Systèmes embarqués et temps réel

Dr ABDELLAOUI Ghouti

ESSA Tlemcen, Algeria

19 octobre 2023

Plan du cour

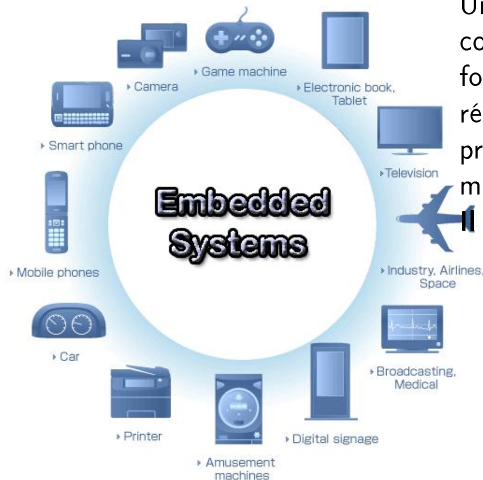
1. Système d'exploitation embarqué temps réel
   - Notions sur les système d'exploitation
     - Présentation
     - Historique
     - Architecture
   - Caractéristiques
   - Processus
   - Ordonnancement
   - Synchronisation et communication inter-tâches
2. Les Systèmes temps réel embarqué
   - Présentation
   - Domaines d'applications
   - Caractéristiques
   - Intégration du technologie (Plate-formes, SE, langages de programmation)
   - Contraintes temporelles et modélisation (Gantt)
   - Criticité
     - Criticité
   - Les différentes vue d'un système temps réel embarqué
     - Vue métier
     - Vue de l'automaticien
     - Vue de l'informaticien

Système d'exploitation embarqué temps réel

Présentation

Qu'est ce qu'un système embarqué ?

Présentation



Un système embarqué est défini comme un système électronique et informatique autonome, souvent temps réel, spécialisé dans une tâche bien précise. Ses ressources sont généralement limitées.

Il sont partout ! ! !

- Radio/réveil
- Machine à café
- Télévision / télécommande
- Moyen de transport (voiture : à foison !)
- Téléphone portable ....

Présentation

Certains systèmes embarqués sont complexes,ils nécessitent un système d'exploitation pour gérer tous les ressources matérielles.

Présentation

Qu'est ce qu'un système d'éxploitation ?

Présentation

En informatique un système d'exploitation est une interface logiciel qui permet de faire le lien entre :

- L'utilisateur,
- Les applications,
- Les composantes matérielles d'un ordinateur.

Présentation

Exemples des systèmes d'exploitation :

- Windows : Windows 3.11, Windows 95, Windows 98, Windows Xp ,...
- Linux : Redhat, Debian, Kali linux,buntu,Mint,...
- Mac OS : OSx

Présentation

Exemples des systèmes d'exploitation mobile :

- Android : Android Alpha, Android Bêta, Android 1.0,..., Marshmallow, Nougat, Oreo
- IOS : IOS1,IOS2,...,IOS11
- symbian
- Blackberry : OS3, OS4,..., OS 10.3.2
- Windows Phone : NoDo,Mango,Refresh,Tango, Apollo, ...

FreeRTOS un système d'exploitation temps réel libre et open source développé par Real Time Engineers Ltd.

Architectures matérielles supportées :

- Altera : Nios II
- ARM architecture : ARM7, ARM9, ARM Cortex-M0, ARM Cortex-M3, ARM Cortex-M4, ARM Cortex-M7
- Atmel : Atmel AVR, AVR32, SAM3, SAM7, SAM9
- Intel : x86, 8052
- Microcontrôleur PIC : PIC18, PIC24, dsPIC, PIC32
- Xilinx : MicroBlaze
- ...

Présentation :Système d'exploitation embarqué temps réel

QNX est conçu principalement pour le marché des systèmes embarqués tels que les voitures mais aussi pour les industries et les services médicaux des hôpitaux. Il utilise un micro-noyau et l'entreprise qui le développe appartient à Blackberry.

Présentation :Système d'exploitation embarqué temps réel

Historique

Histoire des ordinateurs :

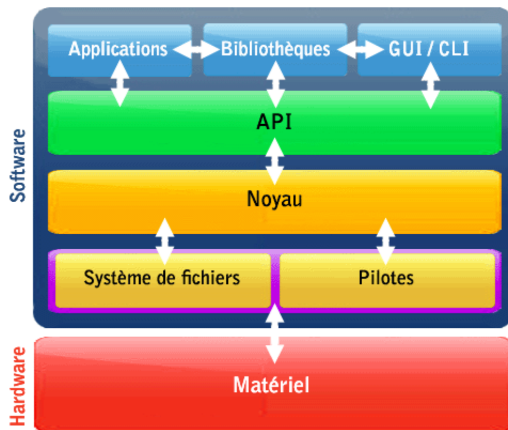
- 1945 - 55 : tubes et interrupteurs
  - Pas de système d'exploitation
- 1955 - 65 : transistors, cartes perforées
  - Traitement par lots
- 1965 - 80 : circuits intégrés, disques
  - Multiprogrammation, temps-partagé, entrées/sorties
  - Unix, version BSD, AT&T(**American Telephone & Telegraph**), interface POSIX
- 1980 – : ordinateurs personnels (PC)
  - Interface graphique (concept crée vers 1960, Stanford)
  - Réseaux et systèmes distribués

Historique

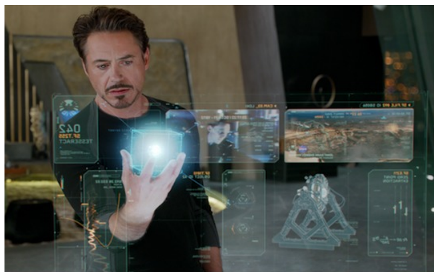
Histoire des systèmes d'exploitation :

- CP/M (**Control Program for Microprocessors**)(depuis 1974), Digital Research
- UNIX (depuis 1969-1979), premier par AT&T
- MS-DOS (depuis 1981), Microsoft
- MacOS (depuis 1984), Apple
- Windows (depuis 1991), Microsoft
- Linux (depuis 1992), OpenSource

Architecture



Architecture : L'interface utilisateur



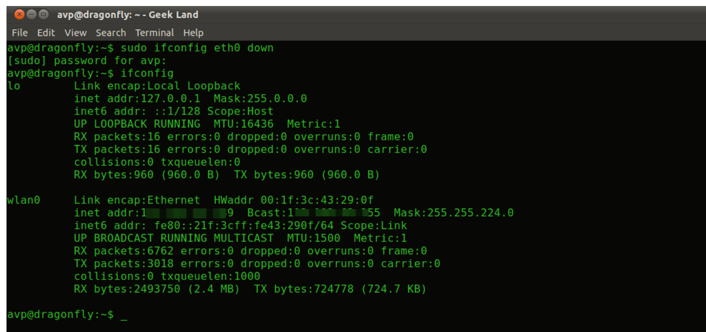
L'interface utilisateur aussi nommée interface homme-machine (IHM) permet à un homme de se servir de l'ordinateur.

Au niveau matériel, elle consistait à l'origine en des interrupteurs et des lampes, puis des cartes perforées. De nos jours, la plupart des ordinateurs sont équipés de clavier, souris et moniteur.

Cette interaction homme machine s'élabore tant par le biais d'interfaces graphiques qu'en ligne de commande par le « Shell».

La ligne de commande

La ligne de commande (en anglais CLI pour Command Line Interface) était la seule interface disponible sur les ordinateurs des années 1970. Elle est encore utilisée en raison de sa puissance, de sa grande rapidité et du peu de ressources nécessaires à son fonctionnement.



Architecture : L'interface utilisateur

L'interface graphique

L'interface graphique (en anglais GUI pour Graphical User Interface) s'oppose à l'interface en ligne de commande.

Les parties les plus typiques de ce type d'environnement sont le pointeur de souris, les fenêtres, le bureau, les icônes.

D'autres contrôles graphiques sont couramment utilisés pour interagir avec l'utilisateur : les boutons, les menus, les barres de défilement.

Architecture : Application

C'est un ensemble de programmes informatiques qui aident un utilisateur à effectuer un certain travail..

Architecture : Les bibliothèques

Les bibliothèques servent à regrouper les opérations les plus utilisées dans les programmes informatiques, afin d'éviter la redondance de la réécriture de ces opérations dans tous les programmes.
On distingue deux types de bibliothèques :

- Les bibliothèques système : Les bibliothèques système sont constituées de fonctions permettant l'utilisation agréable des fonctionnalités système (généralement des points d'entrée vers des fonctions du noyau, mais pas seulement),
- Les bibliothèque utilitaires :Les bibliothèques utilitaires contiennent des fonctions d'usage courant et pratique (fonctions mathématiques, fonctionsde tri, etc).

