Cours 5

Tâches et flots

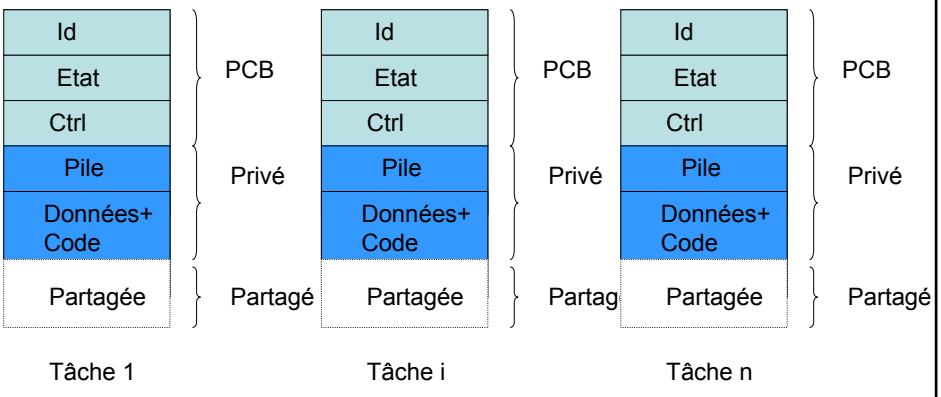
Niveau	Nom	Objets impliqués	Exemple d'opérations
13	Shell	Env utilisateur	Langage shell
12	Processus utilisateur	Processus	Fork, quit, kill, suspend, resume
11	Répertoire	Répertoire	Mkdir, remove, attach, detach
10	Périphériques	Imprimante, écran, clavier	Open, close, read, write
9	Système de Gestion de fichiers	Fichiers	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Mémoire virtuelle	Segments, pages	Read, Write, Fetch
6	Mémoire de masse	Blocs de données, canaux d'E/S	Read, Write, Allocate, free
5	Processus, tâches et flots	Tâches, sémaphores, liste de tâches	Suspend, Resume, Wait, Signal
4	Interruptions	Invocation pilote (handler)	Invoke, mask, unmask, retry
3	Procédures	Appel et retour, contexte	Call, return
2	Jeu d'instructions	Pile d'évaluation, gestion des UF, microprogrammes	Load, Store, add, addf, branch, jsr, rts, rti
1	Carte et circuits	Registres, UF, bus …	Clear, activate, transfert …

Les flots

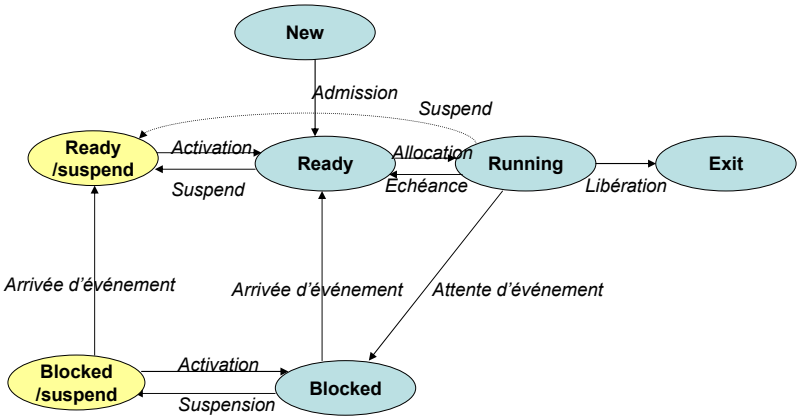
- Rappels
- Modèles multiflot
- Communication
- Exemples

