

Mécanismes de communication et de synchronisation

Introduction

- une application est typiquement constituée de plusieurs processus exécutées de façon concurrente
 - ces processus ne sont pas indépendantes
 - besoin d'échanger des données
 - besoin de synchroniser les moments où les données sont échangées

Introduction

- outils de communication
 - famille des files de messages
 - files de messages
 - tubes
 - FIFOs
 - mémoire partagée
- outils de coordination
 - famille des sémaphores
 - sémaphores binaire et à comptage
 - mutexes
 - famille des signaux
 - signaux

outils de communication

- BUT : échange d'informations entre différentes tâches
 - files de messages
 - mémoire partagée
- en utilisant éventuellement des moyens de synchronisation
- peuvent aussi servir de moyen de synchronisation
 - files de messages avec opérations d'accès bloquantes

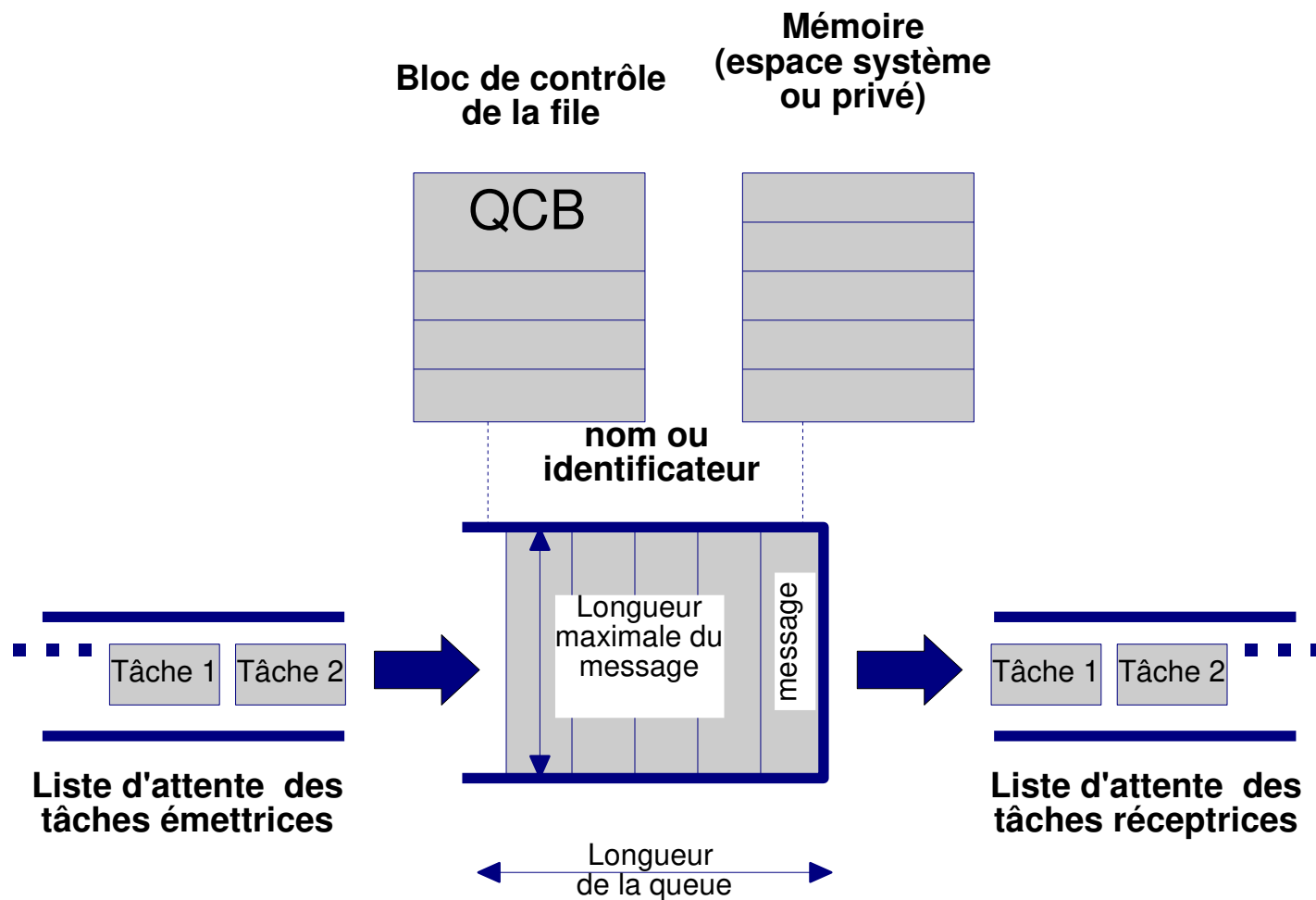
Files de messages : principe

Contrairement aux tubes une file de message contient des données “structurées” composées d’un message de **longueur variable**

- A la différence des pipes, on lit des messages de la taille envoyé sinon le processus sera **bloqué** (attente passive) : **frontières de message préservées**
- Comme les pipes, les files de message sont gérées selon un algorithme FIFO : on lit toujours le message du type cherché qui a séjourné le plus longtemps dans la file **mais avec un système de priorité**
- L’attente est **par défaut bloquante**, mais, comme pour les pipes, et si cela est nécessaire, le processus peut ne pas attendre en positionnant le drapeau IPC_NOWAIT. Il recevra un code d’erreur lui indiquant s’il a ou non obtenu un message.
- Comme les pipes, les files de messages ont des droits d’accès de type Unix (rwx,rwx,rwx)

