Systèmes temps-réel

Matthieu Herrb

https://homepages.laas.fr/matthieu/cours/temps-reel/

Janvier 2018

Plan général

Bibliographie

- A. Tannenbaum, *Les systèmes d'exploitation*, 3e édition, 2008, Pearson.
- A. Silbershatz, P.B. Galvin and G. Gagne, *Operating Systems Concepts*, 9e édition, Wiley, 2016.
- D. Butenhof, *Programming with POSIX Threads*, 1997, Addison Wesley.
- B. Gallmeister, *Posix.4 Programmers Guide : Programming for the Real World*, 1995, O'Reilly.
- D. Abbott, *Linux for embedded and Real-Time systems*, 2003, Architectural Press.
- P. Ficheux, *Linux Embarqué*, 3e édition, 2012, Eyrolles.
- Philipp Koopman, Better Embedded System Software, Drumnadrochit Education LLC, 2010.
- E. Helin, A. Renberg, *The little book about OS development*, `http://littleosbook.github.io/`, 2015.

Système d'exploitation

(*Operating System*)
Programme particulier qui gère la machine :

- Exécution des autres programmes
- Protection contre les fautes
- Gestion des ressources

Propriétés :

- Réactif
- Permanent
- Interfaces standard
- Économe

Temps Réel

Le temps intervient dans la correction (validité) du programme :

- réagir en un temps adapté aux événements externes,
- fonctionnement en continu sans réduire le débit du flot d'informations traité,
- temps de calculs connus et modélisables pour permettre l'analyse de la réactivité.
- **Latence** maitrisée.

Outils :

- horloges matérielles, interruptions, etc.
- style de programmation adaptés : multi-tâches, évenements,
- langages spécifiques (langages synchrones, etc.)
- outils de modélisation : logique temporelle, réseaux de Petri, etc.

Système / logiciel embarqué

- Association matériel + logiciel
- Tout ce qui n'est pas perçu comme un ordinateur
  - équipements industriels ou scientifiques
  - biens de consommation
- Ciblé : limité aux fonctions pour lesquelles il a été créé

- Fiable et sécurisé : autonomie, *systèmes critiques*
- Longue durée de vie
- Optimisé (processeurs moins puissants, contraintes temps-réel)
- Pas toujours un OS

Logiciel embarqué = temps réel ?

- Les applications embarquées historiques étaient TR
- Les systèmes d'exploitation embarqués propriétaires sont TR (VxWorks, ...) → RTOS
- L'apparition des OS libres dans l'industrie et dans l'embarqué modifie la donne !
  - Linux est utilisable dans l'industrie
  - Linux n'est pas TR
  - Linux peut être modifié pour être TR (PREEMPT-RT, Xenomai)
  - Il existe des systèmes TR légers et libres (RTEMS, FreeRTOS,...)

Quelques systèmes temps-réel

Une norme : **IEEE POSIX 1003.1**

Nom	Fournisseur	Posix	Architectures
iRMX/INtime	Intel / TenAsys `http://www.tenasys.com/`	-	ix86
QNX	QNX Software Systems / Blackberry `http://www.qnx.com/`	oui	ix86, arm
VxWorks	WindRiver Systems `http://www.windriver.com/`	-	arm, powerpc, ix86, mipx,...
LynxOS	Lynuxworks `http://www.lynuxworks.com/`	oui	ix86, powerpc
RTLinux	FSMLabs `http://www.fsmlabs.com/`	oui	ix86, powerpc,...
eCos	RedHat Labs `http://ecos.sourceware.org/`	oui	powerpc, ix86, μcontroleurs,...
RTEMS	OAR Corporation `http://www.rtems.org/`	oui	arm, x86, mips, sparc, avr,...
Xenomai	Philippe Gerum `http://www.xenomai.org/`	oui	ix86,ARM,...

`http://en.wikipedia.org/wiki/List_of_real-time_operating_systems`

Systèmes embarqués : architectures matérielles

Processeurs :

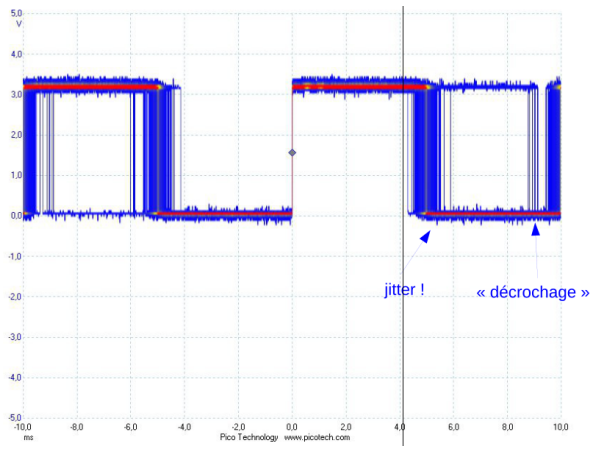
- micro-controleurs divers : monde industriel, hobbyistes (AVR / Arduino)
- ARM :
  - Cortex M (sans MMU), systèmes industriels,
  - Cortex A (MMU), Exynos, i.MX, Allwinnner,... : téléphones, tablettes, internet des objets,
- MIPS : équipements réseau, imprimantes,...
- SPARC : spatial (Leon)
- x86...

Bus dédiés :

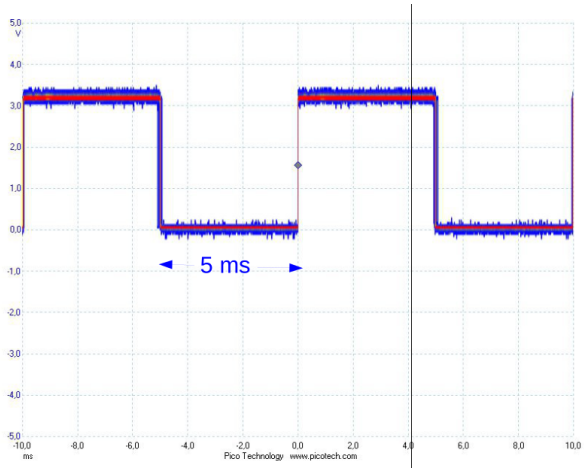
- CAN : bus temps réel (contrôle de process, automobile,...)
- I2C, SPI, OneWire : liaisons séries simples avec E/S bas débit

Linux et le temps réel

Sur un système chargé :