Rappel : Tâches ou processus

- Tâches conventionnelles
  - Unité de gestion des ressources

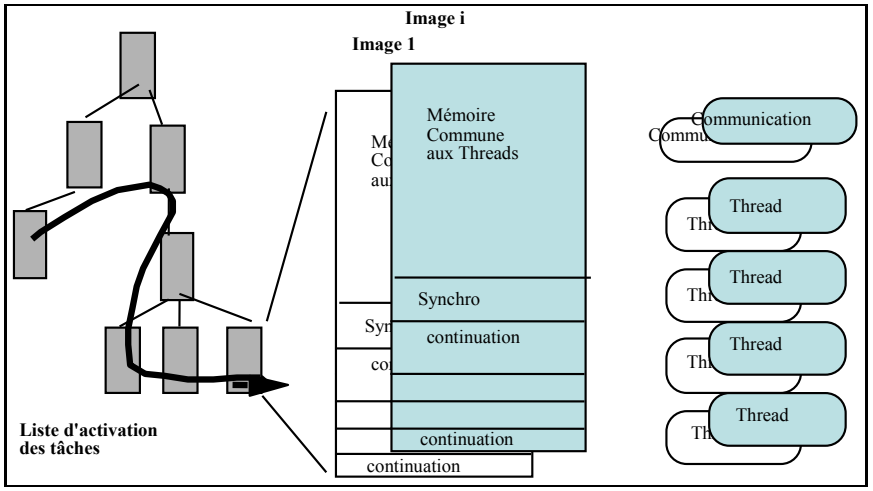


– Unité d'allocation
  - Ordonnancée et allouée par le SE
  - Multitâches

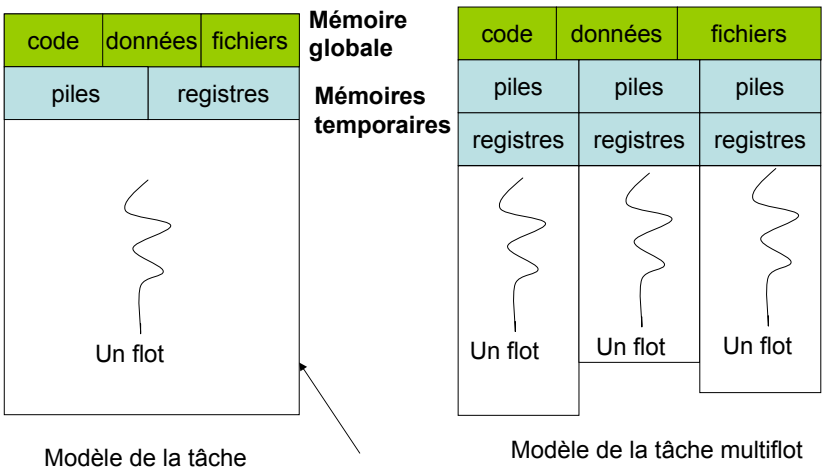


Flots

- Les deux fonctions précédentes sont distribuées dans les SE modernes
  – Tâches
    - Unité de gestion des ressources (allocation et protection)
    - Gestion mémoire virtuelle, E/S, fichiers, mémoire
  – Flots
    - Unité d'exécution dans un espace mémoire de la tâche
    - Ordonnancée et allouée par le SE

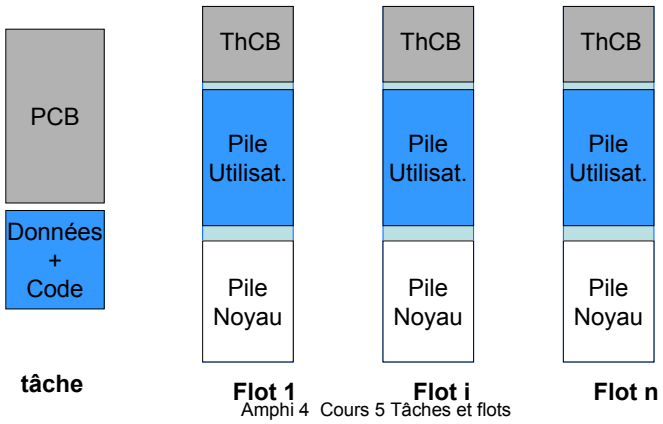


Modèles mémoire tâches et flots



Mécanisme multiflot

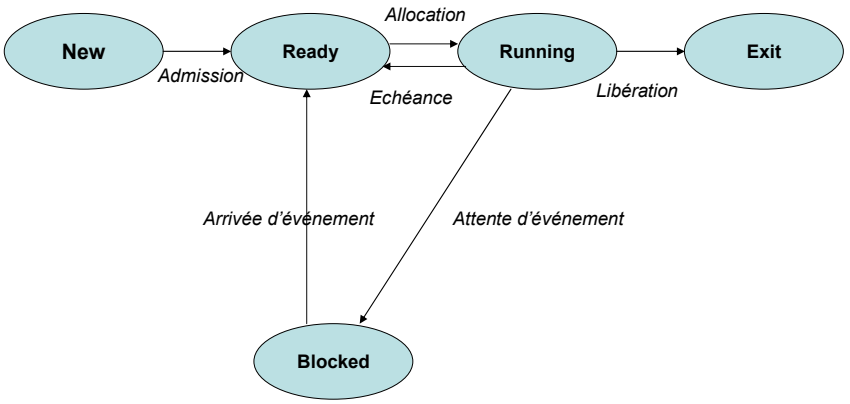
- Consiste pour un SE à supporter l'exécution de plusieurs flots à l'intérieur de l'exécution d'une seule tâche



Caractéristiques

- Performances
  - Création (rapport >10 entre flots et tâches)
  - Terminaison
  - Commutation (rapport >5 entre flots et tâches)
  - Communication (pas d'intervention du noyau)
  - Exploitation d'un multiprocesseur
- Etats
  - Prêt, en exécution, bloqué
- Opérations
  - Création, suspendu, activé, terminé