Les files de messages

- C'est un objet de type **tampon** à travers lequel les tâches peuvent envoyer et recevoir des messages à des fins de communication ou de synchronisation



files de messages POSIX.4

Création

```
mqd_t mq_open(const char *mq_name, int oflags,  
              mode_t cr_mode,  
              struct mq_attr *attr);
```

`nom` "à la système de fichier" (cf sémaphores)

`oflags` réfère aux protections en lecture/écriture, à la politique de blocage (`O_NONBLOCK`) ou à la volonté de créer une nouvelle file (`O_CREAT` et `O_EXCL`)

`cr_mode` définit la protection sur le nom de la file

`attr` contient les informations physiques de la file (nombre maximum de messages, longueur maximum d'un message)

files de messages POSIX.4

- `#include <mqueue.h>`
- structure `mq_attr`

```
struct mq_attr
{
    long int mq_flags;      /* Message queue flags.  */
    long int mq_maxmsg;     /* Maximum number of
                             messages.  */
    long int mq_msgsize;    /* Maximum message size.  */
    long int mq_curmsgs;
    /* Number of messages currently queued.  */
};
```
- mode
 - `#include <sys/stat.h>`
 - protection (`I_IRUSR`, `I_IWUSR`, ...)

files de messages POSIX.4

- destruction

```
int mq_close(mqd_t mq_id);
```

- fermeture de la file

```
mq_unlink(const char *mq_name);
```

- destruction

- demande d'information

```
int mq_getattr(mqd_t mqid,  
               struct mq_attr *attr);
```

- modification des propriétés

```
int mq_setattr(mqd_t mqid,  
               const struct mq_attr *new_attr,  
               struct mq_attr *old_attr);
```

ne permet de modifier que la politique de blocage

files de messages POSIX.4

- écriture

```
int mq_send(mqd_t mqid, const char *msgbuf,  
            size_t msgsize, unsigned int prio);
```

- différents niveaux de priorité pour le message, compris entre MQ_PRIO_MAX et MQ_PRIO_MIN (définis dans <linux/mqueue.h> : **[0,32767]**)
- les messages sont insérés dans la file suivant la priorité, en mode FIFO à niveau égal de priorité
- pas d'effet sur la priorité de la tâche de lecture

- lecture

```
int mq_receive(mqd_t mqid, char *msgbuf,  
              size_t msgsize, unsigned int *prio);
```

- lecture du message de tête de la file (donc le plus prioritaire)

files de messages POSIX.4

- il existe des versions temporisées des fonctions d'écriture et de lecture

```
int mq_timedsend(mqd_t mqid, const char  
*msgbuf, size_t msgsize, unsigned int  
prio, constant struct timespec  
*abs_timeout);
```

```
int mq_timedreceive(mqd_t mqid, char  
*msgbuf, size_t msgsize, unsigned int  
*prio, constant struct timespec  
*abs_timeout);
```

**Echouent si le message n'est pas envoyé/lu avant la limite
(temps absolue)**

files de messages POSIX.4

- demande de notification

```
int mq_notify(mqd_t mqid,  
              const struct sigevent *specs);
```

- un événement POSIX.4 destiné à la tâche le demandant sera généré automatiquement si on écrit dans la file alors qu'elle est vide les caractéristiques de l'événement sont décrites dans la structure **sigevent**
- une seule tâche peut faire la demande de notification
- la demande ne peut se faire que si aucun processus n'est bloqué sur une opération de lecture sur la file
- il faut "réarmer" la demande de notification après exécution

files de messages POSIX.4

- Comportement sur **fork** et **exec**
 - analogue à celui des systèmes de fichier :
 - file héritée au clonage par **fork**
 - file fermée sur **exit** et **_exit**
 - comportement différent sur **exec** :
 - la file est fermée (alors qu'un fichier reste ouvert). Il faut donc refaire un appel à **mq_open**

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char * argv[]) {
mqd_t mq;
int  priorite;

if (argc != 4) {
    fprintf(stderr, "Syntaxe : %s file priorite message\n",
argv[0]);
    exit(EXIT_FAILURE);
}
if ((mq = mq_open(argv[1], O_WRONLY | O_CREAT, 0644,
NULL)) == (mqd_t) -1) {
    perror(argv[1]);
    exit(EXIT_FAILURE);}
}
```

```
if (sscanf(argv[2], "%d", & priorite) != 1) {  
    fprintf(stderr, "Priorite invalide : %s\n", argv[2]);  
    exit(EXIT_FAILURE);  
}  
if (mq_send(mq, argv[3], strlen(argv[3]), priorite) != 0)  
{  
    perror("mq_send");  
    exit(EXIT_FAILURE);  
}  
return EXIT_SUCCESS;  
}
```

```
$ ./exemple-mq-send aze 10 'Premier message'  
aze : Invalid argument  
$ ./exemple-mq-send /aze 10 'Premier message'  
$ ./exemple-mq-send /aze 20 'Deuxieme message'  
$ ./exemple-mq-send /aze 10 'Troisieme message'
```



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

exemple-mq-receive.c

(C) 2000-2010 - Christophe BLAESS