Noyau standard



Avec co-noyau Xenomai

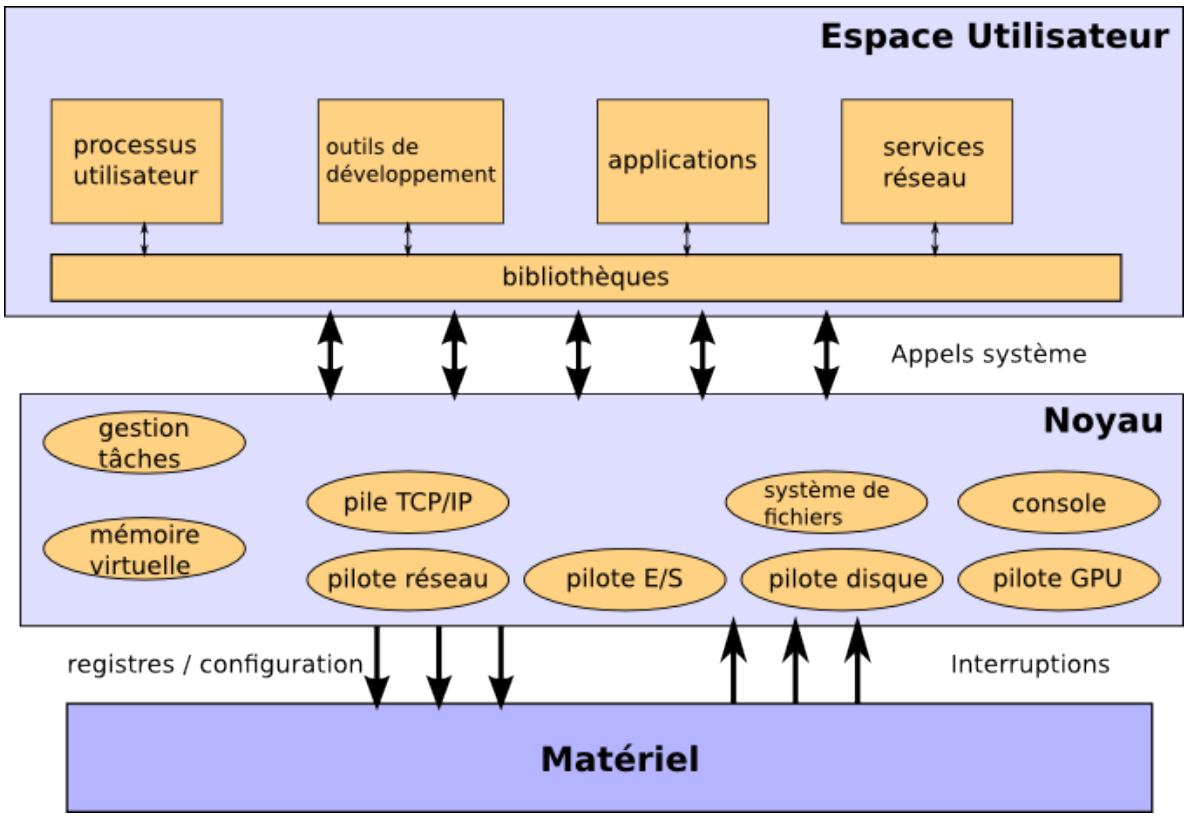
Xenomai

- Xenomai est un sous-système temps-réel de Linux
  - Programmation de tâches en espace utilisateur,
  - API d'application et de pilotes temps-réel (RTDM) dédiés.
- intégré au noyau Linux $\rightarrow$ « Real-time sub-system »
- support de nombreuses architectures
- dispose de « Skins » permettant d'émuler des API temps-réel (POSIX, VxWorks, VRTX,...)
- Plus complexe à mettre en œuvre que PREEMPT-RT mais performances 5 à 10 fois supérieures
- licence GPL (cœur), LGPL (interfaces, espace utilisateur)

RTEMS

- RTEMS = **R**eal **T**ime **E**xecutive for **M**ultiprocessor **S**ystems
- Initialement « Missile Systems » puis « Military Systems »
- Exécutif temps-réel embarqué diffusé sous licence libre (GPL avec exception)
- Pas exactement un système d'exploitation car mono-application (mais *multi-thread*)
- Programmation C, C++, Ada
- Plus de 100 BSP disponibles pour 20 architectures
- API RTEMS « classique » ou POSIX
- Utilisé par Airbus/EADS, ESA, NASA, NAVY,...

Architecture logicielle



Composants d'un système d'exploitation

Noyau partie centrale, ensemble de code qui gére les fonctionnalités du système :

- interfaces avec le matériel,
- démarrage et arrêt,
- exécution des programmes,
- gestion des privilèges des utilisateurs,
- fournit les services aux programmes utilisateurs,
- s'exécute en mode superviseur

Bibliothèques système bibliothèques de fonctions utilisées par les applications pour accéder aux services offerts par le noyau.

Utilitaires ensembles de programmes qui permettent le fonctionnement du système : services réseau, accès aux fichiers,...

Applications programmes exécutés par l'utilisateur.

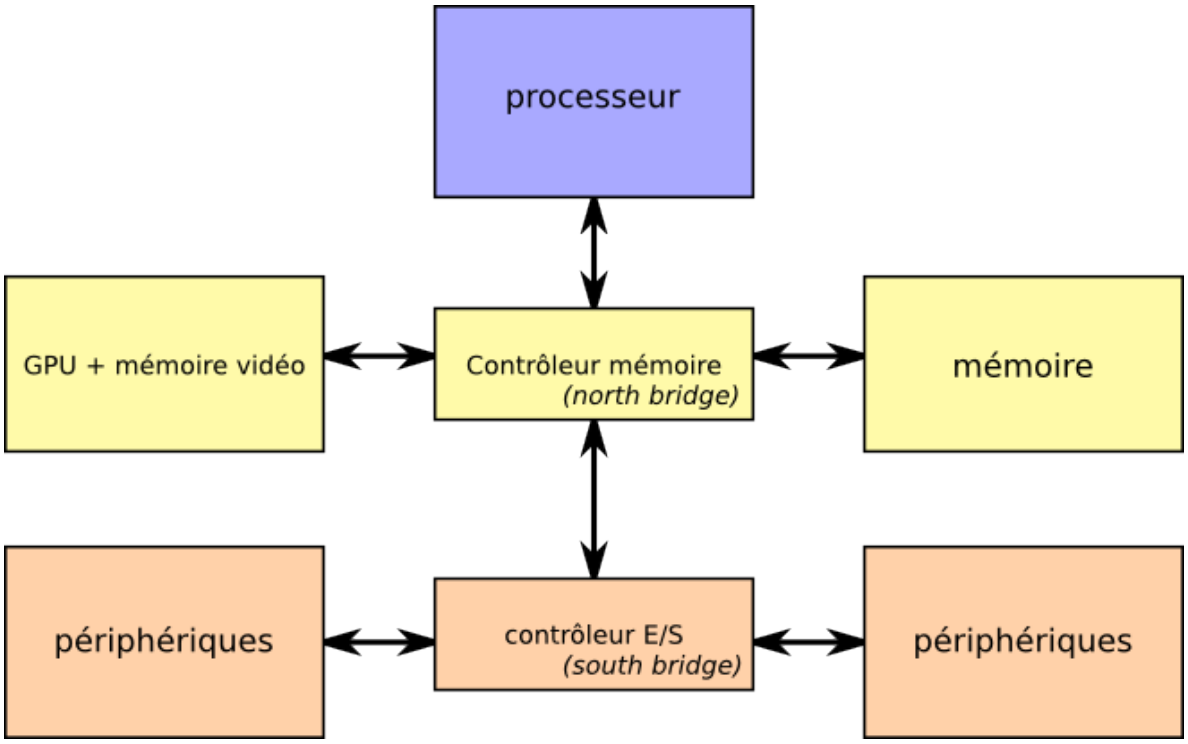
Composants d'un système d'exploitation (2)

Outils de développement compilateurs, assembleur, éditeur de liens, débuggeurs, fichiers d'entête et bibliothèques nécessaires pour le développement et la mise au point des logiciels pour ce système.

