

Modèle producteurs-consommateurs

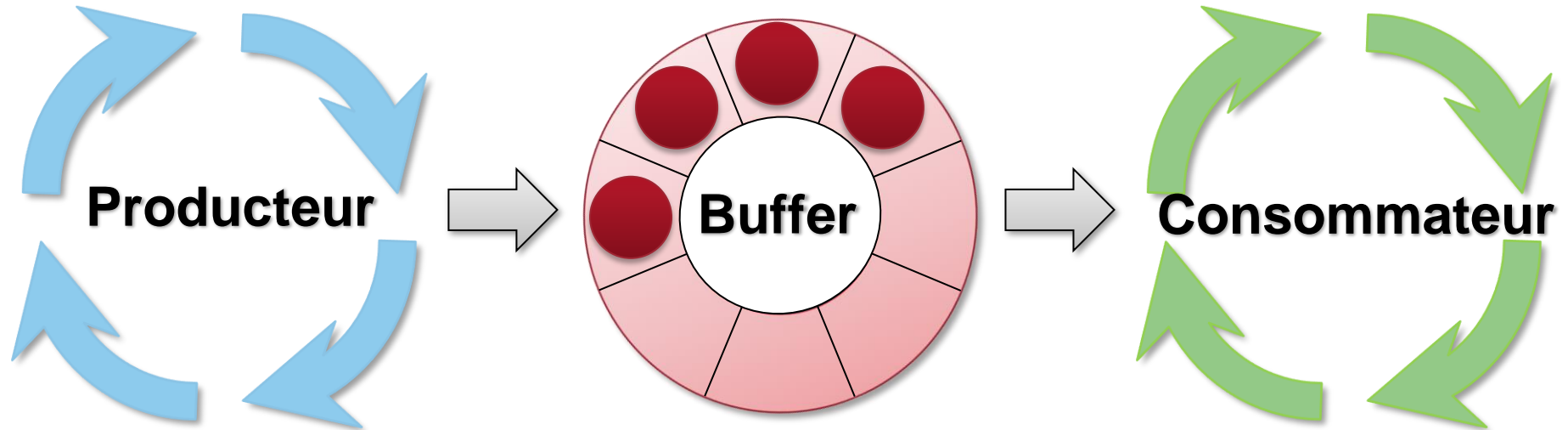
F. Gluck, V. Pilloux

Version 0.3

Introduction

- Le modèle producteurs-consommateurs (*producer-consumer*) est un modèle courant d'échange de données entre threads/processus.
- Des threads produisent des données et des threads les consomment :
 - Les producteurs créent des éléments qu'ils ajoutent à une structure de donnée (typiquement un buffer).
 - Les consommateurs les retirent pour les traiter.
- Modèle utilisé dans de nombreuses situations :
 - Communications (CPU ↔ périphérique, etc.).
 - Systèmes d'exploitation.
 - Toute FIFO accédée de manière concurrente (file d'impression, etc.).

Visuellement



- Le buffer est circulaire et se comporte comme une FIFO.
 - ⇒ les éléments sont consommés dans leur ordre d'insertion.
- La gestion du buffer nécessite deux pointeurs:
 - pointeur d'insertion;
 - pointeur de retrait.

Contraintes

- Les éléments du buffer ne sont consommés qu'une fois.
- Les éléments sont consommés selon leur ordre de production (FIFO).
- Lorsqu'un élément est ajouté ou enlevé du buffer, cela doit être fait de façon exclusive.
- Si un consommateur accède au buffer alors qu'il est vide, le consommateur doit être bloqué jusqu'à production d'un nouvel élément.
- Par soucis de simplicité, nous considérons ici un buffer de taille illimitée.

Modèle

- Chaque producteur effectue continuellement les opérations :

```
item = create_item()  
buffer_put(item)
```

- Chaque consommateur effectue continuellement les opérations :

```
item = buffer_get()  
process(item)
```

- Remarques :
 - L'accès à `buffer` doit être exclusif;
 - `create_item` et `process` sont des opérations considérées atomiques.

