

Séance 4

Synchronisation et communication interprocessus



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Rappels

- Multiprogrammation et **ordonnancement de processus**
 - Cycles de salves CPU et E/S
 - Fonctionnement de l'ordonnanceur et dispatcher
- Algorithmes d'**ordonnancement de processus**
 - Critères d'ordonnancement et règles d'optimisation
 - FCFS, SJF, SRTF, priorité, RR
 - File multi-niveaux (avec rétroaction)
 - Situations d'OS courants : Solaris, Windows, Linux

Objectifs

- Décrire et comprendre la **synchronisation de processus**
 - Problème de la section critique
 - Solution software : Algorithme de Peterson
 - Solution hardware : Test&Set, Compare&Swap
 - Solution apportée par l'OS : Mutex lock et sémaphore
- Principes de la **communication interprocessus**
 - Modèles de communication
 - Partage de données et synchronisation, gestion des signaux
 - Mécanismes OS : mémoire partagée et passage de messages

• SECT ION. •
4 - - 3 - - M

Section critique

Partage de données

- Deux processus peuvent **partager des données**

Par exemple avec une zone de mémoire partagée, un fichier...

- Un processus peut être **retiré du CPU** à tout moment

Exécution concurrente de processus coopératifs

- Des **conditions de course** peuvent se produire

Lorsque plusieurs processus manipulent des données partagées

Condition de course

- Un producteur **ajoute un élément** s'il reste de la place

```
while (true)
{
    while (counter == BUFFER_SIZE){}
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

- Un consommateur **retire un élément** s'il y en a

```
while (true)
{
    while (counter == 0){}
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Traduction en langage machine

- La variable `counter` est **partagée** entre les processus

Problème possible lors d'une exécution concurrente

- Supposons que la valeur de **counter soit 5**

counter modifiée de manière concurrente par les deux processus

■ `counter++`

$$\begin{aligned} R_1 &:= \text{counter} \\ R_1 &:= R_1 + 1 \\ \text{counter} &:= R_1 \end{aligned}$$

■ `counter--`

$$\begin{aligned} R_2 &:= \text{counter} \\ R_2 &:= R_2 - 1 \\ \text{counter} &:= R_2 \end{aligned}$$

Entrelacement des instructions

- Équivalence entre exécution concurrente et **entrelacement**

Exécution séquentielle des instructions de bas niveau

T_0	<i>producteur</i>	$R_1 := \text{counter}$	$\{R_1 = 5\}$
T_1	<i>producteur</i>	$R_1 := R_1 + 1$	$\{R_1 = 6\}$
T_2	<i>consommateur</i>	$R_2 := \text{counter}$	$\{R_2 = 5\}$
T_3	<i>consommateur</i>	$R_2 := R_2 - 1$	$\{R_2 = 4\}$
T_4	<i>producteur</i>	$\text{counter} := R_1$	$\{\text{counter} = 6\}$
T_5	<i>consommateur</i>	$\text{counter} := R_2$	$\{\text{counter} = 4\}$

- Il faut **interdire la modification** concurrente de counter

La solution est de synchroniser les processus

Opération atomique

- Une opération atomique **ne peut être interrompue**
Elle doit être exécutée dans son entièreté
- Il faut pouvoir signaler qu'une opération doit être **atomique**
Atomicité de counter++ et counter-- résout le problème

Problème de la section critique

- Soit un système avec n processus **en compétition** P_1, \dots, P_n
 - Chaque processus possède une section critique
 - Accès aux ressources partagées dans cette section critique
- **Exclusion mutuelle** des processus pour leur section critique
 - Jamais deux processus en même temps en section critique
 - Un processus doit demander l'autorisation d'y accéder

Structure des processus

■ Section d'entrée

Demande pour pouvoir entrer en section critique

■ Section de sortie

Fin de l'utilisation des ressources partagées

```
do
{
    section d'entrée
        // section critique
    section de sortie
        // section restante
}
while (true);
```

Conditions d'une solution

- **Conditions** d'une solution au problème de la section critique

1 Exclusion mutuelle

Si un processus est dans sa section critique, aucun autre processus ne peut être dans la sienne

2 Progression

Si aucun processus n'est en section critique et certains veulent rentrer dans la leur, la sélection de qui va pouvoir y rentrer ne peut être postposée indéfiniment et est prise par les processus pas en section restante