Développement croisé : machine cible $\neq$ machine hôte.

- chaîne de compilation croisée
- outils de chargement sur la machine cible
- outils de mise au point sur la machine cible :
  - émulateurs,
  - JTAG,
  - debogueur à distance

Mécanismes de base : architecture matérielle



Mécanismes de base : le processeur

Rôle : exécution des programmes
Caractérisé par : son jeu d'instructions, sa vitesse d'horloge.
Son **état** n'est observable qu'entre l'exécution de 2 instructions $\longrightarrow$ mot d'état du processeur.
Il dispose d'une pile en mémoire pour l'exécution des programmes (appel de sous-programmes, traitement d'interruptions).
Un **contexte** d'exécution d'un programme du compose de :

- compteur de programme
- pointeur de pile
- mot d'état
- registres généraux

Modes d'exécution :

- superviseur : toutes les instructions et toute la mémoire accessibles,
- utilisateur : accès restreint à certaines instructions et adresses.

Mécanismes de base : Les interruptions

Rôle : prendre en compte des événements externes au processeur (asynchrones).
La prise en compte d'une interruption provoque l'arrêt du programme en cours et l'exécution d'un sous-programme associé.

Système de gestion :
- non hiérarchisé
  toutes les interruptions ont la même priorité
- hiérarchisé
  les interruptions moins prioritaires n'interrompent pas le traitement des interruptions plus prioritaires

Conditions de prise en compte d'une interruption :

- Non masquable $\rightarrow$ doit toujours être traitée
- Masquables $\rightarrow$ peut être ignorée par le logiciel

Vecteur d'interruptions : table de pointeurs vers les routines de traitement.

Mécanismes de base : Les déroutements

Rôle : prendre en compte les événements internes au processeur.

- **détectés** par le processeur durant l'exécution d'un programme :
  - instruction inexistante,
  - violation de mode,
  - violation de protection mémoire,
  - erreur d'adressage (adresse inexistante),
  - erreur arithmétique (division par 0, débordement),...
- **demandés** explicitement par un programme :
  - appels système
  - exécution pas à pas

Mécanisme de prise en compte similaire à celui des interruptions.
Jamais masquables.

Mécanismes de base : les entrées/sorties (1)

Bus : dispositif matériel qui assure la connexion de périphériques au processeur.

Quelques bus standards :

PCI bus parallèle 32 ou 64 bits standardisé par Intel. Nombreuses variantes : PCI-Express, Compact-PCI, Mini-PCI,...

USB Universal Serial Bus : bus d'entrées sortie série grand public.

I^2C Inter-Integrated Circuit bus, standardisé par Philips. Bus série 2 fils simple pour interconnexion de périphériques. Utilisé pour capteurs de température, information sur les moniteurs, ASICS, ...

CAN Controller Area Network. Conçu par l'industrie automobile. Communication temps-réel entre micro-contrôleurs et capteurs/actionneurs.

Mécanismes de base : Les entrées/sorties (2)

Rôle : assurer la communication d'un programme avec le monde extérieur
Assurées par :

- des instructions spécialisées du processeur
- des zones mémoire spécialisées.

périphérique : dispositif d'entrée/sortie (terminal, imprimante, disque, capteur, actionneur,...)

contrôleur : prend en charge le dialogue avec le périphérique et présente une interface logique de haut-niveau au processeur.

driver, gestionnaire de périphérique : élément logiciel qui assure la communication avec un controleur ou un périphérique donné.

Processus : définition

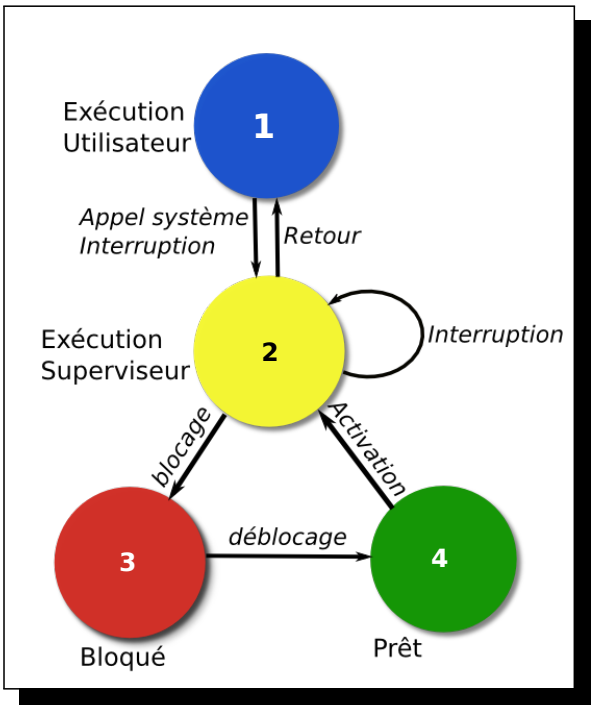
Programme séquentiel en exécution

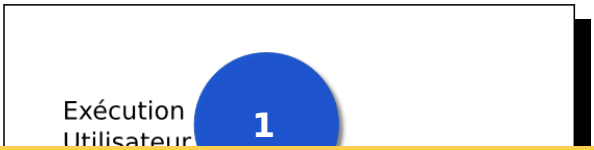
- Notion dynamique (activité)

Attention processus $\neq$ programme
(2 exécutions d'1 programme = 2 processus)

- une zone de code
- une zone de données statiques
- une zone de données allouées dynamiquement
- une pile

États d'un processus





Descripteurs de processus

L'existence d'un processus est matérialisée par une structure de donnnées contenant :

- l'état,
- le nom externe,
- l'identificateur du propriétaire,
- les données d'ordonnancement (priorité...),
- les pointeurs vers l'image mémoire,
- les données de comptabilité.

Les descripteurs de processus sont regroupés dans une *table des processus*.

Opérations sur les processus

Création

Demandée par un autre processus.

- allocation et initialisation d'un descripteur,
- copie en mémoire du programme à exécuter.

$\longrightarrow$ arbre des processus en cours.
Créé dans l'état prêt ou suspendu.

Destruction

Peut être demandée par :

- le processus lui-même,
- un autre processus,
- le noyau.

Libération des ressources associées.
Génération d'un événement.

Opérations sur les processus

Blocage

Passage en mode bloqué en attente d'un événement externe (condition logique).