Architecture :Interface de programmation d'application API

Une interface de programmation est un ensemble de fonctions permettant d'accéder aux services d'une application par l'intermédiaire d'un langage de programmation.

Elle masque, au développeur, la complexité de l'accès à un système ou à une application en proposant un jeu de fonctions standard.

Le développeur n'a donc pas à se soucier de la façon dont une application distante fonctionne, ni de la manière dont les fonctions ont été implémentées pour pouvoir l'utiliser dans un programme.

Architecture :Le noyau (*kernel* en anglais)

Les noyaux ont comme fonction de base d'assurer le chargement et l'exécution des processus, de gérer les entrées-sorties et de proposer une interface entre l'espace noyau et les programmes de l'espace utilisateur.
Ils assurent les fonctionalités suivantes :

- gestion des périphériques (au moyen de pilotes) ;
- gestion de l'exécution des programmes (aussi nommés processus) :
  - gestion de la mémoire attribuée à chaque processus ;
  - ordonnancement des processus (répartition du temps d'exécution sur le ou les processeurs).
  - synchronisation et communication entre processus (services de synchronisation,d'échange de messages, mise en commun de segments de mémoire, etc.)
- gestion des fichiers (au moyen de systèmes de fichiers) ;
- gestion des protocoles réseau (TCP/IP, IPX, etc.).

Architecture :Le système de fichiers

Un système de fichiers (FS ou filesystem en anglais) ou système de gestion de fichiers (SGF) est une structure de données permettant de stocker les informations et de les organiser dans des fichiers sur ce que l'on appelle des mémoires secondaires (disque dur,disquette, CD-ROM, clé USB, etc.).
Ce stockage de l'information est persistant. Une telle gestion des fichiers permet de traiter et de conserver des quantités importantes de données ainsi que de les partager entre plusieurs programmes informatiques. Il offre à l'utilisateur une vue abstraite sur ses données et permet de les localiser à partir d'un chemin d'accès.

Architecture :Le système de fichiers

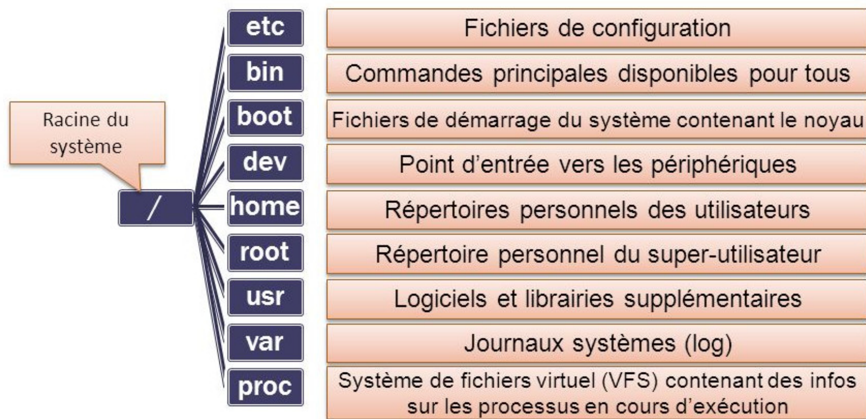


Figure – Système de fichiers Linux

Architecture :Les pilotes (*Drivers* en anglais)

Un pilote informatique souvent abrégé en pilote (driver) est un programme informatique, destiné à permettre à un autre programme (souvent un système d'exploitation) d'interagir avec un périphérique. En général, chaque périphérique a son propre pilote. Sans pilote, l'imprimante ou la carte graphique ne pourraient pas être utilisées.
Certains systèmes d'exploitation comme Windows proposent leurs propres pilotes génériques. Si ces pilotes gèrent les grandes fonctions communes à tous les matériels, ils n'ont pas toujours toutes les capacités des pilotes de constructeurs.

Architecture

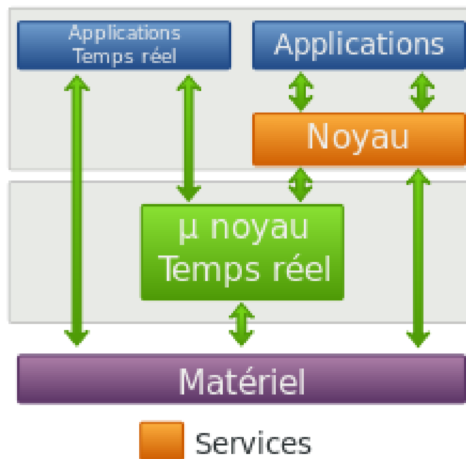


Figure – Architecture d'un système d'exploitation temps réel

OS+μ-noyau temps réel

Utiliser un micronoyau temps réel cohabitant avec le noyau du système d'exploitation.

Ce micronoyau exécute les tâches temps réel.

Le noyau du système d'exploitation est considéré par ce micronoyau comme la tâche de plus faible priorité. Elle n'est exécutée que lorsqu'aucune autre tâche temps réel n'est prête.

$\Rightarrow$ exemple RTLINUX.

Caractéristiques

- Mono-utilisateur ou multi-utilisateur
- Multitâches ou mono-tâche
- Distribué ou non
- Encombrement mémoire (mémoire limitée, pas de disque en général)
- Consommation d'énergie (batterie : point faible des SE)
- Communication (attention : la communication affecte la batterie)
- Contraintes de temps réel
- Contraintes de sécurité
- Coût de produits en relation avec le secteur cible

Processus

- Un processus comporte du code machine exécutable, une zone mémoire (données allouées par le processus), une pile ou *stack* (pour les variables locales des fonctions et la gestion des appels et retour des fonctions) et un tas ou heap pour les allocations dynamiques.
- Ce processus est une entité qui, de sa création à sa mort, est identifié par une valeur numérique : le PID (Process IDentifier).
- Tous les processus sont donc associés à une entrée dans la table des processus qui est interne au noyau.
- Chaque processus a un utilisateur propriétaire, qui est utilisé par le système pour déterminer ses permissions d'accès aux fichiers.
- ***Remarques** : Les commandes **ps** et **top** listent les processus sous UNIX/Linux et, sous Windows on utilisera le gestionnaire de tâches (**taskmgr.exe**).*

Processus : Contexte d'un processus

- Le contexte d'un processus (Process Control Block) est l'ensemble de :
  1. son état
  2. son mot d'état : en particulier la valeur des registres actifs et le compteur ordinal
  3. les valeurs des variables globales statiques ou dynamiques
  4. son entrée dans la table des processus
  5. les données privées du processus
  6. Les piles user et system
  7. les zones de code et de données.
- L'exécution d'un processus se fait dans son contexte.Quand il y a changement de processus courant, il y a **une commutation ou changement de contexte**. Le noyau s'exécute alors dans le nouveau contexte.
- En raison de ce contexte, on parle de processus lourd, en opposition aux processus légers que sont les threads.

Processus :Généalogie des processus

- La création d'un processus étant réalisée par **un appel système** (**fork** sous UNIX/Linux). Chaque processus est identifié par un numéro unique, le **PID** (Processus IDentification).
- Un processus est donc forcément créé par un autre processus (notion **père-fils**).Le **PPID** (Parent PID) d'un processus correspond au PID du processus qui l'a créé (son père)
- Exemple sous UNIX/Linux :

Processus :Généalogie des processus

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
top - 22:41:27 up  2:45,  2 users,  load average: 1,08, 0,54, 0,35
Tâches: 198 total,   1 en cours, 197 en veille,   0 arrêté,   0 zombie
%Cpu(s):  4,4 ut,  1,4 sy,  0,0 ni, 83,1 id, 11,1 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   1871916 total,  1736176 used,   135740 free,    81284 buffers
KiB Swap:  4111352 total,   235988 used,  3875364 free.   584484 cached Mem