Etats d'un système multiflot



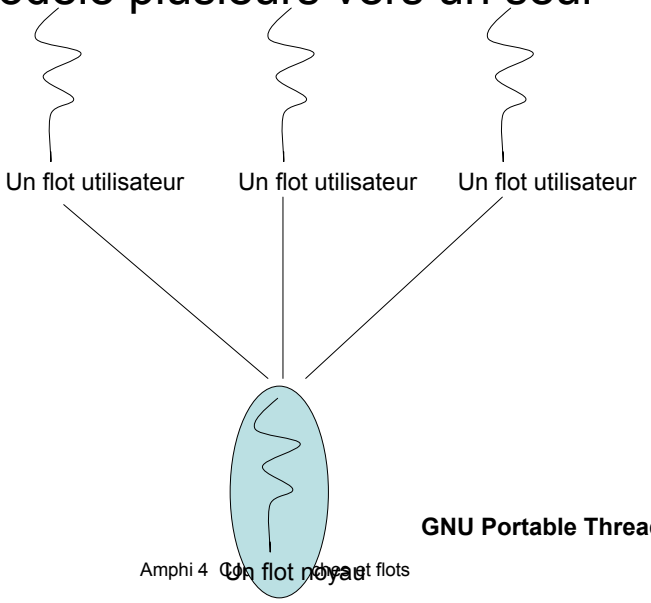
Implémentation des flots

- Niveau utilisateur
  - La gestion des ressources est reportée au niveau utilisateur : réalisé par une bibliothèque de fonctions pour la gestion des flots (création, destruction, communication, ordonnancement, sauvegarde et restauration de contexte de flots, synchronisation) ; le contrôle d'un thread à un autre est passé par un appel de fonction (appel de procédure et non appel système)

- Niveau noyau (kernel thread)
  - Toute la gestion est réalisée au niveau noyau ; définition d'une API pour interfacer application et noyau (Windows, Linux, Unix).
  - Passage par le noyau pour activer un flot

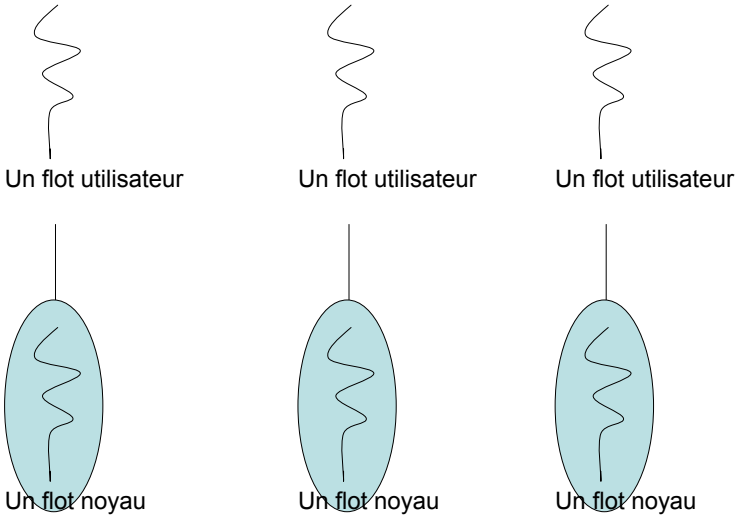
- Mode combiné
  - Création de flots en mode utilisateur ; un certain nombre sont transformés en flot noyau (à l'initiative du programmeur) en fonction de l'application. (Solaris)

Modèle plusieurs vers un seul



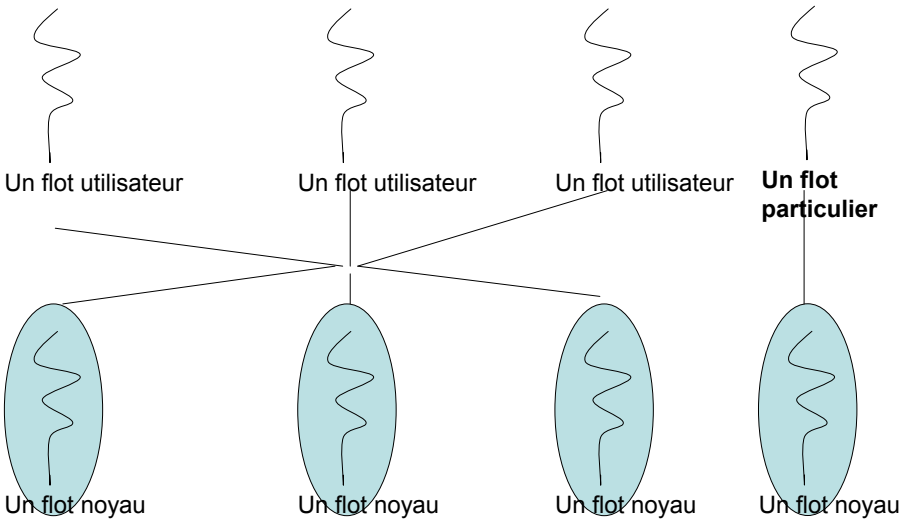
GNU Portable Threads

Modèle un vers un



Linux, Windows NT, 2000, XP

Modèle plusieurs vers plusieurs (modèle à 2 niveaux)