- fin d'une entrée/sortie
- disponibilité d'une ressource
- interruption

Peut être demandé par le processsus ou forcé par le système
$\rightarrow$ *préemption*

Déblocage

Passage en mode prêt d'un processus bloqué lorsque l'évenement attendu se produit.

Activation

Passage en mode exécution d'un processus prêt.

Processus & Threads

Deux notions similaires, implémentation imbriquée..

- **Processus :** Objet « lourd »,
  - espace mémoire virtuelle séparé,...
  - droits d'accès par processus
- **Thread :** Objet plus léger,
  - inclus dans un processus,
  - tous les threads partagent l'espace mémoire virtuelle.
  - une pile par thread
  - Un seul utilisateur pour tous les threads d'un processus

En français : thread $\leftrightarrow$ fil

Processus : threads POSIX

```
#include <pthread.h>

int pthread_create(
    pthread_t *thread,              /* valeur retournee */
    pthread_attr_t *attr,           /* attributs du thread */
    void *(*start_routine)(void *), /* fonction a executer */
    void *arg                       /* argument */
);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void *status);
int pthread_detach(pthread_t thread);
int pthread_cancel(pthread_t thread);
```

Ordonnancement : définition

Choix du processus prêt à activer.

L'ordonnancement définit la manière dont est partagé le (ou les) processeur(s) disponible(s) pour l'exécution des processus.

Nécéssite des mécanismes particuliers :

- files de processus,
- gestion de priorités,
- préemption,
- attentes passives. (Blocage/Déblocage)

Peut être réalisé entièrement par le noyau ou par une tâche spécialisée (*scheduler*).

Ordonnancement non préemptif

L'opération de préemption n'existe pas.

Seul le processus en cours peut décider de quitter l'état d'exécution (blocage).
Exemples : MacOS $\leq$ 9, Windows 3.x,...

Solution simple (simpliste ?) pour le temps réel :

- gestion simple des ressources,
- garantit le temps d'exécution.

$\longrightarrow$ *Coroutines* (langage Go).

Ordonnancement préemptif

Une tâche en cours d'exécution peut être interrompue :

- par une tâche prête plus prioritaire
- par le scheduler après une certaine durée d'exécution

Permet un meilleur partage des ressources, mais nécessite des mécanismes de synchronisation entre les tâches.

Le temps de réponse dépend :

- de la priorité du processus
- des autres processus (nombre, charge, priorités)

Priorités entre processus

Permettent au scheduler de choisir entre plusieurs processus prêts celui qui va être exécuté.

Priorité statique à chaque processus est affecté un niveau de priorité fixe. Le processus prêt le plus prioritaire s'exécute. (VxWorks)

Priorité temporelle *(LRU – least recently used)* gestion de type First In First Out (FIFO).

Priorité dynamique lors de chaque ordonnancement, les priorités entre tâches sont calculées en fonction de différents critères :

- temps d'attente
- priorité *a priori*
- utilisation des ressources

Ordonnancement POSIX

```
#include <sched.h>

int sched_setscheduler(pidt pid, int policy,
                struct sched_param *params);
int sched_getscheduler(pid_t pid);
int sched_setparams(pid_t pid, struct sched_param, *params);
int sched_yield(void);
```

Policy :

SCHED_FIFO ordonnancement préemptif, basé sur les priorités

SCHED_RR ordonnacement préemptif, basé sur les priorités et qantas de temps

SCHED_OTHER *autre* ordonnanceur, dépendant de l'implémentation.

Synchronisation des processus : pourquoi ?

- **Partage de ressources** : garantir que l'exécution en parallèle de plusieurs processus fournit le même résultat qu'une exécution strictement séquentielle (pas d'interférences)
- **Communication, coopération** : garantir que l'échange d'information entre tâches obeit à un protocole défini.

Synchronisation des processus : section critique

Exemple : mise à jour de comptes bancaires

```
    VARIABLE val : ENTIER

    PROCESSUS créditeur(c : ENTIER)
        DÉBUT
(1)     val ← val + c
        FIN
```

```
    PROCESSUS débiteur(d : ENTIER)
        DÉBUT
(2)     SI val < d ALORS
(3)         ÉCRIRE("Découvert...")
(4)     val ← val - d
        FIN
```

(chaque ligne est supposée indivisible)

Pb : si on exécute :
$\{$ val $= 10$ $\}$ debiteur(9) $\|$ débiteur(9)

avec la séquence : $(2)^1$ $(3)^1$ $(2)^2$ $(3)^2$ $(4)^1$ $(4)^2$

Aucun découvert ne sera signalé, mais à la fin $\{$ val $=$ -8 $\}$

$\Longrightarrow$ conflit d'accès à la variable partagée "val".

Synchronisation des processus : exclusion mutuelle

Exemple : compteur /statistiques

```
    VARIABLE cpt : ENTIER

    PROCESSUS compteur
        CYCLE
(1)         Attendre événement
(2)         cpt ← cpt + 1
        FIN
```

```
    PROCESSUS statistique
        CYCLE
(3)         Attendre(periode)
(4)         ÉCRIRE(cpt, 'événements')
(5)         cpt ← 0
        FIN
```

Montrer que certaines séquences conduisent à la perte d'événements.

Les séquences $\{(2)\}$ et $\{(4);(5)\}$ doivent s'exclure.

Synchronisation des processus : solutions

Une solution au problème de l'exclusion mutuelle est correcte si elle vérifie les propriétés :

1. La solution est *indépendante de la vitesse* d'exécution des programmes.
2. Deux processus (ou plus) *ne peuvent se trouver simultanement* en section critique.
3. Un processus hors de sa section critique et qui ne demande pas à y entrer *ne doit pas empecher un autre* d'entrer en section critique.
4. Deux processus *ne doivent pas s'empecher mutuellement et indéfiniment* d'enter en section critique (interblocage).
5. Un processus entre toujours en section critique *au bout d'un temps fini* (pas de famine).

Une solution matérielle : Test And Set

Une instruction indivisible permet de réaliser la fonction :

```
bool test_and_set(bool *var) {
    bool retval = *var;              /* Lecture */
    *var = TRUE;                     /* Ecriture */
    return retval;
}
```

L'exclusion mutuelle est réalisée par une variable booléene :

```
bool busy = FALSE;
```

L'entrée en exclusion mutuelle est assurée par :

```
while (test_and_set(busy));
```

La sortie par : `busy = FALSE;`

Défaut : Cette solution provoque des attentes *actives*.

Test and Set en langage C

Primitives implementées par le compilateur C.
« standard » proposé par Intel

```
type __sync_lock_test_and_set(type *ptr, type value, ...)
```

Ecrit *value* dans **ptr* et retourne la valeur précédente de **ptr* ;

```
void __sync_lock_release(type *ptr, ...)
```

Remet le contenu de **ptr* à zéro

```
int lock = 0;
...
while (__sync_lock_test_and_set(&lock, 1) == 1)
      sched_yield();
/* Section critique */
...
__sync_lock_release(&int);
...
```


Sémaphores : définition

(Dijkstra)
Le sémaphore est un objet sur lequel seulement 2 opérations sont possibles : **P**(s) et **V**(s), toutes 2 atomiques.