Buffer non borné

```
mutex = mutex()  
used = sem_init(0)
```

- `mutex` protège le buffer
- `used` sémaphore comptant le nombre d'éléments dans le buffer

Buffer non borné

```
mutex = mutex()  
used = sem_init(0)
```

- `mutex` protège le buffer
- `used` sémaphore comptant le nombre d'éléments dans le buffer

- Thread producteur :

```
item = create_item()  
  
buffer_put(item)
```

Buffer non borné

```
mutex = mutex()  
used = sem_init(0)
```

- `mutex` protège le buffer
- `used` sémaphore comptant le nombre d'éléments dans le buffer

- Thread producteur :

```
item = create_item()  
lock(mutex)  
buffer_put(item)  
unlock(mutex)  
post(used)
```


Buffer non borné

```
mutex = mutex()  
used = sem_init(0)
```

- `mutex` protège le buffer
- `used` sémaphore comptant le nombre d'éléments dans le buffer

- Thread producteur :

```
item = create_item()  
lock(mutex)  
buffer_put(item)  
unlock(mutex)  
post(used)
```

- Thread consommateur :

```
item = buffer_get()  
  
process(item)
```

Buffer non borné

```
mutex = mutex()  
used = sem_init(0)
```

- `mutex` protège le buffer
- `used` sémaphore comptant le nombre d'éléments dans le buffer

- Thread producteur :

```
item = create_item()  
lock(mutex)  
buffer_put(item)  
unlock(mutex)  
post(used)
```

- Thread consommateur :

```
wait(used)  
lock(mutex)  
item = buffer_get()  
unlock(mutex)  
process(item)
```

Buffer borné

- En situation réelle, un buffer sera toujours borné par les ressources systèmes à disposition (mémoire disponible, etc.).
- Le système d'exploitation utilise en général des buffers de taille fixe; par exemple : requêtes d'entrées/sorties, paquets réseau, commandes entre CPU et GPU, etc.
- Un buffer borné implique donc une contrainte supplémentaire :
 - **Si le buffer est plein, un producteur bloque jusqu'à ce qu'un consommateur retire un élément du buffer.**
- Nous considérons ici un buffer borné de taille N; il s'agit d'un buffer circulaire où l'index est calculé selon :

$$\text{index} = (\text{index} + 1) \% N$$

Buffer borné

- En situation réelle, un buffer sera toujours borné par les ressources systèmes à disposition (mémoire disponible, etc.).
- Le système d'exploitation utilise en général des buffers de taille fixe; par exemple : requêtes d'entrées/sorties, paquets réseau, commandes
- Un buffer borné est un buffer élémentaire :
 - Si le buffer est plein, on ne peut plus ajouter d'éléments jusqu'à ce qu'un consommateur retire un élément du buffer.
- Nous considérons ici un buffer borné de taille N; il s'agit d'un buffer circulaire où l'index est calculé selon :

$$\text{index} = (\text{index} + 1) \% N$$

Buffer borné

```
mutex = mutex()  
used = sem_init(?)  
free = sem_init(?)
```

- `mutex` protège l'accès au buffer
- `used` compte le nombre d'éléments déposés dans le buffer
- `free` compte le nombre d'emplacements disponibles dans le buffer

Buffer borné

```
mutex = mutex()  
used = sem_init(0)  
free = sem_init(N)
```

- `mutex` protège l'accès au buffer
- `used` compte le nombre d'éléments déposés dans le buffer
- `free` compte le nombre d'emplacements disponibles dans le buffer

Buffer borné

```
mutex = mutex()  
used = sem_init(0)  
free = sem_init(N)
```

- `mutex` protège l'accès au buffer
- `used` compte le nombre d'éléments déposés dans le buffer
- `free` compte le nombre d'emplacements disponibles dans le buffer