3 Attente limitée

Entre la demande pour rentrer dans sa section critique et l'autorisation, seul un nombre limité d'autres processus pourront rentrer dans la leur

Kernel préemptif

- Plusieurs processus en mode kernel actifs en même temps

Possibilité de conditions de course au sein du code du kernel

- Structure de données kernel pour liste des fichiers ouverts

Condition de course lors de modification concurrente

- Autoriser un processus en mode kernel à être préempté ou non

■ Kernel non préemptif est libre de condition de course

■ Difficulté avec kernel préemptif, surtout pour SMP

Meilleurs pour responsivité et pour temps réel



Synchronisation

Solution 1 : Attente active (1)

- Une **variable partagée** initialisée à zéro indique le tour

```
int turn = 0;

do
{
    while (turn != i){}

        // section critique

    turn = j;

        // section restante
}
while (true);
```

Solution 1 : Attente active (2)

- Le processus est en **attente active**
 - Les processus passent leur temps à vérifier la valeur de `turn`
 - Les processus ne peuvent rien faire de productif en attendant
- **Inconvénients**
 - Les processus doivent s'alterner de manière stricte
 - Un processus qui plante bloque le suivant
 - Ne satisfait pas à la condition de progression

Solution 2

- Deux **variables partagées** initialisées à false

```
bool flag[2]; flag[0] = flag[1] = false;
```

```
do
{
    while (flag[j]) {}
    flag[i] = true;

        // section critique

    flag[i] = false;

        // section restante
}
while (true);
```

- Ne satisfait pas à la condition d'exclusion mutuelle

Solution 3

- On indique a priori l'**intention de rentrer** en section critique
- Risque d'**interblocage** avec les deux processus qui attendent

Condition de progression pas satisfaite

```
do
{
    flag[i] = true;
    while (flag[j]) {}

    // section critique

    flag[i] = false;
    // section restante
}
while (true);
```

Solution 4

- Risque de « courtoisie mutuelle » entre les processus

Comme quand on croise quelqu'un dans la rue

```
do
{
    flag[i] = true;
    while (flag[j])
    {
        flag[i] = false;
        // attente
        flag[i] = true;
    }

    // section critique

    flag[i] = false;

    // section restante
}
while (true);
```

Algorithm de Peterson (1)

- Deux processus alternent l'exécution dans leur section critique
 - int turn indique à qui c'est le tour d'entrer
 - boolean flag[2] indique si le processus est prêt à rentrer
- Hypothèse que les différentes instructions sont atomiques

```
do
{
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j) {}

        // section critique

    flag[i] = false;

        // section restante
}
while (true);
```

Algorithm de Peterson (2)

- P_i rentre en section critique ssi $\text{flag}[j] == \text{false}$ OU $\text{turn} == i$
 - Si les deux en section critique $\text{flag}[i] == \text{flag}[j] == \text{true}$
 - Ne peuvent donc pas être en même temps en section critique
- P_i empêché d'entrer en section critique s'il reste dans sa while
 - Si P_j pas prêt, $\text{flag}[j] == \text{false}$ et P_i peut rentrer
 - Si P_j est en boucle et a mis $\text{flag}[j] == \text{true}$
 - Soit $\text{turn} == i$ et P_i rentre
 - Sinon c'est P_j qui rentre
 - P_i rentre (progression) après max une entrée P_j (attente limitée)

Désactivation des interruptions

- Système avec **un seul processeur** et exécution concurrente
 - Le processeur est limité dans sa capacité d'entrelacement
 - Exécution continue jusqu'à appel système ou interruption
- Désactivation hardware des **interruptions**

Problème avec le multiprocessing, garantit pas exclusion mutuelle

```
do
{
    /* désactiver interruptions */

    // section critique

    /* réactiver interruptions */

    // section restante
}
while (true);
```

Test&Set (1)

- Lire et modifier une valeur de manière atomique

*Renvoie la zone mémoire *target et la passe à true*

- test_and_set exécutée de manière atomique

Simule une instruction hardware disponible

```
bool test_and_set (boolean *target)
{
    bool rv = *target;
    *target = true;

    return rv;
}
```

Test&Set (2)