Un sémaphore possède une valeur entière, définie entre 2 opérations.

- **P**(s) décrémentation de la valeur du sémaphore et blocage du processus appelant si la valeur est devenue < 0
- **V**(s) incrémentation de la valeur du sémaphore pouvant entrainer le déblocage d'un processus bloqué par **P**.

P est une barrière et **V** un laisser-passer.
Note : **P**asseren, **V**rygeven (néerlandais) = Prendre et libérer.

Sémaphores binaires

Un sémaphore binaire est réalisé avec une variable booléene et une file d'attente de processus.

```
struct semaphore_t {
    bool is_free;
    process_queue_t fifo;

    public semaphore_t semaphore_t(bool val) {
        is_free = val;
        queue_init(self->fifo);
        return (self);
    }
}
```


Sémaphores binaires (2)

```
public void P(semaphore_t s) {
    if (s->is_free == FALSE) {
        queue_insert(s->fifo, proc_self());
        proc_block(proc_self());
    }
    is_free = FALSE;
}
```

```
public void V(semaphore_t s) {
    process_t px;
    s->is_free = queue_empty_p(s->fifo);
    if ( !queue_empty_p(s->fifo) {
        px = queue_first(s->fifo);
        proc_unblock(px);
    }
}
```

Sémaphores d'exclusion mutuelle

Un sémaphore binaire initialisé à **TRUE** est un sémaphore d'exclusion mutuelle.

```
semaphore_t mutex = new sempahore_t(TRUE);
    P(mutex)
        ...
        // Section critique
        ...
    V(mutex)
```

- Protège une section critique.
- Les sémaphores d'exclusion mutuelle peuvent être utilisés pour assurer l'accès en exclusion mutuelle à des ensembles disjoints de variables partagées. Il suffit d'associer un sémaphore d'exclusion mutuelle à chaque ensemble.

Mutex POSIX

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Sémaphores privés

Seul le processus propriétaire du sémaphore peut exécuter l'opération **P**.
Les autres processus ne peuvent exécuter que l'opération **V**.
Utilisation :
Lorsqu'un processus doit être activé, réveillé par un autre sur l'occurence d'un événement (délai, comptage...)

Processus réveilleurs			Processus esclave
P1	...	P2	**CYCLE**
...	...	...	...
V(sp)	...	**V**(sp)	**P**(sp)
...	...	...	...

En général, la valeur initiale d'un sémaphore privé est 0.

Sémaphores généraux

- Sémaphores dont le compteur n'est pas limité à 0..1.
- Utilisés comme compteurs de ressources.
- Valeur initiale = nombre de ressources disponibles.

Lorsqu'un processus désire acquérir une ressource, il exécute une opération **P**. Il est donc bloqué si aucune ressource n'est disponible.

Lorsqu'il libère la ressource, il exécute une opération **V** qui signale la disponibilité de la ressource et débloque donc un processus éventuellement en attente.

Exemple : gestion d'un pool d'imprimantes

Exemple : producteur / consommateur

Le système dispose d'un ensemble de n emplacements capables de stocker de l'information.

- Les processus **producteurs** produisent de l'information vers ces emplacements.
- Les processus **consommateurs** utilisent cette information (et libère la place.

Comment synchroniser les deux types de processus pour ne pas perdre de données ? (Bloquer un producteur si plus de place / Bloquer un consommateur si pas d'information disponible) ?

Semaphores POSIX

```
#include <semaphore.h>

int sem_init(sem_t *semaphore, int pshared,
             unsigned int initial_value);
int sem_destroy(sem_t *semaphore);

int sem_wait(sem_t *semaphore);
int sem_trywait(sem_t *semaphore);
int sem_post(sem_t *semaphore);
```

Moniteurs

Concept de la programmation orientée objet.

- constructeur de type spécialisé dans la synchronisation.
- enferme un type de donnée et les opérateurs qui le manipulent.
- évite la dispersion des opérations de synchronisation.

Sémantique :

> *L'exécution des opérateurs dans un moniteur garantit l'exclusion mutuelle entre ces opérations.*

Méthode de synchronisation utilisée par le langage **Java** (mot clé `synchronized`).

Moniteurs : variables conditions

Utilisées pour bloquer un processus dans un moniteur.
2 opérations possibles sur une variable condition **c** :

wait : provoque le blocage du processus appelant et la libération de l'accès au moniteur.

signal : est une opération vide si aucun processus n'est bloqué sur cette condition. Sinon, le processus "signaleur" est suspendu et un processus bloqué sur **c** est réactivé. Le processus signaleur ne reprend le contrôle du moniteur qu'après la fin de la dernière opération en cours sur celui-ci.

Exemple : Philosophes et spaghettis.

Philosophes et spaghettis

Cinq philosophes sont dans une pièce avec une table ronde sur laquelle se trouvent cinq assiettes contenant des spaghettis.

- Chaque philosophe a sa place attitrée.
- Chaque philosophe alterne entre faire de la philosophie (debout) et manger des spaghettis (assis).
- Comme les philosophes sont peu sociaux, ils ne s'assoient à table que si les deux places de part et d'autre de la leur sont libres.

Représenter le système sous forme de processus indépendants (les philosophes) mais synchronisés.

Variables Condition POSIX

Utilisées avec un mutex.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
            pthread_condattr_t *attributes);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           struct timespec *expiration);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Tickets locks

Sémaphores classiques : l'ordre de déblocage après un **V** n'est pas défini.

Mécanisme de ticket : gestion explicite de la file d'attente.

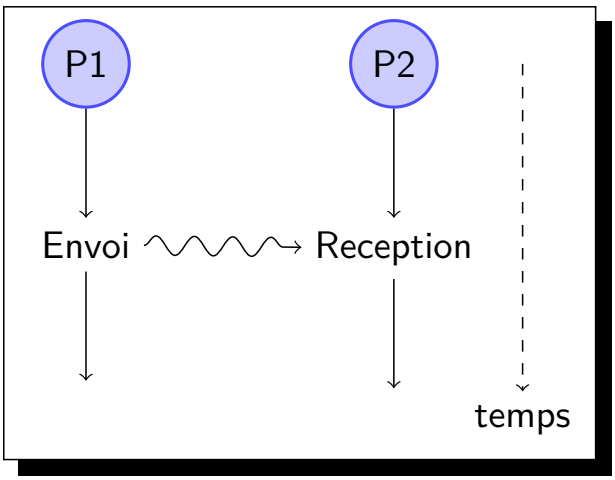
VAR courant : **entier** = 0, suivant : **entier** = 0, c : **condition**

MONITEUR prendre_ticket
    mon_ticket ← suivant
    suivant ← suivant + 1
    TANT QUE courant ≠ mon_ticket
        WAIT(c)
    FIN TANT QUE
FIN