  PID UTIL.     PR  NI    VIRT    RES    SHR S  %CPU %MEM    TEMPS+ COM.
 2369 ghouti    20   0 1896728 227044  37588 S   8,3 12,1   4:46.71 cinnamon
 2644 ghouti    20   0 3212680 144380  63396 S   8,3  7,7   4:33.54 chromium-b+
 2725 ghouti    20   0  929588 111012  59936 S   6,0  5,9   3:37.89 chromium-b+
 1464 root      20   0  652832 107940  90204 S   5,3  5,8   2:48.81 Xorg
 5294 ghouti    20   0  476356  26500  21384 S   1,0  1,4   0:00.19 gnome-scre+
    7 root      20   0       0      0      0 S   0,3  0,0   0:05.09 rcu_sched
 2150 ghouti    20   0  124924   3248   3092 S   0,3  0,2   0:02.14 at-spi2-re+
 4703 ghouti    20   0 1517724 182156  84680 S   0,3  9,7   0:34.37 chromium-b+
 5100 ghouti    20   0  626936  31116  25084 S   0,3  1,7   0:00.51 gnome-term+
 5124 ghouti    20   0   25084   3028   2468 R   0,3  0,2   0:00.48 top
    1 root      20   0   33900   3112   2092 S   0,0  0,2   0:00.84 init
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0,0  0,0   0:00.06 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0,0  0,0   0:00.00 kworker/0:+
    8 root      20   0       0      0      0 S   0,0  0,0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S   0,0  0,0   0:00.00 migration/0
   10 root      rt   0       0      0      0 S   0,0  0,0   0:00.02 watchdog/0
```

Figure – Exemple de la commande top sous linux

Processus :Généalogie des processus

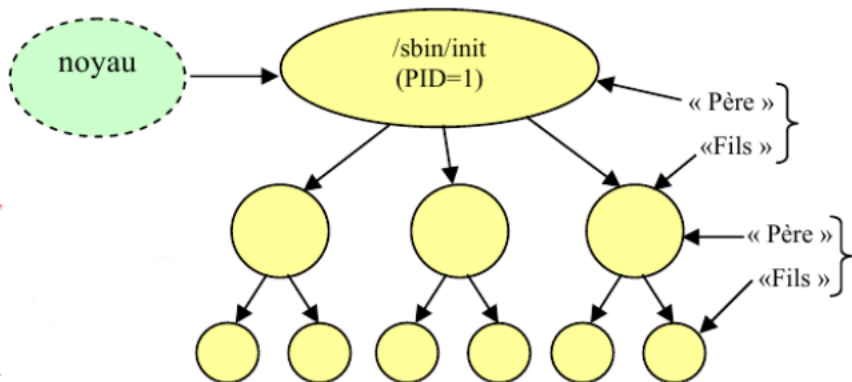


Figure – Arbre genéaologique des processus

Processus :Opérations réalisables sur un processus

1. Création : **fork**
2. Exécution : **exec**
3. Destruction :
   - terminaison normale
   - auto destruction exit
   - meurtre kill, **^C**
4. Mise en attente/réveil : **sleep, wait**
5. Suspension/reprise : **^Z/fg, bg**
6. Changement de priorité : **nice**

Processus :Création dynamique de processus

- Lors de l'initialisation du système (boot), le noyau crée plusieurs processus spontanés dans l'espace de l'utilisateur. Ces processus sont dits spontanés parce qu'ils ne sont pas créés par le mécanisme fork traditionnel.
- Le nom et la nature des processus spontanés varient d'un système à l'autre. init est le seul processus à part entière. Le PID d'init vaut toujours 1 c'est l'ancêtre de tous les processus utilisateurs et de presque tous les processus système.

Processus :Exécution d'un nouveau code

- Lors d'une opération **fork**, le noyau Unix crée un nouveau processus qui est une copie conforme du processus père. Le code, les données et la pile sont copiés et tous les fichiers ouverts par le père sont ainsi hérités par le processus fils.
- Si l'on désire exécuter du code à l'intérieur de ce nouveau processus, on utilisera un appel de type **exec : execl, execlp, execle, execv ou execvp.**
- La famille de fonctions **exec** remplace l'image mémoire du processus en cours par un nouveau processus.

Processus :Exemple programme fork

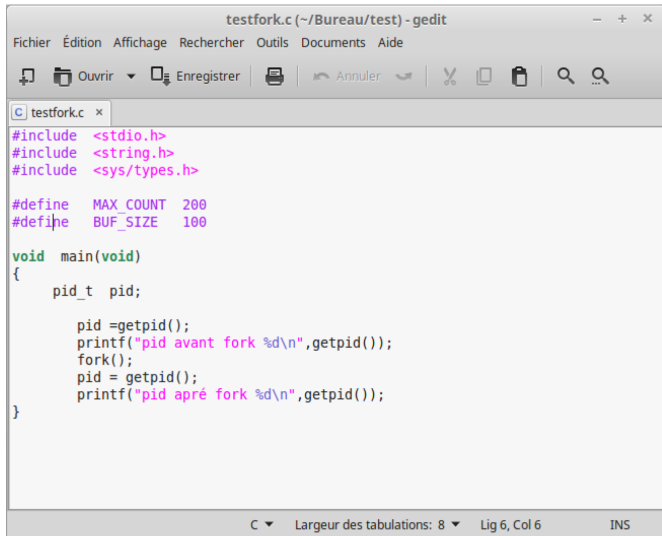


Figure – Exemple d'execution du fork

Processus :Exemple programme fork

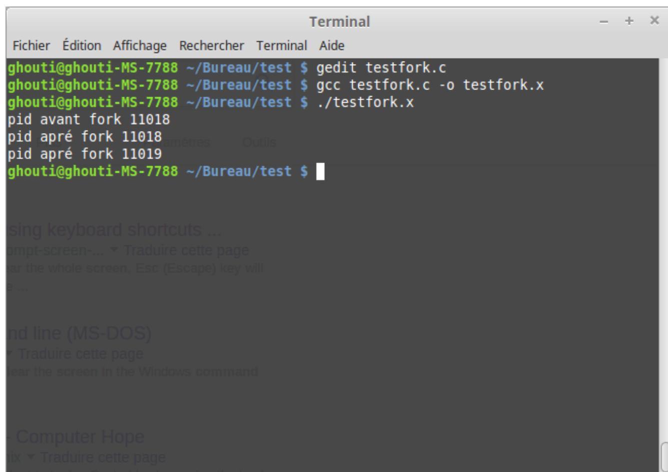


Figure – Exemple d'execution du fork

Processus :Exemple programme fork

testfork2.c (~/Bureau/test) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

testfork2.c

```c
#include  <stdio.h>
#include  <sys/types.h>

#define   MAX_COUNT  10

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
     pid_t  pid;

        pid=getpid();
        printf(" le PID père est: %d\n",pid);
     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}

void  ChildProcess(void)
{       pid_t  pid;
        pid=getpid();
     printf(" je suis le fils mon PID est : %d \n",pid);
}

void  ParentProcess(void)
{       pid_t  pid;
        pid=getpid();
        printf("je suis le papa mon PID est : %d \n",pid);
}
```

C   Largeur des tabulations: 8   Lig 31, Col 33   INS

Figure – Exemple d'execution du fork

Processus :Exemple programme fork

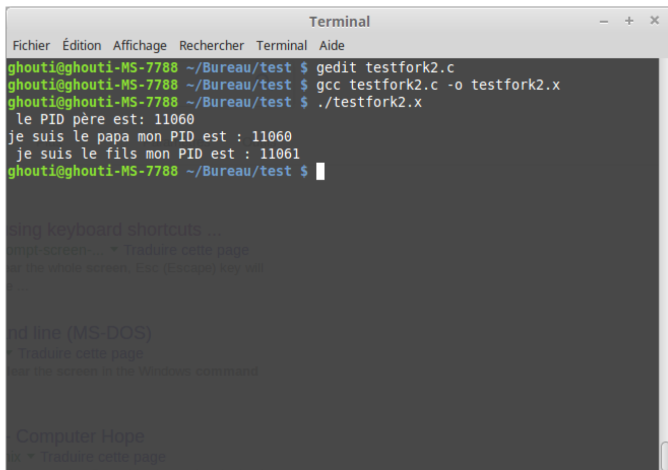
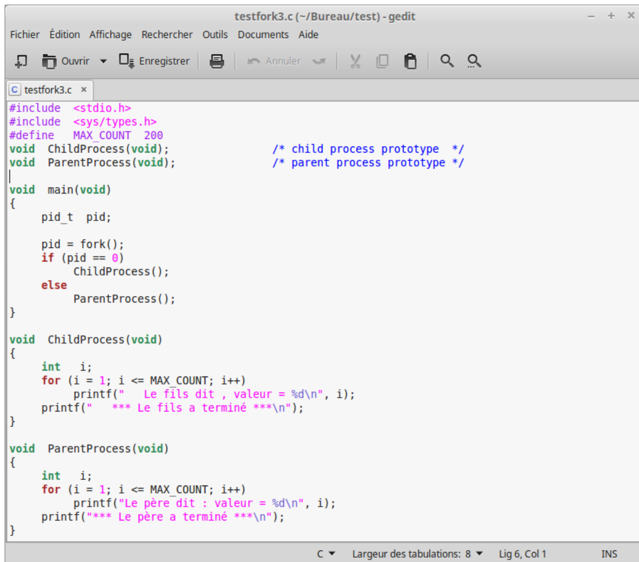


Figure – Exemple d'execution du fork

Processus :Exemple programme fork