- Protection des sections critiques à l'aide de **verrous**

Un verrou est placé lors de l'entrée en section critique

- Une **variable partagée** initialisée à false

```
bool lock = false;
```

```
do
{
    while (test_and_set (&lock)) {}

        // section critique

    lock = false;

        // section restante
}
while (true);
```

Compare&Swap (1)

- Deux valeurs comparées et échangées de manière atomique

*Compare la zone mémoire `*val` avec la valeur `newval`*

- `compare_and_swap` exécutée de manière atomique

Simule une instruction hardware disponible

```
int compare_and_swap (int *val, int exp, int newval)
{
    int temp = *val;
    if (*val == exp)
    {
        *val = newval;
    }

    return temp;
}
```

Compare&Swap (2)

- Protection des sections critiques à l'aide de **verrous**

Un verrou est placé lors de l'entrée en section critique

- Une **variable partagée** initialisée à 0

```
int lock = 0;
```

```
do
{
    while (compare_and_swap (&lock, 0, 1) != 0){}

        // section critique

    lock = 0;

        // section restante
}
while (true);
```

Instruction machine spéciale

- Plusieurs **avantages** à une instruction machine spéciale
 - Fonctionne avec plusieurs processeurs partageant mémoire
 - Simple et très facile à vérifier
 - Support de plusieurs sections critiques avec autant de verrous
- Une série de **désavantages majeurs**
 - Utilisation d'attente active pour avoir le lock
 - Possibilité de famine avec un processus jamais autorisé
 - Deadlock possible si processus plus prioritaire en attente lock

Mutex lock (1)

- L'OS propose d'utiliser des **mutex Lock** (MUTual EXclusion)
Le programmeur n'a pas accès aux instructions hardware spéciales
- **Deux opérations** possibles par appel système
 - **Acquisition** du verrou avec acquire
 - **Libération** du verrou avec release
- Acquisition d'un mutex pour pouvoir entrer en **section critique**
Attente active sur acquire pour avoir le lock

Mutex lock (2)

- Définition des deux appels systèmes d'acquisition et libération

Exécution atomique avec attente active

```
acquire()
{
    while (! available){}
    available = false;
}
```

```
release()
{
    available = true;
}
```

- On parle de **spinlock** car le processus « *spin* » en attendant
 - Utile lorsque les temps d'attente sont courts
 - Un thread « *spin* », l'autre en section critique (multiprocesseur)

Sémaphore (1)

- Le **sémaphore** est un outil plus robuste que le mutex lock

Permet des moyens de synchronisation plus sophistiqués

- Variable entière utilisée pour la **signalisation**

Accédée uniquement via deux opérations, une fois initialisée

- Deux opérations **atomiques** possibles

- Se mettre en attente d'un signal avec `wait`
- Produire un signal avec `signal`

Sémaphore (2)

- Définition des deux appels systèmes d'acquisition et libération

Exécution atomique avec attente active

```
wait (S)
{
    while (S <= 0){}
    S--;
}
```

```
signal (S)
{
    S++;
}
```

- Un mutex n'est rien d'autre qu'un **sémaphore binaire**
Sa valeur varie entre 0 et 1
- Accès à une ressource disponible en un **nombre fini d'instances**

Appel wait pour prendre une instance et signal pour la libérer

Synchronisation de processus

- Utilisation d'un sémaphore pour **synchroniser deux processus**

Par exemple pour forcer l'ordre d'une exécution séquentielle

- Instructions S_2 de P_2 **d'office après** instructions S_1 de P_1
 - Feu rouge au départ attendu par P_2 , passé au vert par P_1
 - Sémaphore initialisé à la valeur 0

```
 $S_1;$ 
```

```
 $signal (synch);$ 
```

```
 $wait (synch);$ 
```

```
 $S_2;$ 
```

Élimination de l'attente active

- Après un wait, le processus doit être **bloqué**

Pour résoudre l'attente active dont souffre le mutex lock

- File d'attente** associée à chaque sémaphore (mis dans le PCB)
 - block ajoute processus à la file et le passe en *WAITING*
 - wakeup passe le processus en *READY*

```
wait (semaphore *S)
{
    S->value--;
    if (S->value < 0)
    {
        add (S->list, P);
        block();
    }
}
```

```
signal (semaphore *S)
{
    S->value++;
    if (S->value <= 0)
    {
        P = remove (S->list);
        wakeup (P);
    }
}
```

