ordonnanceur
```

3.5.4 Sémaphores POSIX : initialisation

2 types de sémaphores :

- Sémaphores nommés : identifiés par une clé de la forme `/un_nom`. Les sémaphores nommés sont persistants.
  - Création / ouverture :

    ```
    sem_t *sem_open(const char *name, int oflag, mode_t mode,
    unsigned int value)
    sem_t *sem_open(const char *name, int oflag)
    ```

  - Fermeture / destruction :

    ```
    int sem_close(sem_t *sem)
    int sem_unlink(const char *name)
    ```

- Sémaphores anonymes : placés dans une zone mémoire partagée par plusieurs threads ou processus (par exemple dans un segment de mémoire partagée).
  - Initialisation :

    ```
    int sem_init(sem_t *sem, int pshared, unsigned int value)
    ```

- Destruction :

```
int sem_destroy(sem_t *sem)
```

3.5.5 Sémaphores POSIX : utilisation

Une fois ouvert/initialisés les deux types de sémaphores peuvent être manipulés avec :

- Prise :

```
int sem_wait(sem_t *sem)
int sem_trywait(sem_t *sem)
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout)
```

- Validation :

```
int sem_post(sem_t *sem)
```

- Consultation :

```
int sem_getvalue(sem_t *sem, int *sval)
```

4 Comparaison des mécanismes de synchronisation

Temps écoulé pour envoyer un message via différents *media* de communication:

1. Tube pour transférer 4096 octets : $2,37$ µs
2. Tube pour transférer 416 octets en 2 écritures (longueur, puis donnée) : $1,27$ µs, soit un gain de 47%
3. Tube pour transférer pointeur (suppose malloc et free) : $2,20$ µs, soit un gain de 7%
4. Tableau de zones mémoire géré par paradigme de synchronisation Producteur/Consommateur avec des sémaphores POSIX : $0,94$ µs, soit un gain de 60%
5. Idem, mais avec des conditions POSIX : $3,29$ µs, soit un surcoût de 39%
6. File de messages (IPC et POSIX): $0,99$ µs, soit un gain de 58%