```c
#include  <stdio.h>
#include  <sys/types.h>
#define   MAX_COUNT  200
void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}

void  ChildProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("   Le fils dit , valeur = %d\n", i);
     printf("   *** Le fils a terminé ***\n");
}

void  ParentProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("Le père dit : valeur = %d\n", i);
     printf("*** Le père a terminé ***\n");
}
```

Figure – Exemple d'execution du fork

Processus :Exemple programme fork

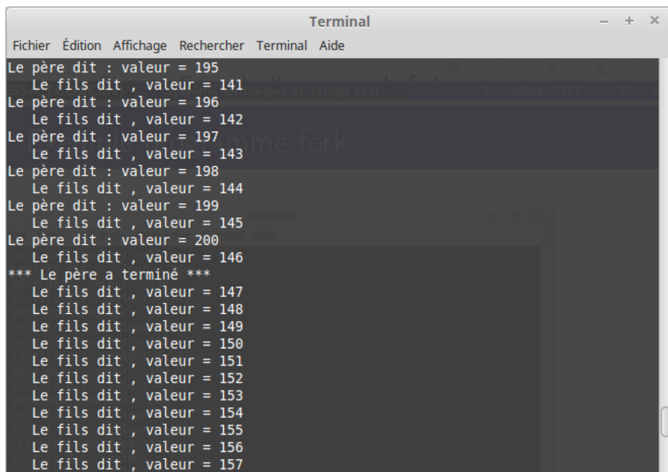


Figure – Exemple d'execution du fork

Processus :Synchronisation des terminaisons

- Rappel : les processus créés par des **fork** s'exécutent de façon concurrente avec leur père. On ne peut présumer l'ordre d'exécution de ces processus (cf. politique de l'ordonnanceur).
- Il sera donc impossible de déterminer quels processus se termineront avant tels autres (y compris leur père). D'où l'existence, dans certains cas, d'un problème de synchronisation.
- La primitive **wait** permet l'élimination de ce problème en provoquant la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.

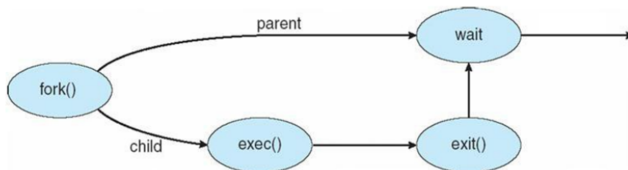


Figure – Synchronisation processus père/fils

Processus :Exemple programme fork et wait

testfork3.c (~/Bureau/test) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

testfork3.c

```c
#include  <stdio.h>
#include  <sys/types.h>
#define   MAX_COUNT  200
void  ChildProcess(void);                 /* child process prototype  */
void  ParentProcess(void);                /* parent process prototype */
pid_t child_pid;
void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0) {
        child_pid=getpid();
          ChildProcess();}
     else
          ParentProcess();
}

void  ChildProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("   Le fils dit , valeur = %d\n", i);
     printf("   *** Le fils a terminé ***\n");
}