```
{
```

```
int      n;
```

```
mqd_t    mq;
```

```
struct mq_attr attr;
```

```
char * buffer = NULL;
```

```
unsigned int priorite;
```

```
if (argc != 2) {
```

```
    fprintf(stderr, "Syntaxe : %s file\n", argv[0]);
```

```
    exit(EXIT_FAILURE);
```

```
}
```

```
if ((mq = mq_open(argv[1], O_RDONLY)) == (mqd_t) -1) {
```

```
    perror(argv[1]);
```

```
    exit(EXIT_FAILURE);
```

```
}
```

```
if (mq_getattr(mq, & attr) != 0) {  
    perror("mq_getattr");  
    exit(EXIT_FAILURE);  
}  
if ((buffer = malloc(attr.mq_msgsize)) == NULL) {  
    perror("malloc");  
    exit(EXIT_FAILURE);  
}  
if ((n = mq_receive(mq, buffer, attr.mq_msgsize, &  
priorite)) < 0) {  
    perror("mq_send");  
    exit(EXIT_FAILURE);  
}  
fprintf(stdout, "[%d] %s\n", priorite, buffer);  
return EXIT_SUCCESS;  
}
```

```
$ ./exemple-mq-receive /aze  
[20] Deuxieme message  
$ ./exemple-mq-receive /aze  
[10] Premier message  
$ ./exemple-mq-receive /aze  
[10] Troisieme message  
$ ./exemple-mq-receive /aze
```

Bloqué en attente

Mémoire partagée

- moyen de communication très efficace entre tâches
 - implémenté naturellement pour les threads
 - Rapide
- mais **dangereux**
 - **accès concurrent** à une même ressource
- nécessite l'utilisation des outils de **synchronisation** (mutex et/ou sémaphores)
- seul moyen par lequel des outils de synchronisation anonymes de POSIX (mutexes et sémaphores) peuvent être partagés entre des modules exécutés dans l'espace noyau et des processus utilisateurs, ou entre des processus utilisateurs différents

Mémoire partagée POSIX.4

- Complexe...
- parce qu'elle utilise la possibilité de projeter en mémoire des fichiers
 - mécanisme très puissant de partage et de sauvegarde de données, même entre machines distantes (il suffit qu'elles partagent un même disque)
- deux étapes :
 - ouvrir (créer) l'objet de mémoire partagée (**shm_open**)
 - utiliser le descripteur retourné pour projeter cet objet dans l'espace mémoire de la tâche (**mmap**)

Mémoire partagée POSIX.4

```
#include <sys/mman.h>
```

- création d'un segment de mémoire partagée

```
int shm_open(const char *name, int oflag,  
             mode_t mode);
```

- retourne un descripteur analogue à celui d'un fichier, avec les mêmes restrictions que pour les sémaphores et les files de messages

- `oflag` et `mode` sont équivalents à ceux utilisés par `open`

- modification de la taille du segment

```
int ftruncate (int fd, off_t size);
```

- la taille du segment est nulle à la création
- `ftruncate` est une fonction "généraliste" qui agit également sur les fichiers, pour augmenter ou diminuer la taille

Mémoire partagée POSIX.4

- fermeture et destruction

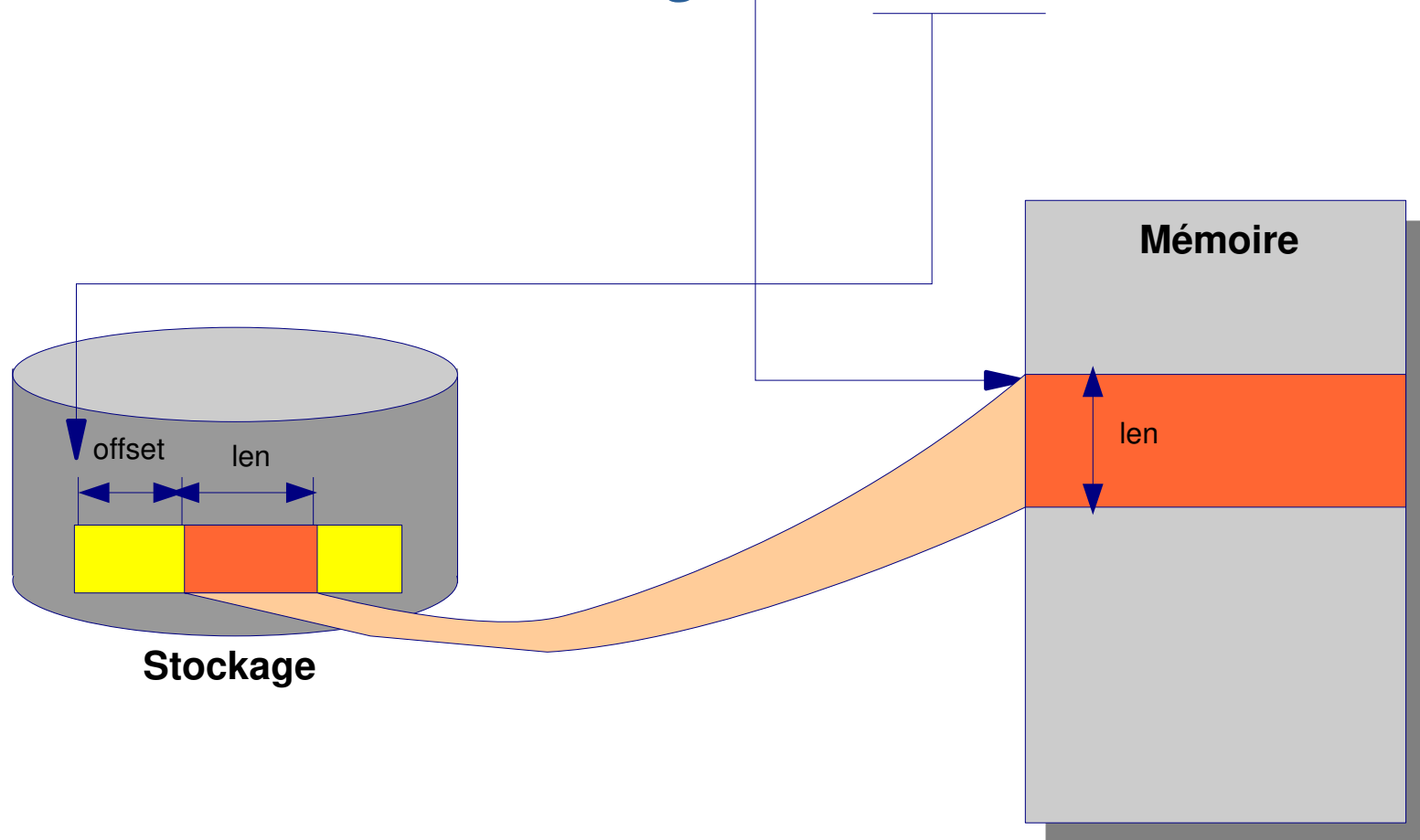
```
close(int fd);
```

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

Mémoire partagée POSIX.4

- projection de la mémoire partagée