MONITEUR libérer_ticket
    courant ← courant + 1
    COND_BROADCAST(c)
FIN

Synchronisation par messages

- Object unique utilisé à la fois pour la synchronisation et l'échange de données.
- modèle de transfert de messages synchrone ou asynchrone



- Utilisé aussi entre processeurs sans mémoire partagée

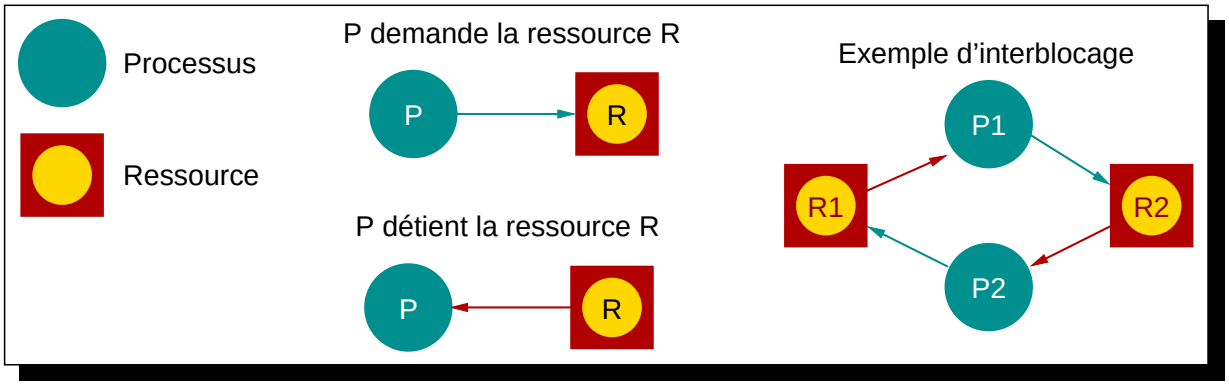
Interblocage

(Dead Lock)

Définition
Un processus est en état d'interblocage s'il est bloqué en attente d'une condition qui ne se produira *jamais*.
Conditions nécéssaires :

1. Les processus accèdent en exclusion mutuelle à des ressources critiques.
2. Les processus gardent les ressources aquises pendant qu'ils sont bloqués en attente d'une ressource.
3. Les ressources acquises par un processus ne peuvent lui être retirées de l'extérieur.
4. Il existe un cycle de processus où chacun attend une ressource acquise par un autre.

Interblocage : représentation



Représentation graphique des cycles de dépendances.

Interblocage : traitement

2 approches :

- Prévenir (éviter)
  - prévention (statique)
  - esquive (dynamique)
- Guérir (vivre avec)
  - détection
  - récupération

Interblocage : prévention

Prévention statique
Il suffit qu'une seule des conditions nécéssaires (1, 2, 3, 4) soit fausse :

- Pas de partage de ressources (pas intéressant)
- Allocation globale des ressources en une seule fois
- Libération des ressources acquises avant blocage (allocation conditionnelle)
- Ordonnancement de l'allocation des ressources

Esquive : Algorithme du banquier : (Dijkstra)
Chaque processus annonce le nombre max de ressources de chaque type qu'il allouera.
Chaque demande d'allocation peut alors être définie comme "sûre" s'il reste suffisament de ressources pour satisfaire le pire cas.
(La condition nécéssaire 4 est évitée).

Interblocage : détection

Identification :

→ des processus en étreinte fatale

→ des ressources concernées

Construction du graphe “de propriété” et “de requête”.
Réduction :
Suppression des arcs constituant le blocage (boucle).

Mécanismes de base : la mémoire

Rôle : stockage des programmes et des données en cours d'exécution.

- adressable par mots de 8, 16, 32 ou 64 bits
- découpée en blocs physiques de 64, 128, 256 koctets ou 1 Moctet.
- accessible simultanément par plusieurs éléments.
- gestion optimisée des temps d'accès.

La mémoire

→ **uniforme**
- Un seul espace d'adressage
- Un seul problème : le découpage de cet espace entre les processus.

→ **hiérarchisée**
- Plusieurs espaces d'adressage
- Notion de hiérarchie entre ces espaces (mémoire primaire, secondaire. . .)
- Exemples : mémoire virtuelle, mémoire cache, mémoire segmentée
- Problèmes : migration entre les niveaux, choix du niveau. . .

Contenu de la mémoire

<table>
<tr><th colspan="3">code</th><th colspan="3">données</th></tr>
<tr><td colspan="2">commun</td><td>privé</td><td colspan="2">communes</td><td>privées</td></tr>
<tr><td>réentrant</td><td>critique</td><td></td><td>partageables
(constantes)</td><td>critiques</td><td></td></tr>
</table>

Mémoire uniforme

Un seul espace d'adressage `MIN..MAX`.
Taille adressable par un processus $\leq$ taille max disponible.
Problème de placement en mémoire :

- partition fixe de la mémoire (ex. MSDOS)
- Allocation dynamique des partitions
  - $\rightarrow$ risque d'interblocage
  - $\rightarrow$ fragmentation
  - $\rightarrow$ fuites

Algorithmes d'allocation par zones

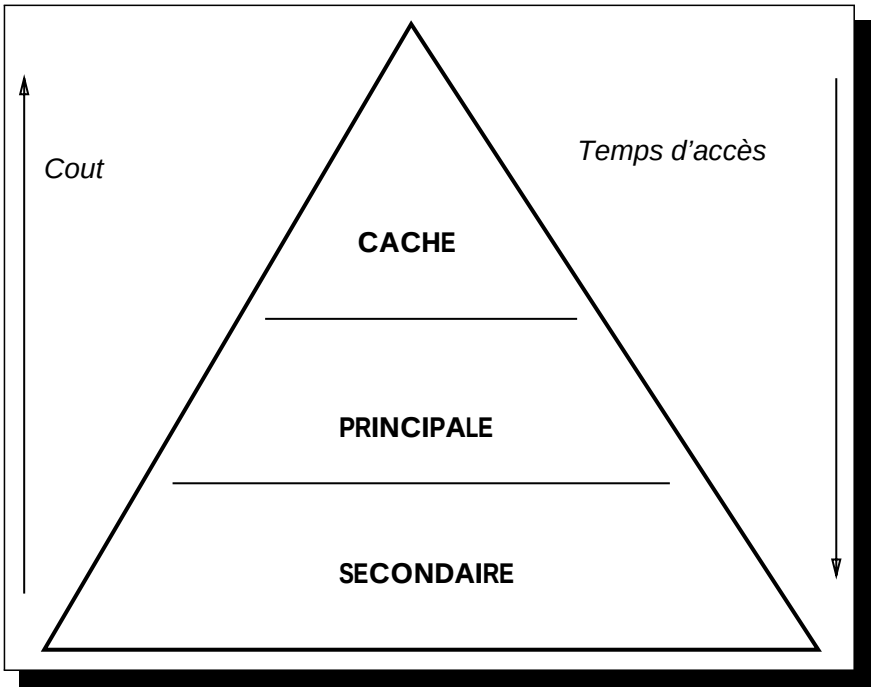
Principe : La mémoire est structurée en zones de taille variables qui sont l'unité d'allocation élémentaire. Une zone est libre ou allouée.
Choix d'une zone libre :

best-fit	la plus petite zone de taille $\geq n$
first-fit	la première zone de taille $\geq n$
worse-fit	la plus grande zone (allocation avec résidu)

Libération

- simple marquage
- fusion des zones libres adjacentes
- ramasse-miettes

Systèmes à mémoire hiérarchisée



Techniques de va et vient

(swapping)

global réquisition globale de l'espace nécéssaire à un processus en mémoire principale et libération de la mémoire secondaire.