void  ParentProcess(void)
{
     int   i;
     for (i = 1; i <= MAX_COUNT; i++)
          printf("Le père dit : valeur = %d\n", i);
        wait(NULL);//<-------------------------------- suspendre l'execution du processus père
     printf("*** Le père a terminé ***\n");
}
```

C   Largeur des tabulations: 8   Lig 32, Col 31   INS

Processus :Exemple programme fork et wait

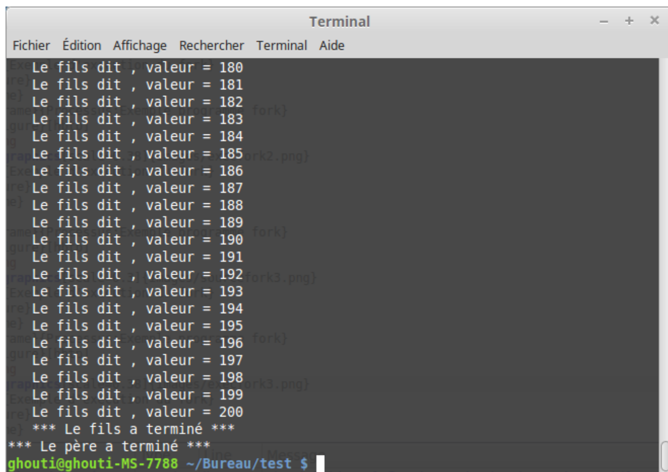


Figure – Exemple d'execution du fork

Processus : Exercice

Donner un programme qui affiche, dans l'ordre, les entiers de 0 à 3 par 4 processus

Processus :États d'un processus

- Le processus est une activité dynamique et il possède un **état** qui évolue au cours du temps. Ce processus transitera par différents états selon que :
  - il s'exécute (ACTIF ou Élu)
  - il attend que le noyau lui alloue le processeur (PRET)
  - il attend qu'un événement se produise (ATTENTE ou Bloqué)
- C'est l'**ordonnanceur** (*scheduler*) qui contrôle l'exécution et les états des processus.

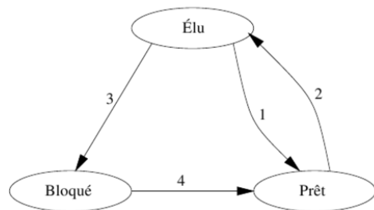


Figure – États d'un processus

Ordonnancement :Problèmatique

- De nombreux processus sont gérés par le SE
- Dans la plupart des cas la machine possède 1 seul processeur (soit quelques coeurs) par conséquent : un grand nombre de processus se partagent le processeur (une ressource limitée).
- Il faut définir :
  - un mécanisme qui permet au SE de garder le contrôle des exécutions
  - une politique d'accès au processeur (ordonnancement ou scheduling) : l'utilisation du processeur est divisée en tranches (slices) affectées au processus sous la forme de quantum.

L'ordonnanceur

L'Ordonnanceur (planificateur, scheduler) est la partie (un programme) du système d'exploitation responsable de régler les états des processus (Prêt, Actif,...etc.) et de gérer les transitions entre ces états ; c'est l'allocateur du processeur aux différent processus, il alloue le processeur au processus en tête de file des Prêts

Les objectifs d'un Ordonnanceur sont :

- Maximiser l'utilisation du processeur
- Présenter un temps de réponse acceptable
- Respecter l'équité entre les processus selon le critère d'ordonnancement utilisé.

Ordonnancement :Les composants de l'ordonnanceur

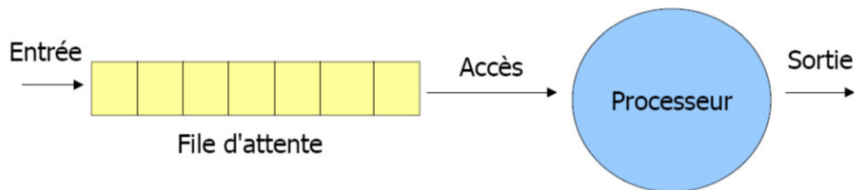


Figure – Les composants de l'ordonnanceur

On distingue 3 grands principes de gestion des accès au processeur selon :

- l'ordre d'arrivée : premier arrivé premier servi ;
- le degré d'urgence : le premier servi est celui dont le besoin d'accès rapide à la ressource est le plus grand ;
- l'importance : le premier servi est celui dont l'accès à la ressource est le plus important.

Ordonnacement :Notion de pénalisation

Pénalisation

Lorsqu'un processus ne peut pas accéder directement à une ressource qu'il convoite on dit qu'il est pénalisé

Temps d'attente

Le temps d'attente est le nombre d'unité de temps durant lesquelles le processus est présent dans la file d'attente (sans être exécuté)

Ordonnacement :Notion de pénalisation

Le taux de retard

$$T = \frac{d}{a+d}$$

- d est durée du processus
- a durée d'attente (cumulée)
- a + d le temps total du processus passé dans le système

dans le cas idéal T = 1

Ordonnacement :Traitement jusqu'à terminaison

Principe : la politique de traitement du processus jusqu'à terminaison

- accorde le processeur à un processus.
- ne l'interrompt jamais quelque soit sa durée ;

Ordonnacement :Traitement jusqu'à terminaison

Stratégie **FCFS** (**First Come First Serve**) c-à-d premier arrivé premier servi

- elle correspond à une gestion FIFO (First In, First Out) de la file d'attente
- chaque processus s'exécute jusqu'à son terme ;
- le processus élu est celui qui est en tête de liste des Prêt
- Cet algorithme est facile à implanter, mais il est loin d'optimiser le temps de traitement moyen

Stratégie **FCFS** (**First Come First Serve**) c-à-d premier arrivé premier servi

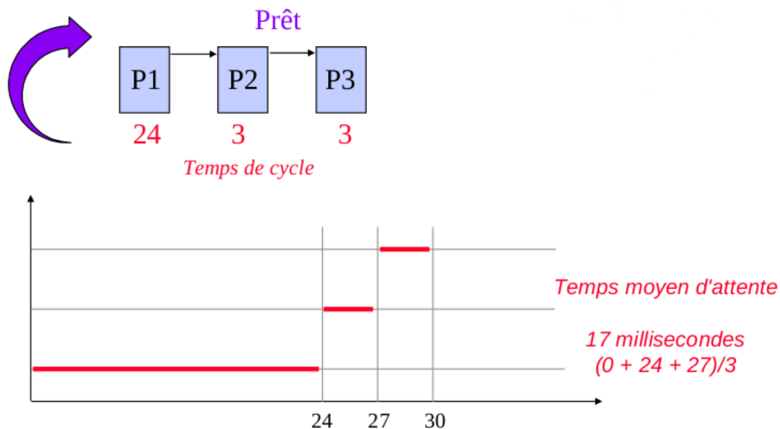
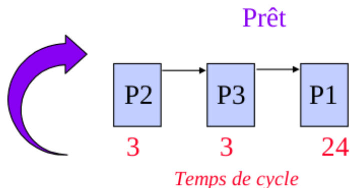


Figure – Exemple de la stratégie FCFS

Stratégie **FCFS** (**First Come First Serve**) c-à-d premier arrivé premier servi



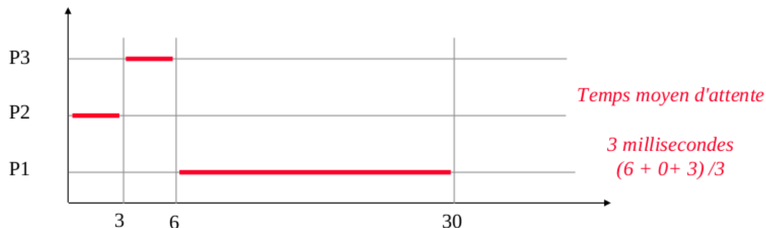


Figure – Exemple de la stratégie FCFS

Ordonnacement :Traitement jusqu'à terminaison

Stratégie **SJF** (Shorstest Job First) c-à-d moindre durée :

- on garde le principe d'occupation du processeur jusqu'à terminaison
- la file d'attente est ordonnée non plus de façon chronologique mais en fonction du temps déxécution nécessaire (on fait passer en tête les travaux courts)
- cependant la durée totale est a-priori inconnue, on utiliser une décomposition en rafales (burst) CPU et IO (nombre d'instructions)
- prouvé comme étant optimal vis à vis de temps d'attente moyen

Ordonnacement :Traitement jusqu'à terminaison

Ordonnancement basé sur une priorité

- Une priorité est associé à chaque processus
- Le CPU est alloué à celui qui a la priorité la plus haute
- En cas de priorité égale on utilise FCFS
- Le SFJ est un cas simple de stratégie utilisant une prioritée (inverse de t)
- D'une manière générales les algorithmes utilisant un priorité peuvent provoquer une attente infinie ou une famine
- Une solution est de corriger la priorité avec l'age du processus

La famine

On dit que l'on a une situation de famine (en anglais starvation), lorsque un ou plusieurs processus n'obtiennent pas les ressources dont ils ont besoin par suite du comportement de l'ensemble des processus, sans être en situation d'interblocage.

Ordonnacement :Traitement avec préemption

La préemption concerne la gestion du processeur.

- Elle consiste à décider en fonction de certains critères, de remettre le processus en file d'attente avant la fin de son exécution.
- Les processus font plusieurs passages dans la file d'attente
- Le temps d'attente d'un processus est sa durée d'attente cumulée
- Ce mécanisme est surtout utilisé dans la stratégie du **tourniquet**
- La stratégie SJF peut être adaptée pour un mode préemptif : si un processus plus court que le processus actif arrive dans la file, le processus actif est préempté

Stratégie du tourniquet RR (Round-Robin)

- objectif : vider les processus qui s'attardent trop dans le processeur
- l'ordonnanceur réalise la commutation de contexte
- un processus est remis en file d'attente dès que sa durée d'occupation du processeur dépasse une durée prédéfinie : quantum de temps
- la gestion de la file d'attente est faite selon le principe FIFO :
  - les travaux assez courts sont vite servis
  - les travaux longs sortiront du système au bout d'un temps fini

Stratégie du tourniquet RR (Round-Robin)

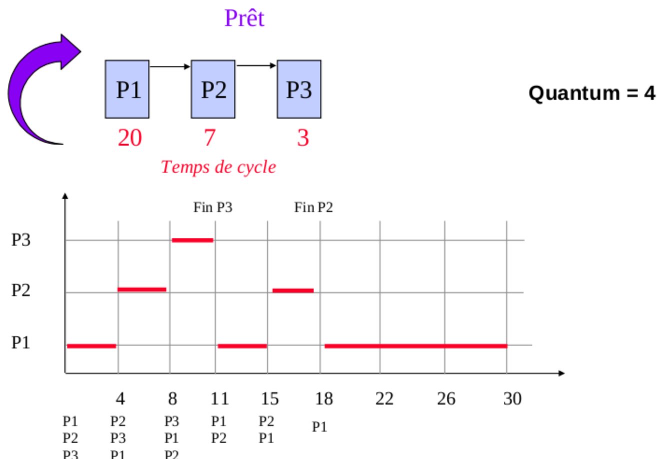


Figure – Exemple de l'algorithme Round-Robin

Ordonnacement :Performance des algorithmes d'Ordonnancement

- Temps de rotation = Temps fin d'exécution − Temps d'arrivée
- Temps d'attente = Temps de rotation − Durée d'exécution
- Temps moyen d'attente = $\frac{\Sigma \text{Temps attente}}{\text{nombre de processus}}$
- Rendement = $\frac{\Sigma \text{Temps d'execution}}{\text{nombre de processus}}$

La notion de processus concurrents

Définition

Ressource La notion de ressource dans les SE désigne tout élément qui est utile au déroulement d'un processus.

Une ressource peut être :

- Physique (si on se place du point de vue du système d'exploitation) : processeur, mémoire, périphérique
- Logique (si on se place du point de vue du programmeur) : fichier, variable, objet

La notion de processus concurrents

Plusieurs processus peuvent avoir besoin des mêmes ressources.

Définition

Ressource une ressource est dite partageable si elle peut être utilisée en même temps à plusieurs processus. Une ressource est dite non partageable si elle doit être à utilisée exclusivement par un seul processus.

Dans ce cas on dit que les processus doivent être exclusion mutuelle pour l'accès à la ressource.

La notion verrouillage

Lorsqu'une ressource est partagée, si plusieurs processus accèdent à la ressource, des incohérences peuvent en résulter : il faut un arbitre

Exemple

des guichets automatiques de retrait d'argent : simulation du comportement

Comment éviter les situation de compétition sur des ressources non partageable ?

- interdire l'accès partagé à la ressource :exclusion mutuelle
- interdire l'exécution simultanée du codé accédant à la ressource : section critique

La notion verrouillage

Pour résoudre ce problème on utilise la notion de verrou (lock)

Deux opérations sont proposées :

- Verouiller(v) permet à un processus d'acquérir un verrou,s'il n'est pas disponible, le processus est bloqué en attente du verrou
- Déverouiller(v) permet de libérer un verrou.

Sémaphores

P()

P(){
Si S $>$ 0 **alors** S = S -1
sinon
    ajouter tâche à la file d'attente
}

V()

V(){
S = S + 1
Si file d'attente n'est pas vide **alors**
    transfert de contrôle a une autre tache dans la fille t'attente(réveil)
}

Sémaphore binaire

- Un sémaphore binaire est un sémaphore qui est initialisé avec la valeur 1.
- Ceci a pour effet de contrôler l'accès une ressource unique.
- Le sémaphore binaire permet l'exclusion mutuelle (mutex).

Sémaphores
Accès à une section critique

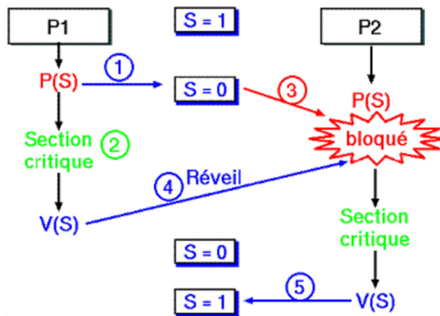


Figure – Principe des sémaphores

Interblocage

Processus 1

Début

P(S1)
P(S2)

Section Critique

V(S2)
V(S1)

Fin

Processus 2

Début

P(S2)
P(S1)

Section Critique

V(S1)
V(S2)

Fin

Figure – Cas d'interblocage entre deux processus

Interblocage

Un interblocage est possible si :

- Le processus 1 obtient S1.
- Le processus 2 obtient S2.
- Le processus 1 attend pour obtenir S2 (qui est entre les mains du processus 2).
- Le processus 2 attend pour obtenir S1 (qui est entre les mains du processus 1).

Dans cette situation, les deux tâches sont définitivement bloquées.

Prévention : Une méthode consiste à toujours acquérir les mutex dans le même ordre.

Sémaphore bloquant

- Un sémaphore **bloquant** est un sémaphore qui est initialisé avec la valeur **0**
- Ceci a pour effet de bloquer n'importe quel thread qui effectue P(S) tant qu'un autre thread n'aura pas fait un V(S). Ce type d'utilisation est très utile lorsque l'on a besoin de contrôler l'ordre d'exécution entre threads.

Sémaphores
Producteur/Consommateur

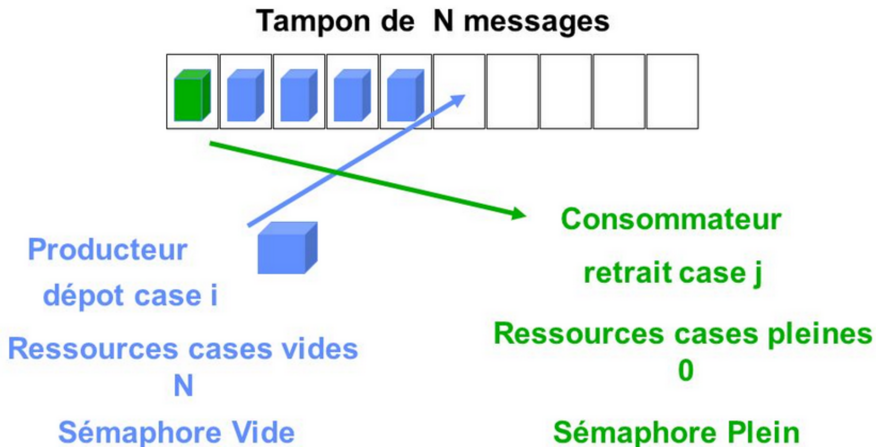


Figure – Problème producteur/consommateur

Sémaphores
Producteur/Consommateur

plein = 0
vide = N