```
caddr_t mmap(caddr_t addr, size_t len, int prot,  
             int flag, int fildes, off_t offset);
```



Mémoire partagée POSIX.4

- projection de la mémoire partagée

```
void* mmap(caddr_t addr, size_t len, int  
prot, int flag, int fildes, off_t offset);
```

projection dans l'espace d'adressage de l'objet associé au descripteur **fildes** des **len** octets à partir du décalage **offset**

mmap permet de projeter d'autres objets que des segments de mémoire partagée (par exemple, des fichiers “classiques”)

addr est une proposition d'adresse (souvent **NULL**).
L'adresse effectivement choisie par le système est retournée par **mmap** (**add** = **mmap**(...) contiendra l'adresse effective)
il est recommandé de prendre des multiples de **PAGESIZE** pour **len**

Mémoire partagée POSIX.4

- projection de la mémoire partagée

```
void* mmap(caddr_t addr, size_t len, int  
prot, int flag, int fildes, off_t offset);
```

- **len** longueur en octets
- **offset**
 - il est recommandé de prendre des multiples de **PAGESIZE** pour **len**
- **prot**
 - **PROT_EXEC**
On peut exécuter du code dans la zone mémoire.
 - **PROT_READ**
On peut lire le contenu de la zone mémoire
 - **PROT_WRITE**
On peut écrire dans la zone mémoire.
 - **PROT_NONE**
Le contenu de la zone memoire est inaccessible.

Mémoire partagée POSIX.4

- projection de la mémoire partagée

```
void* mmap(caddr_t addr, size_t len, int  
prot, int flag, int fildes, off_t  
offset);
```

- flag

MAP_FIXED

N'utiliser que l'adresse indiquée. Si c'est impossible, mmap échouera.

MAP_SHARED

Partager la projection avec tout autre processus utilisant l'objet. L'écriture dans la zone est équivalente à une écriture dans le fichier.

MAP_PRIVATE

Créer une projection privée

Mémoire partagée POSIX.4

```
int munmap (void * addr, size_t  
len);
```

- annulation de la projection

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
```

```
int main (int argc, char * argv[]) {
    int    fd;
    int * compteur;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_segment\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = shm_open(argv[1], O_RDWR | O_CREAT, 0600)) == -
1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
}
```

```
if (ftruncate(fd, sizeof(int)) != 0) {
    perror("ftruncate");
    exit(EXIT_FAILURE);
}

compteur = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
if (compteur == MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
}

while (1) {
    (* compteur) ++;
    fprintf(stdout, "compteur = %d\n", (* compteur));
    sleep(1);
}

return EXIT_SUCCESS;
}
```

```
$ ./exemple-shm /abc  
compteur=1  
compteur=2  
compteur=3  
CTRL-C  
$ ./exemple-shm /abc  
compteur=4  
compteur=5  
compteur=6  
CTRL-C
```

Persistence jusqu'au reboot

```
$ ./exemple-shm /abc  
compteur=7  
compteur=8  
compteur=10
```

```
$ ./exemple-shm /abc  
compteur=9  
compteur=11
```

Dans deux terminaux distincts

Ici pas de contrôle de l'accès concurrent...