Interblocage

- **Interblocage** de processus en l'attente d'un autre

Chaque processus attend un évènement que l'autre doit produire

- Problème dû à compétition pour un **ensemble de ressources**

Et qu'elles sont à acquérir une à la fois

P_0

```
wait (S);  
wait (Q);  
  
...  
  
signal (S);  
signal (Q);
```

P_1

```
wait (Q);  
wait (S);  
  
...  
  
signal (Q);  
signal (S);
```

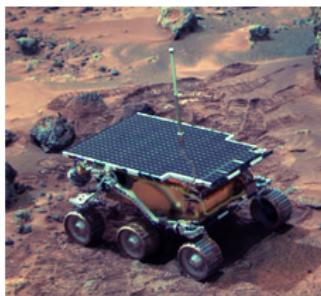
Inversion de priorité

- Inversion de priorité lors du partage d'une ressources
Une faible priorité peut maintenir lock face à une haute priorité
- Trois processus $L < M < H$ et ressource R détenue par L
 - H veut R et normalement, attend juste que L la libère
 - si M devient runnable, il préempte L et fait attendre H
- Résolution par héritage de priorité (temporaire)

Reçoit priorité haute du processus qui demande une ressource

Sojourner Rover

- Rover Sojourner déposé par Mars Pathfinder en 1997
Plusieurs resets software fréquents après le début de sa mission
- Tâche temps réel bc_dist trop de temps, reseté par bc_sched
 - Attente d'une ressource détenue par ASI/MET (priorité faible)
 - Configuration de l'héritage de priorité sur VxWorks à distance



Interaction de processus

■ Interaction entre processus par degré de conscience des autres

Trois principaux niveaux de conscience des autres processus

Conscience	Relation	Problème de contrôle potentiel
Pas de conscience	Compétition	<ul style="list-style-type: none">– Exclusion mutuelle– Deadlock (ressource renouvelable)– Famine
Conscience indirecte (partage de données)	Coopération par partage	<ul style="list-style-type: none">– Exclusion mutuelle– Deadlock (ressource renouvelable)– Famine– Cohérence de données
Conscience directe (communication)	Coopération par communication	<ul style="list-style-type: none">– Deadlock (ressource consommable)– Famine

Communication interprocessus



Types de processus

- Processus **indépendants**

Ne peuvent pas affecter ou être affecté par d'autres processus

- Processus **coopératifs**

Peut être affecté ou affecter d'autres processus

- Le **partage de données** définit le type de processus

Indépendant sans partage et coopératif avec

Processus coopératif

- Plusieurs raisons pour la **coopération de processus**
 - **Partage d'information** : accès concurrent au même fichier
 - **Accélération du calcul** : découpe en sous-tâches
 - **Modularité** : séparer les fonctions en processus/thread
 - **Commodité** : plusieurs tâches utilisateurs en même temps
- Besoin d'un moyen de **communication interprocessus**
InterProcessus Communication (IPC)

Architecture multiprocessus

- Google Chrome exécute un processus distinct par onglet
Plus robuste si un site crashe (JavaScript, Flash, HTML5...)
- Plusieurs types de processus identifiés
 - **Browser** : gère la GUI, le disque et les E/S
 - **Renderer** : gestion HTML, JS, image pour créer le rendu
 - **Plugin** : gère chaque plugin chargé (Flash, QuickTime...)
- Un **renderer** par onglet et un **plugin** par plugin chargé
Le renderer tourne dans une sandbox contre les exploits

Modèles de communication (1)

- Deux principaux **moyens de communication** entre processus

Tous les deux implémentés et offerts par la plupart des OS

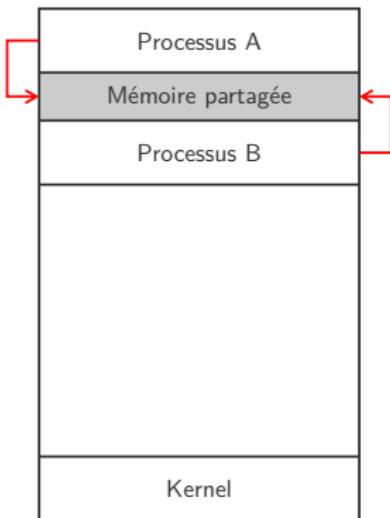
- Création d'une zone de **mémoire partagée**