```
Producteur
  Début
    Tant que (Vrai)
      Produire_objet()
      P(vide)
      Section Critique
      V(plein)
    Fin Tant que
  Fin

Consommateur
  Début
    Tant que (Vrai)
      P(plein)
      Section Critique
      V(vide)
      Consommer_objet()
    Fin Tant que
  Fin
```

Figure – Problème producteur/consommateur

Lecteurs/Rédacteurs

Sémaphores et Le langage C

#include <semaphore.h>

- int **sem_init(sem_t** *semaphore, int pshared, unsigned int valeur);
  Création d'un sémaphore et préparation d'une valeur initiale.
- int **sem_wait(sem_t** * semaphore);
  Opération P sur un sémaphore.
- int **sem_trywait(sem_t** * semaphore);
  Version non bloquante de l'opération P sur un sémaphore.
- int **sem_post(sem_t** * semaphore);
  Opération V sur un sémaphore.
- int **sem_getvalue(sem_t** * semaphore, int * sval);
  Récupérer le compteur d'un sémaphore.
- int **sem_destroy(sem_t** * semaphore);
  Destruction d'un sémaphore.

Sémaphore : exemple

semaphore.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

semaphore.c

```
void* affichage (void* name) {
int i, j;
for(i = 0; i < 20; i++) {
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
printf("\n ");
}
return NULL;
}

int main (void) {
pthread_t filsA, filsB;

if (pthread_create(&filsA, NULL, affichage, "AA")) {
perror("pthread_create");
exit(EXIT_FAILURE);
}
if (pthread_create(&filsB, NULL, affichage, "BB")) {
perror("pthread_create");
exit(EXIT_FAILURE);
}
if (pthread_join(filsA, NULL))
perror("pthread_join");
if (pthread_join(filsB, NULL))
perror("pthread_join");
printf("Fin du pere\n");
```

Sémaphore : exemple

Terminal

Fichier Édition Affichage Rechercher Terminal Aide

```
AA AA AA AA AA AA AA BB BB BB BB BB BB BB AA BB BB AA BB AA
 BB BB BB BB BB BB BB BB BB BB
 AA AA AA AA AA AA AA AA AA AA

 BB BB BB BB AA BB AA AA BB AA BB BB AA BB AA BB AA
 BB BB BB BB AA BB AA BB AA BB
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA BB BB BB
 BB BB BB BB AA BB AA BB AA BB BB
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA BB
 BB BB BB BB BB BB BB BB AA BB AA BB AA
 BB BB BB BB BB BB BB BB AA BB AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA BB AA
 BB BB BB BB BB BB BB BB AA BB
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA BB AA
 BB BB BB BB BB BB BB BB AA BB AA AA BB AA
 BB BB BB BB BB BB BB BB
 AA AA AA AA AA AA AA BB AA BB AA
 BB BB BB BB BB BB BB BB AA BB
 AA AA AA AA AA AA AA AA BB AA
 BB BB BB BB BB BB BB BB AA BB
 AA AA AA AA AA AA AA AA AA BB AA
 BB BB BB BB BB BB BB
 AA AA AA AA AA AA AA AA BB BB BB
 BB BB BB BB BB BB BB BB AA BB AA BB
 AA AA AA AA AA AA AA AA
 BB BB BB BB BB BB BB BB AA AA
 AA AA AA AA AA AA AA AA AA BB AA BB
 AA AA AA AA AA AA AA AA
 BB BB BB BB BB BB BB BB AA BB AA BB
```

Sémaphore : exemple

semaphore.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

```c
sem_t mutex;//<----------------------------Sémaphore mutex
void* affichage (void* name) {
int i, j;
for(i = 0; i < 20; i++) {
        sem_wait(&mutex); //<--------------P(mutex)
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
        for(j=0; j<5; j++)
        printf("%s ",(char*)name);
printf("\n ");
sem_post(&mutex);//<----------------------V(mutex)
}
return NULL;
}
int main (void) {
pthread_t filsA, filsB;
sem_init(&mutex, 0, 1);//<----------------init(s,1)
if (pthread_create(&filsA, NULL, affichage, "AA")) {
perror("pthread_create");
exit(EXIT_FAILURE);
}
```

C Largeur des tabulations : 8 Lig 6, Col 60 INS

Figure – Exemple de deux threads avec sémaphore

Sémaphore : exemple

Terminal

Fichier Édition Affichage Rechercher Terminal Aide