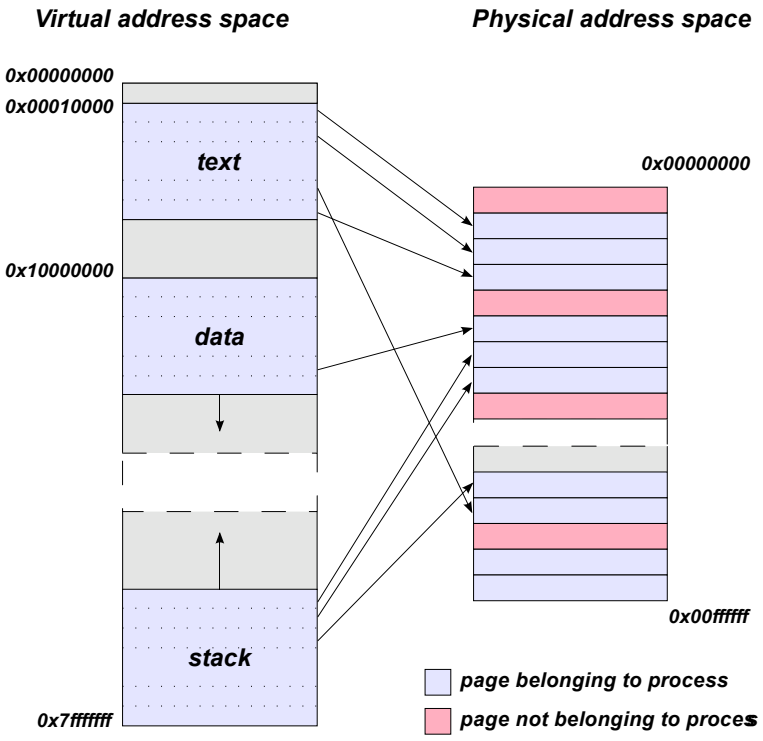
mémoire virtuelle Espace d'adressage virtuel unique géré par le système, découpé en pages qui font le va-et-vient.

- pages de taille fixe
- pages de taille variable (segments)

Problème de la correspondance entre adresses physiques et logiques.

- technique directe
- technique associative
- technique associative par groupes

Mémoire virtuelle et swap



Fonctionnement

- **TLB :** Translation Lookaside Buffer : cache des traductions d'adresse récentes.
- **PT :** Page Table : table des pages → contient pour chaque page sa localisation réelle.

Traduction d'adressses

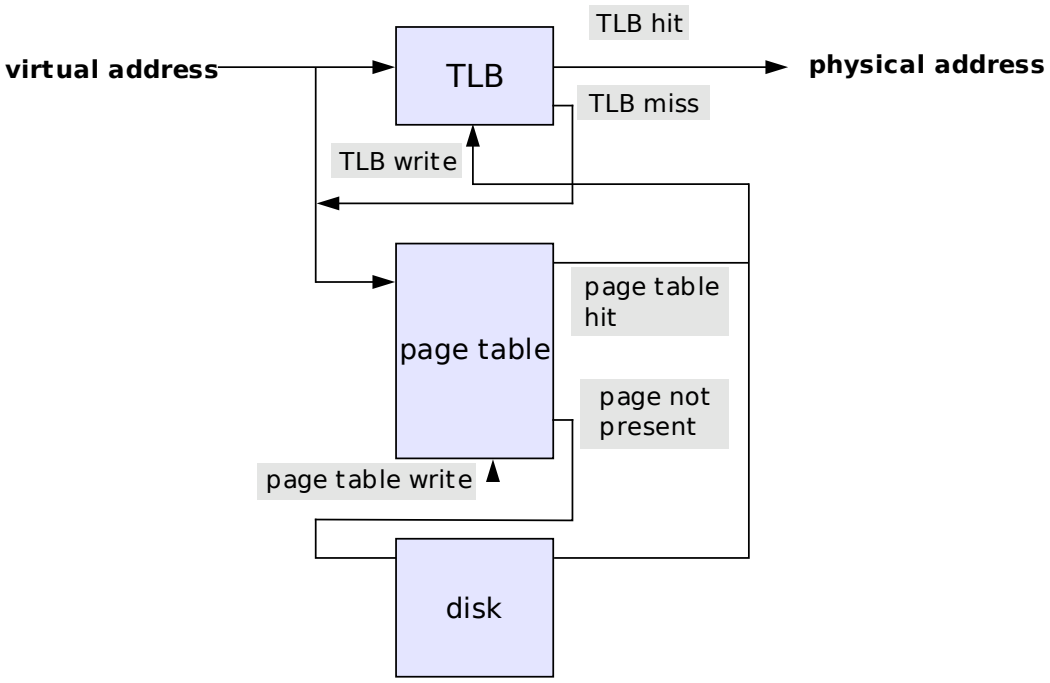


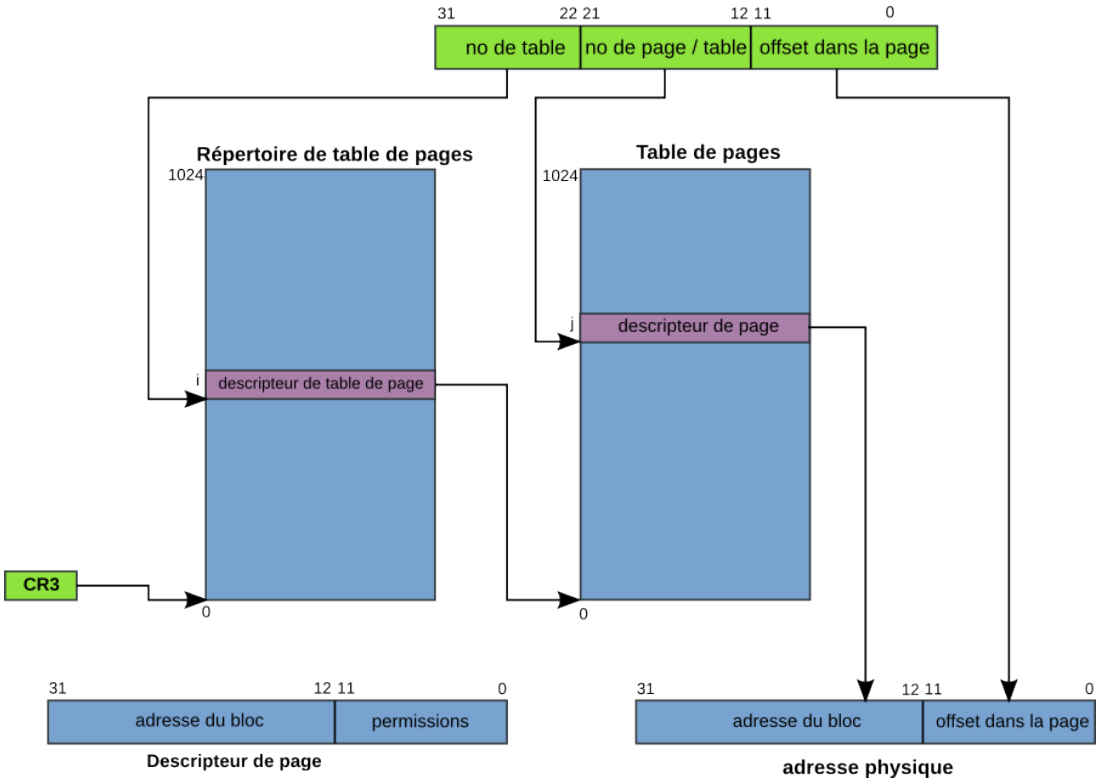
Table des pages : cas des processeurs x86