IRIX, HP-UX, Solaris

Comparaison des différents modèles

- Utilisateur (plusieurs vers un seul)
  - Avantages
    - Pas d'overhead de commutation
    - Ordonnancement à la discrétion de l'application
    - Indépendant du SE support
  - Inconvénients
    - Sur l'exécution d'un appel système, si le flot est bloqué, tous les flots sont bloqués
    - Impossible d'utiliser un multiprocesseur : une tâche par processeur ; donc un seul flot par tâche sur un processeur à un instant donné.

- Noyau (un vers un)
  - Avantages
    - Véritable parallélisme : le noyau peut ordonnancer de multiples flots de la même tâche sur plusieurs processeurs
    - Un flot bloqué provoque un changement de contexte pour exécuter un nouveau flot
    - Le noyau peut être lui-même multiflot
  - Inconvénients
    - Coût de commutation
    - Possibilité d'un excès de flots noyau difficile à gérer
- Mixte (plusieurs vers plusieurs)
  - Avantages
    - Maîtrise du nombre de flots concurrents
  - Inconvénients
    - Le noyau ne peut ordonnancer qu'un flot à la fois ; c'est un organe centralisateur.

Bibliothèque de flots

- Il existe aujourd'hui trois bibliothèques :
  - Les threads Posix (niveau utilisateur ou noyau)
  - Les threads Win32 (niveau noyau)
  - Les threads Java (implémentation de la JVM sur le système associé)

Exemple 1 : Posix

```
#include <pthread.h>
#include <stdio.h>

int somme ; /* données partagées par tous les flots */
int val_init = 10;

int main()
{
  pthread_t tid ;              /* l'identificateur du flot */
  pthread_attr_t attr ;        /* attributs du flot */

/* initialisation en positionnant les attributs par défaut */
ptgread_attr_init (&attr) ;
/* creation du flot */
pthead_create (&tid, &attr, calcul, &val_init) ;

/* attente que le flot précédent se termine */
pthread_join(tid,NULL) ;

printf ("somme = %d\n", somme) ;
}
```

```
/* le flot commence son exécution ici */

void *calcul (void *valeur)
{
 int i ;                /* variable locale à la fonction */

 somme = 0 ;  /* initialisation de la variable globale */
 for (i=1 ; i <= *valeur ; i++)
 somme  += i ;

 pthread_exit(0) ;
}
```


Exemple 2

- Deux flots partagent un tableau de N entiers
- Les flots doivent régulièrement incrémenter chaque entrée du tableau
- L'incrémentation doit se faire en N opérations

```
#include <stdio.h>
#include <pthread.h>
#define TAILLE_TABLEAU 10
pthread_t id1, id2 ;

typedef int valeur_tableau[TAILLE_TABLEAU] ;

valeur_tableau LeTableau ;

void LitTableau(Valeur_tableau T) {
    int i ;
    for (i=0 ; i < TAILLE_TABLEAU ; i++) T[i] = LeTableau[i];
}

void EcritTableau(Valeur_tableau T) {
    int i ;
    for (i=0 ; i < TAILLE_TABLEAU ; i++)  LeTableau[i] = T[i];
}
```



```
void f(void) {              /* le code des flots */
    int i, j ;
     valeur_tableau Tlocal ;

    for (j=0 ; j < 1000000 ; j++) {
            LitTableau (Tlocal) ;
            for (i=0 ; i < TAILLE_TABLEAU ; i++) {
                                Tlocal[i]++] ;
            }
            EcritTableau(Tlocal) ;
    }
}
```

```
int main () {
    int i ;
    for ( i= 0 ; i < TAILLE_TABLEAU ; i++)
        LeTableau[i] = 0 ;

    pthread_create(&id1, NULL, (void *(*)() f, NULL) ;
    pthread_create(&id2, NULL, (void *(*)() f, NULL) ;

    pthread_join (id1, NULL) ;
    pthread_join (id2, NULL) ;

    printf (‘‘valeur finale du tableau : \n ‘‘) ;

    for ( i= 0 ; i < TAILLE_TABLEAU ; i++)
        printf( ‘‘%d\n’’, LeTableau[i]) ;
}
```


Valeur finale du tableau :