Échange coordonné d'information par cette zone de mémoire

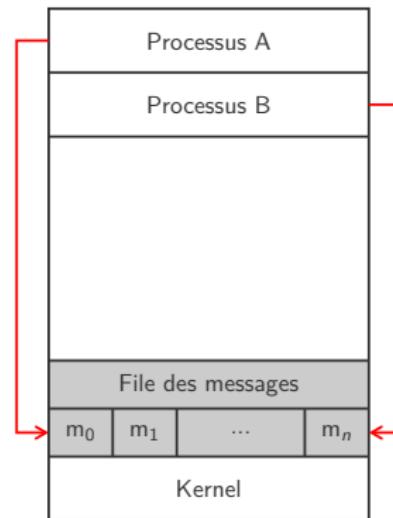
- Utilisation d'un **passage de messages**

Échange de messages entre les processus

Modèles de communication (2)



Mémoire partagée



Passage de messages

Mémoire partagée

- Très **rapide à exécuter** une fois la zone partagée établie

Aucune sollicitation de l'OS une fois la zone établie

- Protection à faire contre les **modifications concurrentes**

Utilisation, par exemple, de sémaphores

- Éventuel problèmes de **cohérence de cache**

En particulier sur les systèmes multiprocesseurs

Passage de messages

- Passage de messages utilisé pour des **petits messages**

Car il n'y a pas de conflits à éviter

- Plus **facile à implémenter** dans un système distribué

Primitives simples à utiliser

- Plus **lent à exécuter** et consommateur de temps

De nombreux appels systèmes à utiliser en permanence

Mécanisme d'IPC



Mémoire partagée

- Plusieurs étapes pour établir une communication
 - 1 Création/recherche d'une zone de mémoire partagée
 - 2 Attachement à la zone de mémoire partagée
 - 3 Lecture/écriture dans la zone de mémoire partagée
 - 4 Libération de la ressource
- Retraite de la protection mémoire pour les deux processus

L'OS autorise deux processus à accéder à la même zone mémoire
- Gestion du format et de la concurrence par les processus

L'OS n'intervient qu'à la création et à la libération

Problème du producteur–consommateur

- Un **buffer partagé** entre deux processus
 - Un processus *produit* des éléments dans un buffer
 - Un processus *consomme* les éléments dans le buffer
- Cas limites pour un **buffer limité**
 - On ne peut produire dans un *buffer plein*
 - On ne peut consommer dans un *buffer vide*
- On peut représenter un tel buffer par de la **mémoire partagée**

Tableau circulaire

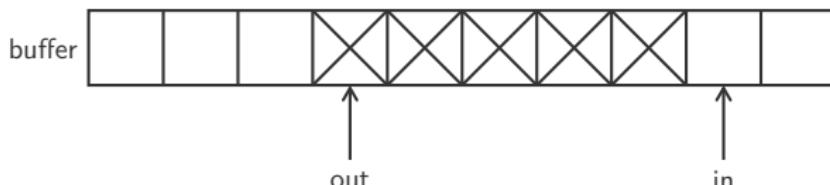
- Implémentation du buffer limité comme un **tableau circulaire**

On pourra stocker BUFFER_SIZE - 1 items dans le buffer

```
#define BUFFER_SIZE 10

typedef struct {
    // ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;           // prochaine case à remplir
int out = 0;          // prochaine case à vider
```



Producteur

- 1 Attendre que le buffer ne soit **pas plein**
- 2 **Ajouter** un élément dans le buffer
- 3 Avancer le **pointeur in** d'une position

```
while (true)
{
    // tant que le buffer est plein
    while (((in + 1) % BUFFER_SIZE) == out)
    {
        // on ne fait rien
    }

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consommateur

- 1 Attendre que le buffer ne soit **pas vide**
- 2 **Prendre** un élément dans le buffer
- 3 Avancer le **pointeur out** d'une position

```
while (true)
{
    // tant que le buffer est vide
    while (in == out)
    {
        // on ne fait rien
    }

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

Création (1)

- Crée ou localise une zone de mémoire partagée avec `shmget`

En fonction des paramètres qu'on donne à l'appel

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

int main()
{
    int shmid;
    char *shm;

    if ((shmid = shmget (5678, sizeof (int), IPC_CREAT | 0666)) < 0)
    {
        printf ("Impossible de créer la mémoire partagée.\n");
        exit (1);
    }