```
AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 AA AA AA AA AA AA AA AA AA AA
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
 BB BB BB BB BB BB BB BB BB BB
```

Moniteurs

- Les moniteurs proposent une solution de "haut-niveau" pour la protection de données partagées (Hoare 1974)
- Ils simplifient la mise en place de sections critiques
- Ils sont définis par :
  1. des données internes (appelées aussi variables d'état)
  2. des primitives d'accès aux moniteurs (points d'entrée)
  3. des primitives internes (uniquement accessibles depuis l'intérieur du moniteur)
  4. une ou plusieurs files d'attentes

Structure d'un d'un moniteur

Type m = **moniteur**
Début
 Déclaration des variables locales (ressources partagées) ;
 Déclaration et corps des procédures du moniteur (points d'entrée) ;
 Initialisation des variables locales ;
Fin

Principe de fonctionnement

- Seul un processus (ou tâche ou thread) peut être actif à un moment donné à l'intérieur du moniteur
- La demande d'entrée dans un moniteur (ou d'exécution d'une primitive du moniteur) sera bloquante tant qu'il y aura un processus actif à l'intérieur du moniteur

Remarque : L'accès à un moniteur construit donc implicitement une exclusion mutuelle

Les Moniteurs

- Lorsqu'un processus actif au sein d'un moniteur ne peut progresser dans son travail (une certaine condition est fausse), il libère l'accès au moniteur avant de se bloquer.
- Lorsque des variables internes du moniteur ont changé, le moniteur doit pouvoir « réveiller » un processus bloqué.
- Pour cela, il existe deux types de primitives :
  1. **wait** : qui libère l'accès au moniteur, puis bloque le processus appelant sur une condition
  2. **signal** : qui réveille sur une condition un des processus en attente à l'intérieur du moniteur (un processus qui a exécuté précédemment un **wait** sur la même condition)

Les variables condition

- Une variable condition : est une variable
  - qui est définie à l'aide du type condition ;
  - qui a un identificateur mais,
  - qui n'a pas de valeur (contrairement à un sémaphore).
- Une condition :
  - ne doit pas être initialisée
  - ne peut être manipulée que par les primitives Wait et Signal.
  - est représentée par une file d'attente de processus bloqués sur la même cause ;
  - est donc assimilée à sa file d'attente.

Les primitives Wait et Signal

- La primitive **Wait** bloque systématiquement le processus qui l'exécute
- La primitive **Signal** réveille un processus de la file d'attente de la condition spécifiée, si cette file d'attente n'est pas vide ; sinon elle ne fait absolument rien.

Syntaxe

1. Syntaxe :
   cond.**Wait** ;
   cond.**Signal** ;
   /* cond est la variable de type condition déclarée comme variable locale */
2. Autre syntaxe :
   Wait(*cond*) ;
   Signal(*cond*) ;

Un processus réveillé par *Signal* continue son exécution à l'instruction qui suit le *Wait* qui l'a bloqué.

Exemple RDV entre N processus à l'aide d'un moniteur

```
Type Rendez_vous = moniteur
        {variables locales }
        Var Nb_arrivés : entier ; Tous_Arrivés : condition   ;
        {procédure accessible aux programmes utilisateurs }
        Procedure Entry Arriver ;
        Début
                Nb_arrivés = Nb_arrivés + 1 ;
                Si Nb_arrivés < N Alors Tous_Arrivés.Wait ;
                Tous_Arrivés.Signal ;
        Fin
        Début   {Initialisations }
                Nb_arrivés = 0;
        Fin.
```

Figure – Exemple de RDV entre N processus

Exemple RDV entre N processus à l'aide d'un moniteur

```
Type Rendez_vous = moniteur
        {variables locales }
        Var Nb_arrivés : entier ; Tous_Arrivés : condition ;
        {procédure accessible aux programmes utilisateurs }
        Procedure Entry Arriver ;
        Début
                Nb_arrivés = Nb_arrivés + 1 ;
                Si Nb_arrivés < N Alors Tous_Arrivés.Wait ;
                Tous_Arrivés.Signal ;
        Fin
        Début   {Initialisations }
                Nb_arrivés = 0;
        Fin.
```

Figure – Exemple de RDV entre N processus

Exemple RDV entre N processus à l'aide d'un moniteur Les programmes des processus s'écrivent alors :

Processus Pi

....

Rendez_vous.Arriver ; {*Point de rendez-vous : sera bloquant si au moins un processus n'est pas arrivé au point de rendez-vous*}

....

Producteur-Consommateur à l'aide des moniteurs

```
Type ProducteurConsomateur = Moniteur
{variables locales}
Var Compte : entier ; Plein, Vide : condition ;
{procédures accessibles aux programmes utilisateurs }
Procedure Entry Déposer(message M) ;
Début
      si Compte=N alors Plein.Wait ;
      dépôt(M) ;
      Compte=Compte+1 ;
      si Compte==1 alors Vide.Signal ;
Fin
```

Producteur-Consommateur à l'aide des moniteurs(Suite)

Procedure Entry Retirer(message M) ;
Début
    si Compte=0 alors Vide.Wait ;
    retrait(M) ;
    Compte=Compte-1 ;
    si Compte==N-1 alors Plein.Signal ;
Fin
Début {*Initialisations*}Compte= 0 ; **Fin.**

Les Moniteurs
Producteur-Consommateur à l'aide des moniteurs(Suite)

- **Processus Producteur**

  message M ;

  Début

  tant que vrai faire

  Produire(M) ;

  ProducteurConsommateur.déposer(M) ;

  Fin

- **Processus Consommateur**

  message M ;

  Début

  tant que vrai faire

  ProducteurConsommateur.retirer(M) ;

  Consommer(M) ;

  Fin

Les moniteurs en langage C

- d'un mutex (type **pthread_mutex_t**) qui sert à protéger la partie de code où l'on teste les conditions de progression
- et d'une variable condition (type **pthread_cond_t**) qui sert de point de signalisation :
- on se met en attente sur cette variable par la primitive : Libération du mutex + Blocage systématique de l'appelant de manière atomique **pthread_cond_wait**(&*laVariableCondition*,&*leMutex*) ;
- on est réveillé sur cette variable avec la primitive : Réveil d'une thread en attente qui acquiert à nouveau le mutex **pthread_cond_signal**(&*laVariableCondition*) ;

Les moniteurs en langage C

- Un mutex peut être verrouillé par la primitive
  int pthread_mutex_lock(pthread_mutex_t *mutex);
- Un mutex peut être déverrouillé par la primitive
  int pthread_mutex_unlock(pthread_mutex_t *mutex);

Les Moniteurs

moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

moniteur.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

/* définition du tampon */
#define N 10 /* Nb de cases du tampon */
#define NbMess 20 /* Nb de messages échangés */

int NbPleins=0;
int tete=0, queue=0;
int tampon[N];

/* définition des conditions et du mutex */
pthread_cond_t vide;
pthread_cond_t plein;
pthread_mutex_t mutex;
pthread_t producteur;
pthread_t consomateur;
```

C Largeur des tabulations : 8 Lig 5, Col 1 INS

Les Moniteurs

*moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

*moniteur.c

```c
void Deposer(int m){
pthread_mutex_lock(&mutex);
        if(NbPleins == N) pthread_cond_wait(&plein, &mutex);
        tampon[queue]=m;
        queue=(queue+1)%N;
        NbPleins++;
pthread_cond_signal(&vide);
pthread_mutex_unlock(&mutex);
}
int Prelever(void){
int m;
pthread_mutex_lock(&mutex);
if(NbPleins ==0) pthread_cond_wait(&vide, &mutex);
m=tampon[tete];
tete=(tete+1)%N;
NbPleins--;
pthread_cond_signal(&plein);
pthread_mutex_unlock(&mutex);
return m;
}
```

C Largeur des tabulations : 8 Lig 29, Col 1 INS

Les Moniteurs

*moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

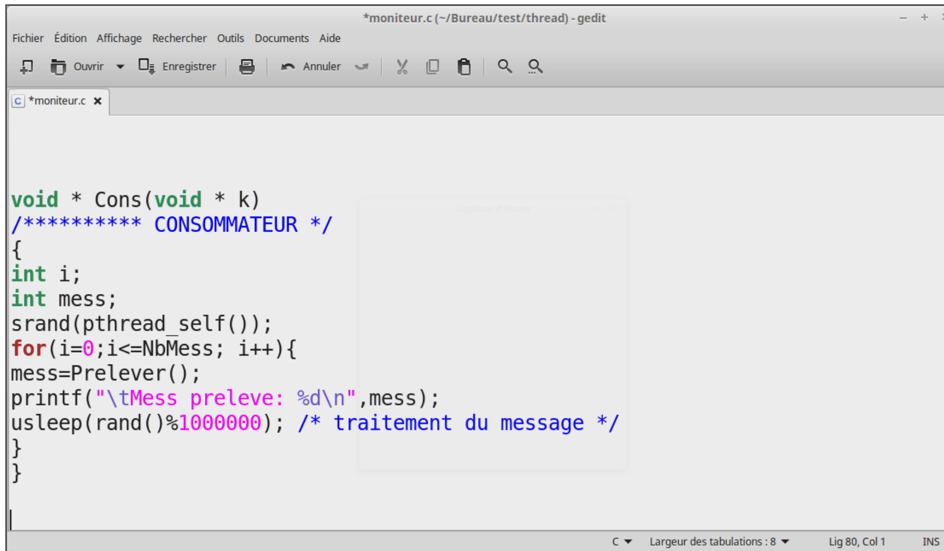
Ouvrir Enregistrer Annuler

*moniteur.c

```
void * Prod(void * k)
/********** PRODUCTEUR */
{
int i;
int mess;
srand(pthread_self());
for(i=0;i<=NbMess; i++){
usleep(rand()%10000); /* fabrication du message */
mess=rand()%1000;
Deposer(mess);
printf("Mess depose: %d\n",mess);
}
}
```