Utilisation de MAP_ANONYMOUS

```
int * addr = mmap(NULL, sizeof(int),  
PROT_READ | PROT_WRITE, MAP_SHARED |  
MAP_ANONYMOUS, -1, 0);
```

Quand il n'est pas nécessaire de passer par un nom de fichier (donc pas de descripteur de fichier donné par ~~open ou shm_open~~), il est possible d'utiliser cette forme avec `-1` comme descripteur de fichier

Pas de nom donc partage par héritage père/fils (`fork`)

Outils de synchronisation

Outils de coordination

- but : protéger l'accès à des variables partagées
- exemple : gestion d'un stock
 - un processus lit dans un registre la valeur courante V_0 du stock et veut la décrémenter
 - un autre processus préempte le premier avant, lit la valeur courante (toujours V_0) , la décrémente et met à jour le registre qui contient alors V_0-1
 - le premier processus reprend son exécution et met à jour le registre en écrivant V_0-1 , au lieu de V_0-2

Pourquoi utiliser un sémaphore ?

D'une manière générale, un sémaphore sert à la **synchronisation** entre les processus

Un sémaphore utilisé pour **l'exclusion mutuelle** (garantir qu'un seul processus accède à un moment donné à une variable partagée), utilisera 2 primitives

Attendre l'accès exclusif à la ressource partagée : si la ressource est déjà utilisée par un processus, le processus qui fait cette demande sera endormi et mis dans une file d'attente de processus

Libérer l'accès exclusif à la ressource partagée : si la file d'attente des processus attendant cette ressource est non vide, le processus qui a dormi le plus longtemps est réveillé

Pourquoi utiliser un sémaphore ?

La section de programme qui se trouve entre les 2 primitives **attendre** la ressource et **libérer** la ressource s'appelle une **section critique**

Les sémaphores n'empêchent pas les processus d'accéder à une ressource: un processus « **qui ne respecterait pas la discipline** » et accéderait à la ressources sans attendre son tour pourrait le faire sans aucune restriction

Aux sémaphores sont associés des **droits d'accès** de type Unix

Le "mécanisme" des sémaphores

- Sémaphore = objet composé :
 - une variable (la valeur du sémaphore)
 - une file d'attente (les processus bloqués attendant la ressource)
- Primitives associées :
 - Initialisation (avec une valeur positive ou nulle)
- Manipulation :
 - **Prendre** (P ou Wait) = demande d'autorisation
 - **Vendre** (V ou Signal) = fin d'utilisation

Le "mécanisme" des sémaphores

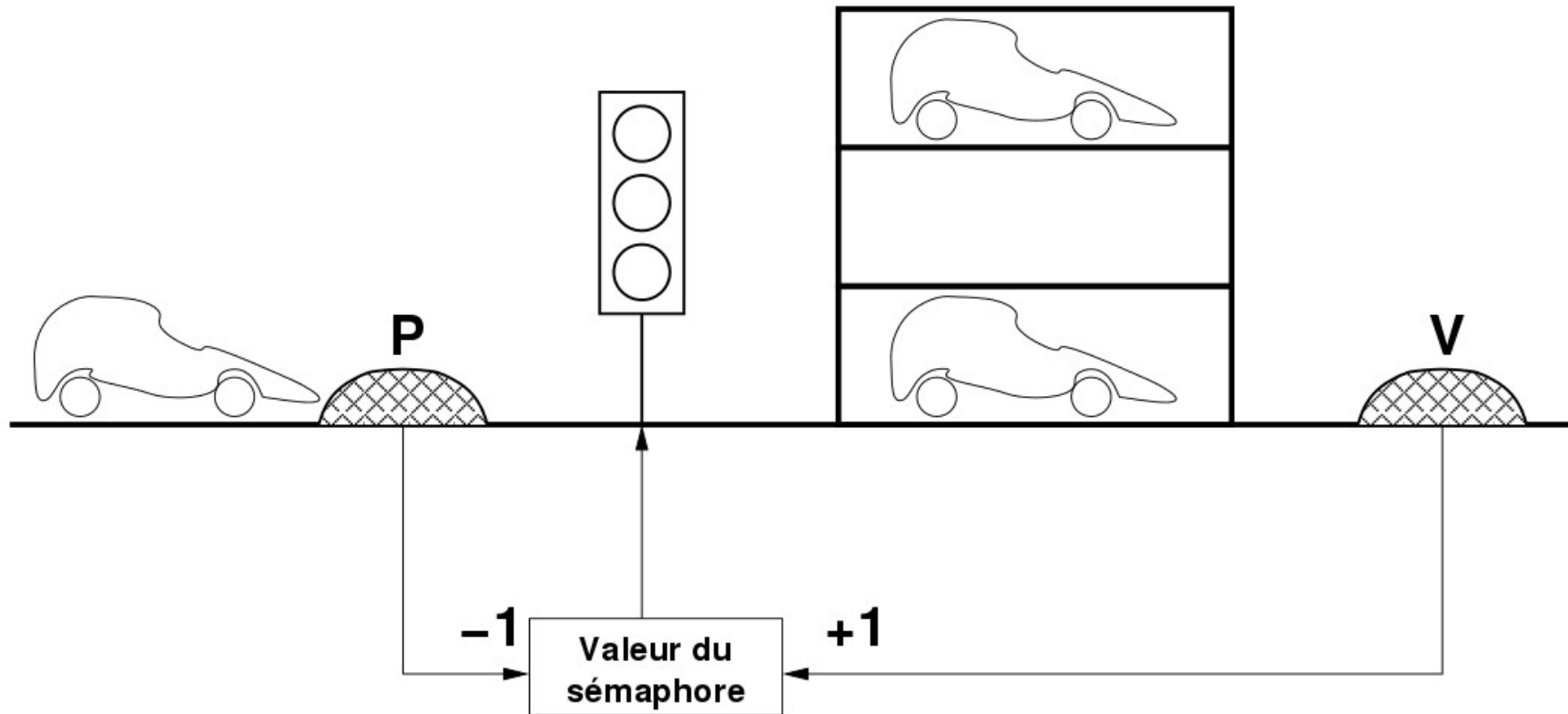
Principe : sémaphore associé à une ressource