    // [...]

    return 0;
}
```

Création (2)

- Informations sur les IPC's avec la commande ipcs

La commande ipcs -a donne toutes les informations

- Chaque élément possède un **identifiant unique** (key)

On utilise cet identifiant pour faire des opérations sur un élément

```
>>> ipcs -m
IPC status from <running system> as of Sun Oct  9 22:14:00 CEST
2016
T      ID      KEY      MODE      OWNER      GROUP
Shared Memory:
m  65536  0x0000162e  --rw-rw-rw-  combefis    staff
```

Attachement

- **Attacher** la zone de mémoire partagée avec shmat

Placement de la mémoire dans l'espace d'adresses du processus

```
// [...]  
  
int main()  
{  
    // [...]  
  
    if ((shm = shmat (shmid, NULL, 0)) == (char*) -1)  
    {  
        printf ("Impossible d'attacher la mémoire partagée.\n");  
        exit (1);  
    }  
  
    // [...]  
  
    return 0;  
}
```

Écriture

- Utilisation de la zone de mémoire tout à fait normalement

Comme si on avait alloué la mémoire avec malloc

```
// [...]

int main()
{
    // [...]

    *shm = 'A';

    while (*shm != '*')
    {
        sleep(1);
    }
    printf ("Consumer has read me.\n");

    // [...]

    return 0;
}
```

Libération des ressources

- Demande de suppression de la zone mémoire avec shmctl

Sera supprimée lorsque plus personne n'y sera attaché

```
// [...]

int main()
{
    // [...]

    if ((shmctl (shmid, IPC_RMID , 0)) == -1)
    {
        printf ("Impossible de libérer la mémoire partagée.\n");
        exit (1);
    }

    return 0;
}
```

Recherche

■ Recherche d'une zone mémoire partagée avec `shmget`

Ne pas préciser le flag `IPC_CREAT`

```
// [...]

int main()
{
    // [...]

    if ((shmid = shmget (5678, sizeof (char), 0666)) < 0)
    {
        printf ("Impossible de trouver la mémoire partagée.\n");
        exit (1);
    }

    if ((shm = shmat (shmid, NULL, 0)) == (char*) -1)
    {
        printf ("Impossible d'attacher la mémoire partagée.\n");
        exit (1);
    }

    printf ("I read %c from the memory.\n", *shm);
    *shm = '*';

    return 0;
}
```

Producteur–consommateur simplifié

■ Côté producteur

```
./producer  
Consumer has read me.
```

■ Côté consommateur

```
./consumer  
I read A from the memory.
```

Passage de messages

- Communiquer par messages avec deux opérations principales
 - `send (msg)` : *envoie* un message
 - `receive (msg)` : *reçoit* un message
- Messages de tailles fixes ou variables

Le premier choix facilite l'implémentation du système
- Différentes caractéristiques du lien de communication
 - Communication directe ou indirecte
 - Communication synchrone ou asynchrone
 - Buffer automatique ou implicite

Communication directe

- Les processus communicants sont **nommés**
 - `send (P, msg)` : envoie un message à P
 - `receive (Q, msg)` : reçoit un message de Q
- Il y a **exactement un lien** entre toute paire de processus
 - Ceux qui ont manifesté l'envie de communiquer entre eux*
- On peut avoir une communication **asymétrique**
 - receive (id, message) ne doit pas spécifier un processus*
- Le code est **dépendant des identifiants** des processus
 - Souvent hard-codés, ce qui rend difficile l'évolution du code*

Communication indirecte (1)

- Les processus envoient/reçoivent des messages vers des **ports**

Les ports représentent des boîtes aux lettres, à partager

- Deux processus communiquant doivent **partager une boîte**

- `send (A, msg)` : envoie un message à la boîte A
- `receive (A, msg)` : reçoit un message de la boîte A

- Il y a **un lien** lorsqu'une boîte est partagée

Un lien peut être associé à plus de deux processus

Communication indirecte (2)

- La communication peut se faire entre **plusieurs processus**

Chaque message ne sera lu que par un processus au plus

- **Plusieurs liens** possibles entre la même paire de processus

Il suffit d'avoir plusieurs boîtes partagées

- L'OS propose des **opérations** pour gérer les boîtes

- Créer une nouvelle boîte
- Envoyer et recevoir des messages par la boîte
- Supprimer une boîte

Synchronisation

- Deux modes de communication : **synchrone** et **asynchrone**

Les appels send et receive peuvent être bloquants ou non

- **Quatre situations** différentes

	bloquant	non-bloquant
send	Envoi et attente de réception	Envoi et poursuite
receive	Blocage jusqu'à réception	Réception d'un message ou NULL

- On peut aussi bloquer pendant un **certain temps** spécifié

L'appel peut donc débloquer en renvoyant une valeur d'erreur

Buffer de messages

- File de messages implémentée avec des buffers

Que ce soit une communication directe ou indirecte

- Trois choix d'implémentation possibles

- Capacité nulle

L'émetteur doit attendre que le récepteur ait lu le message

- Buffer limité

n messages max, l'émetteur doit attendre si la file est pleine

- Buffer illimité

Longueur infinie, l'émetteur n'attend jamais

Création (1)

■ Créer ou localiser un file de messages msgget

En fonction des paramètres qu'on donne à l'appel

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

typedef struct msgbuf {
    long type;
    char data[2];
} msg;

int main()
{
    int msqid;
    msg msg;

    if ((msqid = msgget (5678, IPC_CREAT | 0666)) < 0)
    {
        printf ("Impossible de créer la file de messages.\n");
        exit (1);
    }

    // [...]
}
```