Table à 2 niveaux :

- **PDT :** Page Directory table : répertoire des pages
  10 bits
- **PT :** Page Table.
  10 bits

Taille d'une page : 4096 octets (2^{12}).

Table des pages : cas des processeurs x86



Exemple

Adresse virtuelle : `0xff1f5da0`

- Offset dans la page (12 derniers bits) : `0xda0`
- Page directory address : `0x3fc`
- Page table address : `0x1f5`
- Lecture de `PDT[3FC]` $\rightarrow$ `0x0103b164`
- Lecture de `PT[PDT[3FC]+1F5]` $\rightarrow$ `0x0017d000`
- Ajout offset $\rightarrow$ adresse physique `0x0017dda0`

Trashing

Lorsque la charge d'un système augmente, le nombre de pages libres en mémoire principal chute.
$\longrightarrow$ de plus en plus de va-et-vient.
Il existe un seuil au dessus duquel le systeme ne peut plus fonctionner.
Nécessité d'un mécanisme de limitation de la charge.

Horloges et mesure du temps

Deux types d'horloges :

- Horloge du CPU (ticks) (*TSC*, *HPET*,...)
- Horloge murale (temps universel)

Variations de la fréquence de l'électronique : $\rightarrow$ écarts entre les deux.

NTP (**N**etwork **T**ime **P**rotocol) :
protocole de synchronisation des horloges via internet sur temps universel.

Temps universel POSIX

```
#include <time.h>

time_t time(time_t *tloc);
```

`time(NULL)` retourne le nombre de secondes écoulées depuis le 1er janvier 1970 à 00 :00 UTC.

- `time_t` est un entier signé 32 bits. (débordement le 19 janvier 2038)
- le noyau utilise l'heure UTC. Gestion du fuseau horaire (timezone) dans les bibliothèques.

Temps universel décodé

```
#include <sys/types.h>
#include <time.h>

struct tm *localtime_r(const time_t *clock, struct tm* tp);
time_t mktime(struct tm* t);

struct tm {
        int tm_sec;     /* seconds (0 - 60) */
        int tm_min;     /* minutes (0 - 59) */
        int tm_hour;    /* hours (0 - 23) */
        int tm_mday;    /* day of month (1 - 31) */
        int tm_mon;     /* month of year (0 - 11) */
        int tm_year;    /* year - 1900 */
        int tm_wday;    /* day of week (Sunday = 0) */
        int tm_yday;    /* day of year (0 - 365) */
        int tm_isdst;   /* is summer time in effect? */
        long tm_gmtoff; /* offset from UTC in seconds */
        char *tm_zone;  /* abbreviation of timezone name */
}
```

Horloges POSIX

```
#include <time.h>
int clock_gettime(clockid_t, struct timespec *tp);

struct timespec {
      time_t tv_sec;  /* seconds */
      long tv_nsec;   /* nanoseconds */
};
```

clockid_t :

CLOCK_REALTIME temps universel ;

CLOCK_MONOTONIC horloge CPU ;

CLOCK_PROCESS_CPUTIME_ID horloge du processus ;

Timers

Mécanisme de déclenchement d'une interruption (signal) dans le futur.

- périodique ou non.
- exécution d'une fonction associée au moment du déclenchement.
- contexte interruption : sections critiques, fonctions interdites.
- peuvent être associés à différentes horloges.

Utilisations :

- Permettent le cadencement précis d'un traitement périodique
- Mécanisme de garde (*watchdog*) pour traitements blocants

Timers POSIX

```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *sevp,
                 timer_t *timerid);
int timer_settime(timer_t timerid, int flags,
                 const struct itimerspec *new_value,
                 struct itimerspec *old_value);

struct itimerspec {
        struct timespec it_interval;  /* Timer interval */
        struct timespec it_value;     /* Initial expiration */
};
```

`sevp` défini le signal qui est déclenché (`SIGALRM`)
`signal(SIGALRM, fonction)` permet de définir la fonction qui sera appelée.

Quelques questions ?

Quelques choix à faire pour implémenter un système temps-réel :

- Mode superviseur ou mode utilisateur ?
- Noyau monolithique ou micro-noyaux ?
- Noyau preemptif ou non ?
- Gestion de la mémoire : mémoire virtuelle ou non ?
- Système général ou dédié ?
- Système d'exploitation ou pas ?

(A) Allocation de ressources

Contrôler l'accès à un pool de n imprimantes par plusieurs utilisateurs.

Écrire les procédures d'allocation/dé-allocation telles que :

- chaque imprimante n'est utilisée que par un utilisateur à la fois
- si au moins une imprimante est libre, un utilisateur n'attend pas
- si toutes les imprimantes sont occupées, l'allocation bloque l'utilisateur en attendant qu'une se libère
- si plusieurs utilisateurs sont en attente, le premier arrivé sera le premier servi.

(B) Producteurs/consomateurs

Gestion de la communication avec une file d'attente entre processus

→ Messages de taille fixe

→ Zone d'échange circulaire permettant de stocker n messages

Écrire les procédure de dépot et de lecture de messages telles que :

- si il reste de la place dans le buffer, le dépot ne bloque pas.
- si au moins un message est disponible dans le buffer, la lecture retourne immédiatement ce message
- si le buffer est plein (resp. vide), le dépot (resp. la lecture) sera bloqué en attendant la libération (resp. le dépot) d'un message.
- si plusieurs processus sont bloqués, ceux-cis seront débloqués de manière équitable.

(C) Ping-pong

Créer un mécanisme de synchronisation entre 2 processus permettant de garantir qu'ils s'exécutent chacun à tour de rôle.

Le premier affiche « **ping** » et le second affiche « **pong** » lors de chaque itération.

Utiliser pour cela un moniteur et une variable condition.