Prendre = demande d'autorisation
(puis-je utiliser la ressource?)
Si valeur > 0 accord, sinon blocage

Vendre = restitution d'autorisation
(je n'ai plus besoin de la ressource)
eventuellement déblocage d'un processus

Sémaphores : exemple

Parking de N places contrôlé par un feu



Sémaphores : algorithmes des primitives P et V

Initialisation(sémaphore,n)

```
valeur[sémaphore] = n
```

P(sémaphore)

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

Sémaphores : algorithmes des primitives P et V

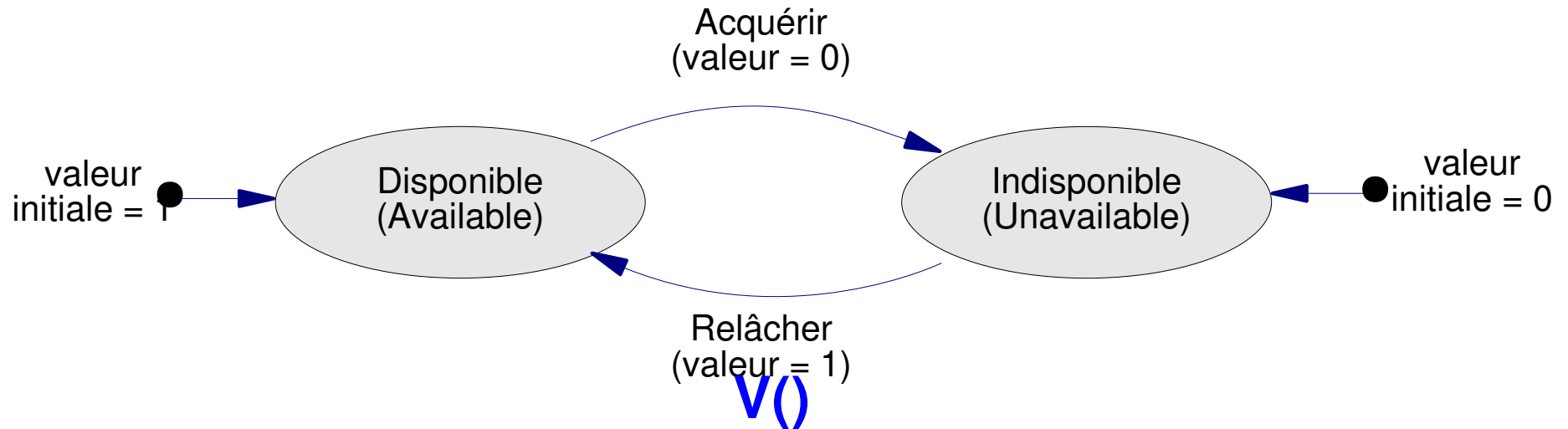
V(sémaphore)