1543280
1543280
1177867
1177867
1177867
1177867
1177867
1177867
1177867
1177867

On aurait du avoir 10 lignes avec la valeur 2000000

Il faut qu’une séquence lecture incrémentation écriture soit terminée avant qu’une autre puisse commencer.

```
void f(void) {
    int i, j ;
     valeur_tableau Tlocal ;
    pthread_mutex_ t mutex  ;

    pthread_mutex_init(&mutex, NULL);

    for (j=0 ; j < 1000000 ; j++) {
            pthread_mutex_lock(&mutex) ;
            LitTableau (Tlocal) ;
            for (i=0 ; i < TAILLE_TABLEAU ; i++) {
                                Tlocal[i]++] ;
            }
            EcritTableau(Tlocal) ;
            pthread_mutex_unlock(&mutex) ;
    }
}
```


Considérations sur les flots

- Status des primitives fork() et exec() en multiflot
  - Pour le fork() il existe deux versions (Unix) :
    - Duplication de tous les flots
    - Duplication du seul flot qui invoque le fork()
  - Pour le exec() le programme paramètre de l'exec() remplace la tâche et tous les flots.
- Abandon de l'exécution
  - Arrêt asynchrone (un flot arrête immédiatement un flot ciblé).
  - Arrêt par interrogation (un flot teste une variable condition pour savoir s'il doit continuer ou non) à des points identifiés (Pthreads).

Gestion des signaux

- Un signal notifie une tâche de l'arrivée d'un événement (terminologie Unix) ; il obéit au patron suivant :
  - Un signal est engendré par l'apparition d'un événement.
  - Le signal engendré est transmis à la tâche.
  - Une fois transmis, le signal est traité.
- Un signal peut être reçu en synchrone (envoyé à la tâche qui l'a engendré) ou en asynchrone (engendré par une autre tâche)
- Les signaux peuvent être traités « par défaut » ou par des tâches utilisateurs.

Gestion des signaux (suite)

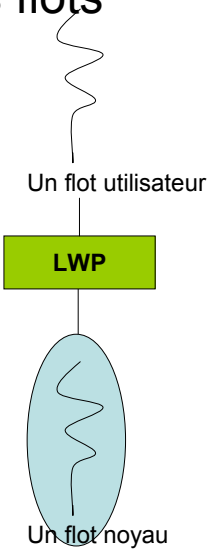
- Dans un système monotâche un signal est toujours envoyé à la tâche qui l'a engendré.

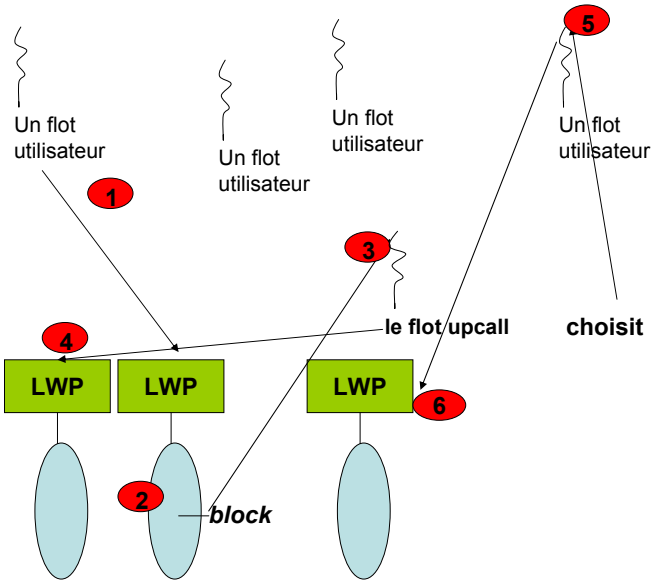
- Dans un système multiflot différentes options existent :
  - Envoyer le signal au flot qui l'a engendré (synchrone)
  - Envoyer le signal à chacun des flots de la tâche
  - Envoyer le signal à certains flots
  - Identifier un flot spécifique pour recevoir tous les signaux pour la tâche.

Ptheads Posix : pthread_kill (pthread_t tid, int signal) ;
signal est envoyé à tid

Ordonnancement des flots

- Les systèmes implémentant le modèle « plusieurs vers plusieurs » ou le modèle à deux niveaux utilisent une structure de données intermédiaire entre le noyau et l'utilisateur : le LWP (Lightweight process) qui apparaît comme un processeur virtuel sur lequel l'application peut ordonnancer les tâches utilisateur.
- Chaque LWP est attaché à un flot noyau et ce sont les flots noyau qui sont ordonnancés.