Création (2)

- Informations sur les IPC's avec la commande ipcs

La commande ipcs -a donne toutes les informations

- Chaque élément possède un **identifiant unique** (key)

On utilise cet identifiant pour faire des opérations sur un élément

```
>>> ipcs -q
IPC status from <running system> as of Sun Oct  9 23:43:51 CEST
2016
T      ID      KEY      MODE      OWNER      GROUP
Message Queues:
q  65536  0x0000162e  --rw-rw-rw-  combefis    staff
```

Envoi

- Envoi d'un message dans une file avec msgsnd

Spécification du type du message dans la structure

```
// [...]

int main()
{
    // [...]

    msg.type = 1;
    msg.data[0] = 'A';
    msg.data[1] = '\0';

    if (msgsnd (msqid, &msg, 2, IPC_NOWAIT) < 0)
    {
        printf ("Impossible d'envoyer le message.\n");
        exit (1);
    }

    printf ("Message bien envoyé.\n");

    return 0;
}
```

Réception

■ Réception d'un message dans une file avec msgrecv

Suppression de la file de message avec msgctl

```
// [...]

int main()
{
    // [...]

    if ((msqid = msgget (5678, 0666)) < 0)
    {
        printf ("Impossible de trouver la file de messages.\n");
        exit (1);
    }

    if (msgrecv (msqid, &msg, 2, 1, 0) < 0)
    {
        printf ("Impossible de recevoir le message.\n");
        exit (1);
    }

    printf ("I received %s in the message.\n", msg.data);

    // [...]

    return 0;
}
```

Envoi de message simplifié

■ Côté producteur

```
./producer  
Message bien envoyé.
```

■ Côté consommateur

```
./consumer  
I received A in the message.
```

Pipe ordinaire

- Deux processus peuvent communiquer par un **pipe**
 - Pipe **ordinaire** lorsque partage du même environnement
 - Pipe **nommé** entre deux processus quelconques
- Communication **unidirectionnelle**
- S'apparente à un **fichier** en mémoire



Exemple de pipe

```
int main()
{
    int p[2];
    char buffer[6];
    pipe (p);
    pid_t pid = fork();

    if (pid == 0) // Dans le fils
    {
        close (p[1]); // Fermeture en écriture

        read (p[0], buffer, 5);
        close (p[0]);

        buffer[5] = '\0';
        printf ("Le fils a reçu : %s\n", buffer);
    }
    else // Dans le parent
    {
        close (p[0]); // Fermeture en lecture

        write (p[1], "Hello", 5);
        close (p[1]);
    }

    return 0;
}
```

Crédits

- <https://www.flickr.com/photos/scoobyfoo/241651927>
- <https://www.flickr.com/photos/16210667@N02/14735983982>
- https://en.wikipedia.org/wiki/File:Sojourner_on_Mars_PIA01122.jpg
- <https://www.flickr.com/photos/30003006@N00/2439637326>
- https://www.flickr.com/photos/vlastimil_koutecky/9205853995