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

Les différents types de sémaphores

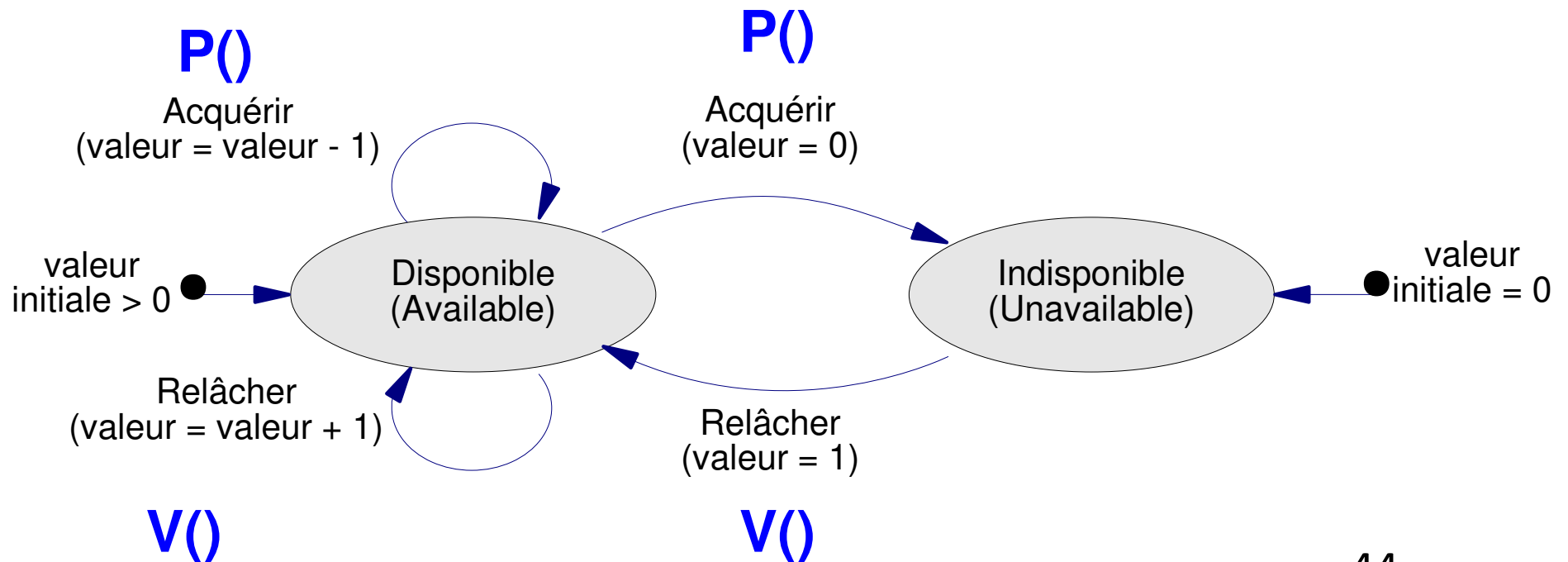
- sémaphores binaires
 - valeur : 0 ou 1

P()



Les différents types de sémaphores

- sémaphores à compte
 - valeur initiale positive ou nulle
 - décrémentée à chaque fois que l'acquisition est accordée, incrémentée quand le sémaphore est relâché
 - quand la valeur devient nulle, la demande d'acquisition est bloquante
 - ressource globale



Les différents types de sémaphores

- mutex
 - sémaphores binaires améliorées
 - les détails d'implémentation peuvent varier
 - Voir Threads POSIX

Sémaphores POSIX.4

- création
- destruction
- acquisition
- libération

Sémaphores POSIX.4

- **sémaphores**
 - `#include <semaphore.h>`
 - `-lrt` et `-lpthread` pour gcc
 - existent sous deux formes :
 - **sémaphores nommés**
 - **sémaphores anonymes**
- **sémaphores nommés**
 - mécanisme de gestion analogue à celui d'un système de fichiers : **accessible par tous les processus** (selon les droits)
- **sémaphores anonymes (basés sur la mémoire)**

Sémaphores nommés

- **sémaphores nommés**

```
sem_t *sem_open(const char *sem_name,  
                int oflags, mode_t creation_mode,  
                unsigned int initial_val);
```

crée le sémaphore

- **sem_name** : il est *recommandé* de fournir un nom "compatible" avec un nom de fichier, et commençant par un "/" , et n'en contenant pas d'autre (ex : `"/bidule"`)
- **oflags** : `O_CREAT` pour créer un sémaphore
`O_CREAT` et `O_EXCL` erreur si existe déjà
- **initial_val** : valeur initiale
- **creation_mode** : comme pour open
- **Retour** :
 - adresse du nouveau sémaphore
 - si echec SEM_FAILED + errno

- S_IRWXU
00700 L'utilisateur (propriétaire du fichier) a les autorisations de lecture, écriture, exécution.
- S_IRUSR
00400 L'utilisateur a l'autorisation de lecture.
- S_IWUSR
00200 L'utilisateur a l'autorisation d'écriture.
- S_IXUSR
00100 L'utilisateur a l'autorisation d'exécution.
- S_IRWXG
00070 Le groupe a les autorisations de lecture, écriture, exécution.
- S_IRGRP
00040 Le groupe a l'autorisation de lecture.
- S_IWGRP
00020 Le groupe a l'autorisation d'écriture.
- S_IXGRP
00010 Le groupe a l'autorisation d'exécution.
- S_IRWXO
00007 Tout le monde a les autorisations de lecture, écriture, exécution.
- S_IROTH
00004 Tout le monde a l'autorisation de lecture.
- S_IWOTH
00002 Tout le monde a l'autorisation d'écriture.
- S_IXOTH
00001 Tout le monde a l'autorisation d'exécution.

Sémaphores nommés

```
sem_t *sem_open(const char *sem_name,  
                int oflags);
```

accède à un sémaphore déjà créé

```
int sem_close (sem_t *sem_id);
```

ferme le sémaphore

(mais le nom reste : noyau)

- Retour :
 - renvoie 0.
 - Si erreur : -1 + errno

```
int sem_unlink(const char *sem_name);
```

supprime le sémaphore. Le nom du sémaphore est supprimé. Le sémaphore est détruit une fois que tous les autres processus qui l'avaient ouvert l'ont fermé qui peuvent donc l'utiliser jusqu'à cette fermeture.