Ordonnancement des flots (suite)

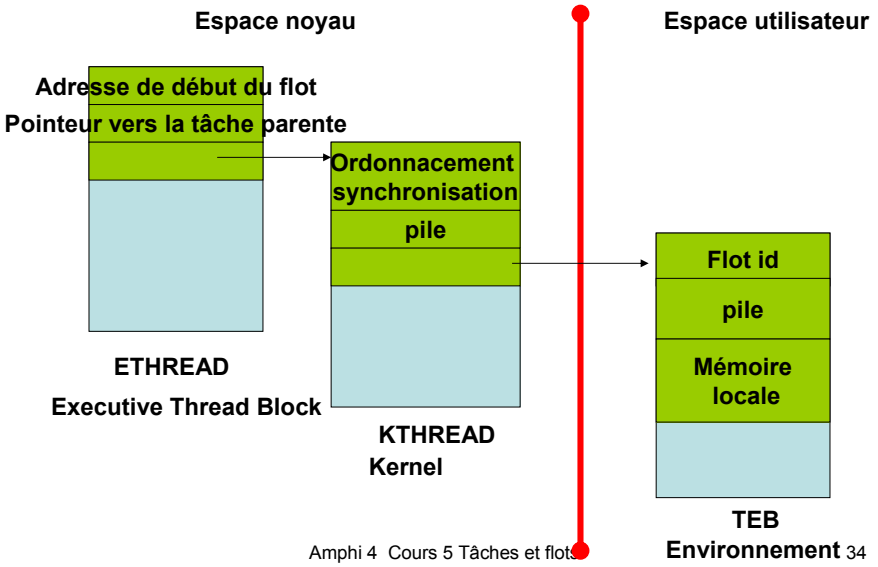
- Le fonctionnement est le suivant :

Le noyau fournit à l'application un ensemble déterminé de LWP, et l'application doit préempter un Processeur Virtuel (PV) pour activer un flot.

Certains événements déterminés par le noyau sont traités dans une procédure dédiée (upcall) qui nécessite un PV pour s'exécuter ; par exemple un flot qui se bloque en présence d'une latence importante signale, via un upcall la nécessité d'ordonnancer un nouveau flot.

Cet upcall préempte un PV pour déterminer le nouveau thread et affecte un LWP disponible si possible.

Exemple Windows XP



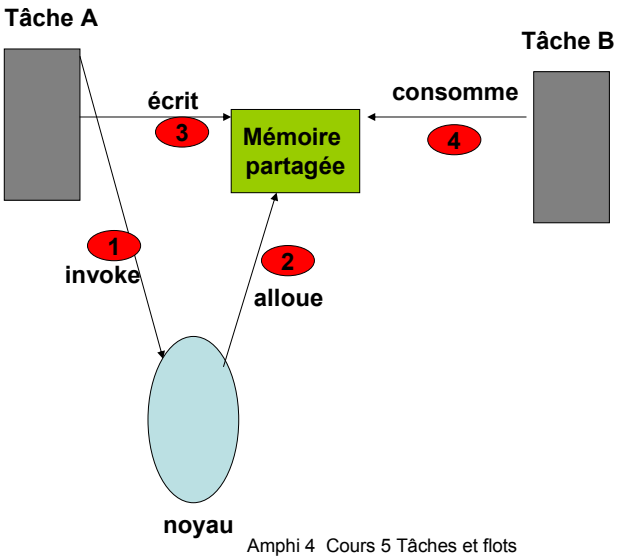
- XP utilise le modèle « un pour un » mais fournit la bibliothèque « fiber » pour le modèle « plusieurs vers plusieurs »
- Le contexte d'un thread est formé de :
  - Un Id
  - Un ensemble de registres pour les mémoires du proc.
  - Une pile utilisateur
  - Un espace mémoire local (dll, bibliothèque dédiée)

Communication entre tâches et flots

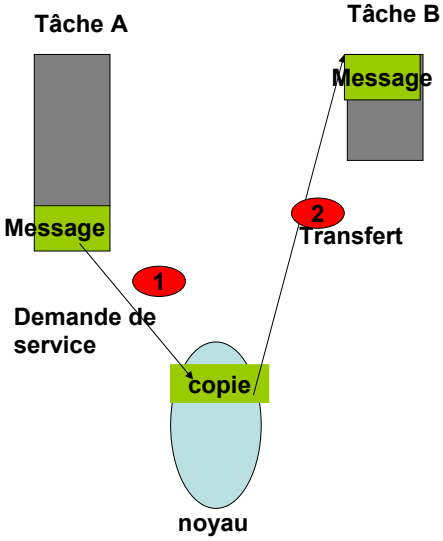
- Une tâche possède un espace d'adressage propre inaccessible par les autres.
- Un flot s'exécute dans une tâche et partage l'espace d'adressage de la tâche avec tous les autres flots.