- Thread producteur :

```
item = create_item()  
  
buffer_put(item)
```

Buffer borné

```
mutex = mutex()  
used = sem_init(0)  
free = sem_init(N)
```

- `mutex` protège l'accès au buffer
- `used` compte le nombre d'éléments déposés dans le buffer
- `free` compte le nombre d'emplacements disponibles dans le buffer

- Thread producteur :

```
item = create_item()  
wait(free)  
lock(mutex)  
buffer_put(item)  
unlock(mutex)  
post(used)
```


Buffer borné

```
mutex = mutex()  
used = sem_init(0)  
free = sem_init(N)
```

- `mutex` protège l'accès au buffer
- `used` compte le nombre d'éléments déposés dans le buffer
- `free` compte le nombre d'emplacements disponibles dans le buffer

- Thread producteur :

```
item = create_item()  
wait(free)  
lock(mutex)  
buffer_put(item)  
unlock(mutex)  
post(used)
```

- Thread consommateur :

```
item = buffer_get()  
  
process(item)
```

Buffer borné

```
mutex = mutex()  
used = sem_init(0)  
free = sem_init(N)
```

- `mutex` protège l'accès au buffer
- `used` compte le nombre d'éléments déposés dans le buffer
- `free` compte le nombre d'emplacements disponibles dans le buffer

- Thread producteur :

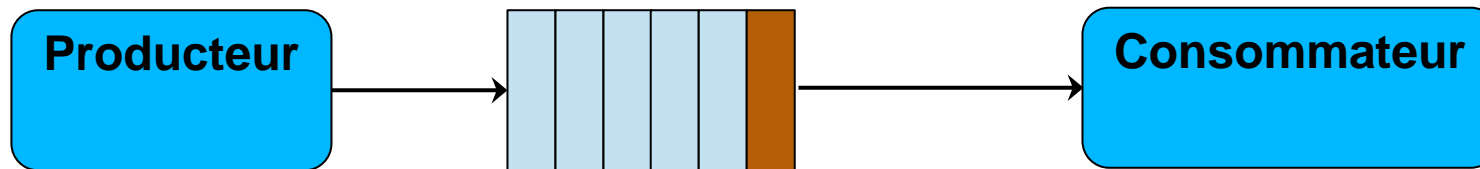
```
item = create_item()  
wait(free)  
lock(mutex)  
buffer_put(item)  
unlock(mutex)  
post(used)
```

- Thread consommateur :

```
wait(used)  
lock(mutex)  
item = buffer_get()  
unlock(mutex)  
post(free)  
process(item)
```

Implémentation existante: boîte au lettre / queues

- Les queues permettent de communiquer une information d'une tâche à une autre **sans la perdre**, car les informations à transmettre sont placées dans une FIFO qui est lue lorsque la tâche receveuse est prête. La taille de la FIFO est configurable.



- Sous Linux, les queues permettent aussi de communiquer entre **processus**.
- Exemple d'initialisation:

```
mqd_t message;
struct mq_attr attr;           // queue attributes

attr.mq_maxmsg = 10;
attr.mq_msgsize = 1000;
attr.mq_flags = attr.mq_curmsgs = 0;

message = mq_open ("/my_message", O_RDWR | O_CREAT,
                  0664, &attr)
```

Note: la création de la queue est visible sous /dev/mqueue/<mq_open name>

Boîte au lettre / Queues



- La priorité indique où le nouvel élément doit être inséré dans la FIFO:
nombre élevé = devant

Emetteur

```
unsigned int prio=1;  
char str="message de test";  
  
mq_send(message, str,  
          strlen(str), prio);
```

Bloquant si la FIFO est pleine

Récepteur

```
unsigned int prio;  
char str[1000];  
  
mq_receive(message, str,  
            1000, &prio)
```

Bloquant si la FIFO est vide

Détruit la queue ouverte avec mq_open():

```
mq_close(message);
```