- Retour :
 - renvoie 0.
 - Si erreur : -1 + errno

Sémaphores anonymes

- **sémaphores anonymes (basés sur la mémoire)**
`int sem_init(sem_t *sem_location, int pshared, unsigned int initial_value);`
crée le sémaphore
 - **sem_location** est une zone mémoire, éventuellement en zone partagée
 - **pshared** indique la visibilité des sémaphores (un ou plusieurs processus) :
 - si le sémaphore est local au processus courant (`pshared==0`) ou partagé entre plusieurs processus (`pshared!=0`)
 - Retour :
 - renvoie 0.
 - Si erreur : -1 + `errno`

Sémaphores anonymes

`int sem_destroy(sem_t sem_location);`
détruit le sémaphore

- Retour :
 - renvoie 0.
 - Si erreur : -1 + errno

Sémaphores POSIX.4

- **opérations sur les sémaphores**
mêmes primitives pour les deux types de sémaphores

V() :

```
int sem_post(sem_t *sem_id);
```

incrémente le compteur et s'il devient positif réveille une tâche en attente.

Jamais bloquant.

- Retour :
 - renvoie 0.
 - Si erreur : -1 + errno

Sémaphores POSIX.4

P() :

```
int sem_wait(sem_t *sem_id);
```

décrémente (verrouille) le sémaphore pointé par sem.

Si > 0 : décrémentation et retour immédiat.

Si zéro, appel bloquant jusqu'à ce que :

- soit il devienne disponible pour effectuer la décrémentation
(c'est-à-dire la valeur du sémaphore n'est plus nulle)
- soit un gestionnaire de signaux interrompe l'appel :
un gestionnaire de signaux interrompra toujours un appel bloqué à l'une de ces fonctions
- Retour :
 - renvoie 0.
 - Si erreur : -1 + errno

Sémaphores POSIX.4

P() :

```
int sem_trywait(sem_t *sem_id);  
idem sem_wait(),
```

mais si la décrémentation ne peut pas être effectuée immédiatement, l'appel n'est **pas bloquant** et renvoie une erreur (errno vaut EAGAIN).

Sémaphores POSIX.4

P() :

- **int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);**
Idem `sem_wait` mais `abs_timeout` spécifie une limite sur le temps pendant lequel l'appel bloquera si la décrémentation ne peut pas être effectuée immédiatement.

`abs_timeout` pointe sur une structure qui spécifie un temps absolu en secondes et nanosecondes depuis l'Époque (« epoch »)
(1er janvier 1970, 00:00:00):

```
struct timespec {  
    time_t tv_sec;           /* Secondes */  
    long   tv_nsec;         /* Nanosecondes [0 ..  
999999999] */  
};
```


Sémaphores POSIX.4

Si le délai est déjà expiré à l'heure de l'appel et si le sémaphore ne peut pas être verrouillé immédiatement, `sem_timedwait()` échoue avec l'erreur d'expiration de délai (`errno` vaut `ETIMEDOUT`).

Si l'opération peut être effectuée immédiatement, `sem_timedwait()` n'échoue jamais avec une valeur d'expiration de délai, quelque soit la valeur de `abs_timeout`. De plus, la validité de `abs_timeout` n'est pas vérifiée dans ce cas.

Sémaphores POSIX.4

```
int sem_getvalue(sem_t sem_id, int  
*value);
```

récupère la valeur du sémaphore dans value.

- **Retour :**
 - renvoie 0.
 - Si erreur : -1 + errno

```
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main(int argc, char * argv[])
{
    int    i;
    sem_t * sem;

    if (argc != 2) {
        fprintf(stderr, "usage: %s nom_semaphore\n",
            argv[0]);
        exit(EXIT_FAILURE);
    }
    sem = sem_open(argv[1], O_RDWR);
```

```
if (sem == SEM_FAILED) {  
    if (errno != ENOENT) {  
        perror(argv[1]);  
        exit(EXIT_FAILURE);  
    }  
    sem = sem_open(argv[1], O_RDWR | O_CREAT, 0666, 1);  
    if (sem == SEM_FAILED) {  
        perror(argv[1]);  
        exit(EXIT_FAILURE);  
    }  
    fprintf(stderr, "[%d] Creation de %s\n", getpid(), argv[1]);  
}
```

```
for (i = 0; i < 3; i ++) {  
    fprintf(stderr, "[%d] en attente...\n", getpid());  
    sem_wait(sem);  
    fprintf(stderr, "    [%d] tient le semaphore\n", getpid());  
    sleep(4);  
    fprintf(stderr, "    [%d] lache le semaphore\n", getpid());  
    sem_post(sem);  
    sleep(2);  
}  
return EXIT_SUCCESS;  
}
```

```
$ ./exemple-semaphore /azerty & ./exemple-semaphore  
/azerty  
[1] 8771  
[8771] Creation de /azerty  
[8771] en attente...  
    [8771] tient le sémaphore  
[8772] en attente...  
    [8771] lache le sémaphore  
    [8772] tient le sémaphore  
[8771] en attente...  
    [8772] lache le sémaphore  
    [8771] tient le sémaphore
```

Etc...

Administration par le shell

- Que faire si pas de `sem_unlink` ?
- Persiste jusqu'au reboot !
- **Solution** : système virtuel de fichiers `tmpfs`

```
$ mount  
...  
tmpfs on /dev/shm type tmpfs (rw)  
$ ls -l /dev/shm  
-rw-rw-r-- 1 lm lm 16 23 octobre 10:00 sem.azerty
```

- `rm`, `cp`, `mv`, `cat` `read/write` fonctionnent.