- La coopération entre deux tâches nécessite un mécanisme de communication (IPC) :
  - Par partage de mémoire
  - Par passage de messages

Mémoire partagée



Passage de messages



Exemple Posix Shmem

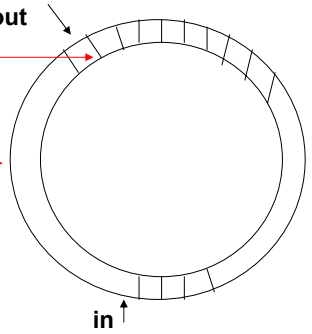
```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
 int main ()
{
 int segment_id ;              /* l'id du segment partagé */
 char * shared_memory ;  /* un ptr sur le segment de mem partagée */
 const int size = 4096 ; /* la taille du segment partagé */

    /* allocation d'un segment de mémoire partagée */
    segment_id = shmget (IPC_PRIVATE, size, S_IRUSR | S_IWUSR) ;

    /* attachement du segment */
    shared_memory = (char *) shmat (segment_id, NULL, 0) ;
```


Exemple Posix Shmem (suite)

```
/* écriture d'une donnée dans ce segment */
    sprintf (shared_memory, ''on partage cela à deux '') ;

/* impression du contenu du buffer partagé */
    printf (''*%s\n'', shared_memory ) ;

/* détachement du segment */
    shmdt (shared_memory) ;

/* libération mémoire */

 shmctl ( segment_id, IPC_RMID, NULL) ;

 return(0) ;
}
```

Exemple : un producteur consommateur

```
#define BUFFER_SIZE 10
typedef struct {
        ....
} item ;

item buffer [BUFFER_SIZE];
int in = 0 ;
int out = 0 ;
```



- in = out => buffer vide
- ((in+1) % BUFFER_SIZE) == out => buffer plein

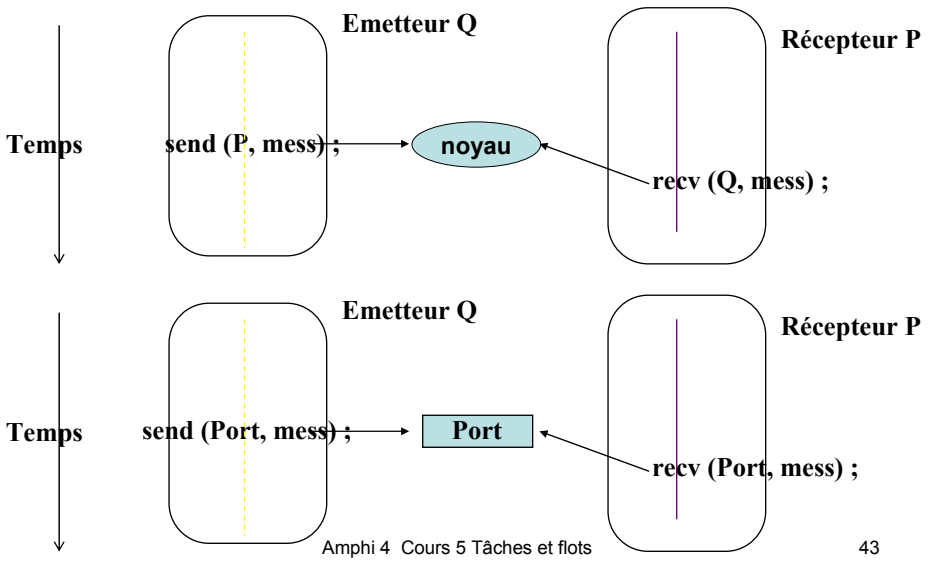
Le producteur

```
item nextProduced ;
while (true) {
    /* l'item à écrire est dans nextProduced */
    while (((in+1) % BUFFER_SIZE) == out) /* le buffer est plein  */
          ; /* rien à faire */
    buffer [in] = nextProduced ;
    in = (in+1) % BUFFER_SIZE ;
}
```

Le consommateur

```
item nextConsumed ;
while (true) {
          while (in == out) /* le buffer est vide  */
          ; /* rien à faire */
    nextConsumed = buffer [out] ;
    out = (out+1) % BUFFER_SIZE ;
/* l'item est transféré dans nextConsumed */
}
```

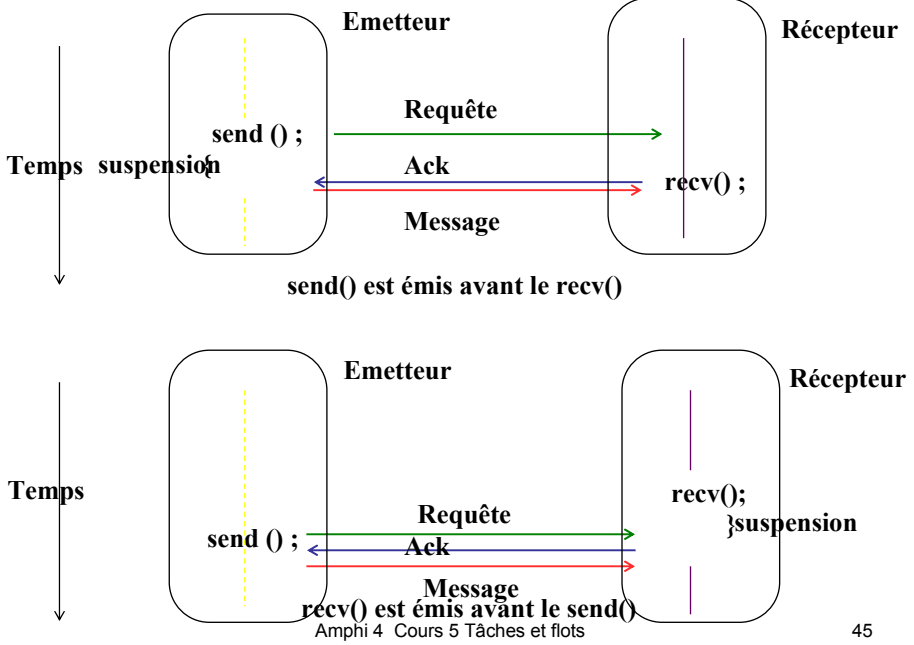
Le passage de messages : nommage



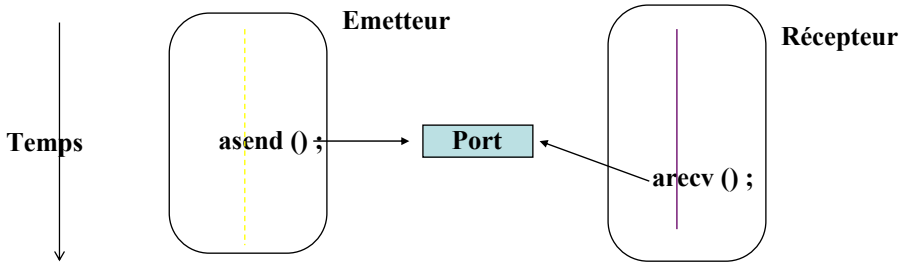
Synchronisation

- Communication synchrone bloquante
- Communication asynchrone non bloquante

send() et recv() synchrone utilisant un protocole 3 voies



Transfer asynchrone de messages



Annexe : exemple Linux

Historique

- Linus Thorvalds 1991
- Free Software Foundation
  - GNU public licence (GPL)
  - Décliné sur toutes les plates-formes

Linux

Architecture

Une architecture monolithique composée d'une collection de modules

- Blocs indépendants sont des modules chargeables
- Un module peut être lié et délié du noyau à l'exécution
- Un module implémente des fonctions comme :
  - Le gestionnaire de fichiers
  - Un pilote de périphérique
  - Une interface réseau
- Un module est exécuté en mode noyau pour le compte de la tâche en cours

Caractéristiques des modules chargeables :

- Dynamique
  - Seuls les modules nécessaires sont en mémoire à un instant donné. (insmod et rmmod) commandes
- Hiérarchie de modules
  - Bibliothèque de modules pour l'application et/ou
  - Clients de modules de services
  - Définition des dépendances entre modules
- Modules
  - Chaque module est défini par deux tables :
    - La table des modules
      - Liste chaînée de table de modules
      - Contient toutes les informations nécessaires pour gérer un module
    - La table des symboles
      - Symboles contrôlés par ce module et utilisés ailleurs

Composants du noyau

- Unité d'exécution
  - Pas de distinction entre tâche et flots
- Architecture du noyau
  - Collection de composants dont les principaux sont :
    - Signaux
    - Appels système : exemples
    - Ordonnanceur de tâches
    - Gestionnaire de mémoire virtuelle
    - Gestionnaire de fichiers
    - L'interface Sockets
    - Gestionnaire de périphérique caractère
    - Gestionnaire de périphérique mode bloc
    - Gestionnaire cartes réseau
    - Récupération d'erreurs engendrés par l'UC
    - Gestion d'interruptions

Gestion des tâches Linux

- Les tâches Linux
  - Une tâche est représentée par une structure *task_struct* contenant :
    - Les informations d'état
    - Les informations d'ordonnancement
      - Normal, temps-réel, priorités
    - Des identifieurs
      - Individu, groupe
    - Les IPC
      - L'identification des sockets
    - Des liens
      - Parents enfants et autres
    - Gestion du temps
      - Timer et temps utilisé
    - Table des fichiers ouverts
    - Table définissant la mémoire virtuelle
    - Le contexte processeur

- Les flots Linux
  - Les flots utilisateurs sont mappés dans des flots noyau
  - Un ensemble de flots utilisateur qui constitue une seule tâche sont portés au niveau noyau en partageant le même ID.
  - Une tâche est créée en copiant les attributs de la tâche mère : définit un clone. Lorsqu'ils partagent la même mémoire ils deviennent des flots d'une même tâche.
  - Clone() identique à la commande fork() d'Unix
  - Sur une commutation de CTX le noyau teste si le répertoire de MV est le même : si oui, pas de commutation et un branchement vers le code du flot en créant une nouvelle pile d'exécution.