C Largeur des tabulations : 8 Lig 66, Col 1 INS

Les Moniteurs



Les Moniteurs

*moniteur.c (~/Bureau/test/thread) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

*moniteur.c

```c
void main()
/* M A I N */
{
int i, num;
pthread_mutex_init(&mutex,0);
pthread_cond_init(&vide,0);
pthread_cond_init(&plein,0);
/* creation des threads */
pthread_create(&producteur, 0, (void * (*)()) Prod, NULL);
pthread_create(&consomateur, 0, (void * (*)()) Cons, NULL);
// attente de la fin des threads
pthread_join(producteur,NULL);
pthread_join(consomateur,NULL);
// libération des ressources
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&vide);
pthread_cond_destroy(&plein);
exit(0);
}
```

C Largeur des tabulations : 8 Lig 80, Col 1 INS

Les interblocages

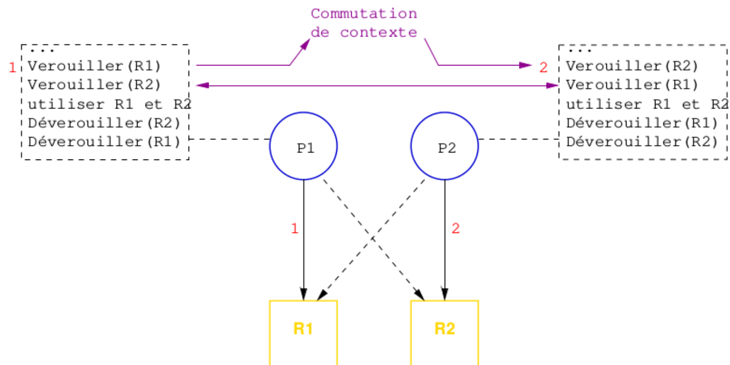


Figure – Exemple d'interblocage

Synchronisation et communication inter-tâches

Synchronisation

La synchronisation des processus est un problème plus large que celui de la concurrence :

- concerne les méthodes mises en œuvre afin que les processus coordonnent leurs activités ;
- attente de la réalisation d'un calcul (fork/join) ;
- utilisation de ressources (modèles Lecteurs/Rédacteurs et Producteurs/consommateurs, etc.) ;
- file d'attente de traitements ;
- barrière de synchronisation etc.

La communication inter-processus

Le terme IPC (Inter Process Communication) recouvre les modes de communication entre processus à l'intérieur du SE ou entre plusieurs SE sur des machines différentes :

- Communication par message (Message Passing)
- Mémoire partagé et buffering (Shared Memory)
- Appel de procédures / méthodes (Remote Procedure Call)
- Socket pour les systèmes distribués,communiquant via un réseau

Exercice

Créez un code qui crée deux threads : un qui incrémente une variable compteur par un nombre tiré au hasard entre 0 et 10, et l'autre qui affiche un message lorsque la variable compteur dépasse 20.

threadExemple.c (~/testCodes) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

Ouvrir Enregistrer Annuler

threadExemple.c

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

pthread_cond_t condition = PTHREAD_COND_INITIALIZER; /* Création de la condition */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* Création du mutex */


void* threadAlarme (void* arg);
void* threadCompteur (void* arg);


int main (void)

{

    pthread_t monThreadCompteur;

    pthread_t monThreadAlarme;


    pthread_create (&monThreadCompteur, NULL, threadCompteur, (void*)NULL);

    pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL); /* Création des
threads */


    pthread_join (monThreadCompteur, NULL);

    pthread_join (monThreadAlarme, NULL); /* Attente de la fin des threads */


    return 0;
```

C Largeur des tabulations: 8 Lig 47, Col 66 INS

Synchronisation et communication inter-tâches

threadExemple.c (~/testCodes) - gedit

Fichier Édition Affichage Rechercher Outils Documents Aide

```c
void* threadCompteur (void* arg)

{
    int compteur = 0, nombre = 0;

    srand(time(NULL));

    while(1) /* Boucle infinie */

    {
        nombre = rand()%10; /* On tire un nombre entre 0 et 10 */

        compteur += nombre; /* On ajoute ce nombre à la variable compteur */

        printf("\n%d", compteur);

        if(compteur >= 20) /* Si compteur est plus grand ou égal à 20 */

        {
            pthread_mutex_lock (&mutex); /* On verrouille le mutex */

            pthread_cond_signal (&condition); /* On délivre le signal : condition remplie */

            pthread_mutex_unlock (&mutex); /* On déverrouille le mutex */

            compteur = 0; /* On remet la variable compteur à 0 */

        }

        sleep (1); /* On laisse 1 seconde de repos */
    }
    pthread_exit(NULL); /* Fin du thread */
}
```

C Largeur des tabulations: 8 Lig 54, Col 1 INS

Synchronisation et communication inter-tâches

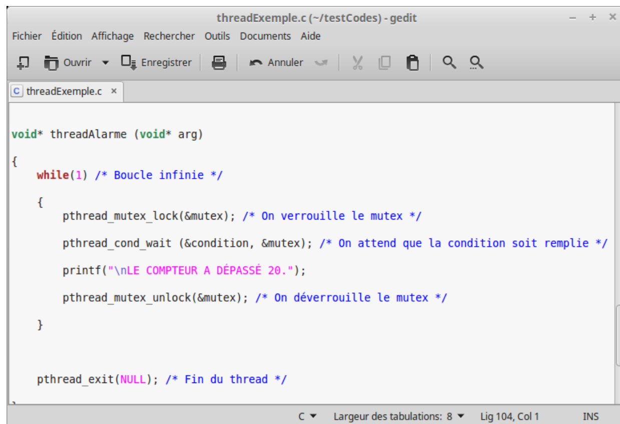


Figure – Exemple de la fonction threadAlamre

Synchronisation et communication inter-tâches

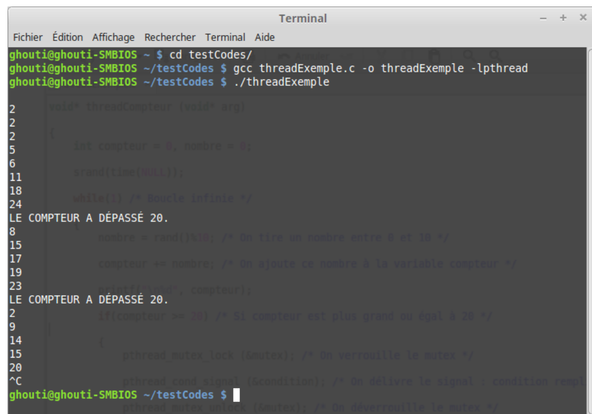


Figure – Exemple d'Exécution

Les Systèmes temps réel embarqué

Les OS classiques (temps partagé)

- Un système d'exploitation (OS) classique, c.-à-d. orienté temps partagé, doit organiser et optimiser l'utilisation des ressources de façon à ce que l'accès à ces ressources soit **équitable**.
  Un OS n'a donc pour seule contrainte de temps que celle d'un temps de réponse satisfaisant pour les utilisateurs avec lesquels il dialogue.
  Les traitements qu'on lui soumet sont effectués en parallèle au fil de l'allocation des quanta de temps aux diverses applications.
- Un OS est capable de **dater les événements** qui surviennent au cours du déroulement de l'application ! : mise à jour d'un fichier, envoi d'un message, etc. Il permet aussi de suspendre un traitement pendant un certain délai, d'en lancer un à une certaine date...
- En aucun cas, un OS classique ne garantit que les résultats seront obtenus pour une date précise, surtout si ces résultats sont le fruit d'un traitement déclenché par un événement **EXTERIEUR** (émission d'un signal par un périphérique quelconque,...)

Présentation

Définitions

- Temps réel signifie l'aptitude d'un système d'exploitation de fournir le niveau de service requis au bout d'un temps de réponse borné ».
- Un système temps réel est un système informatique qui doit répondre à des stimuli fournis par un environnement externe afin de le contrôler.
- Un STR est un système dont la correction dépend non seulement de la justesse des calculs mais aussi du temps auquel est fourni la réponse

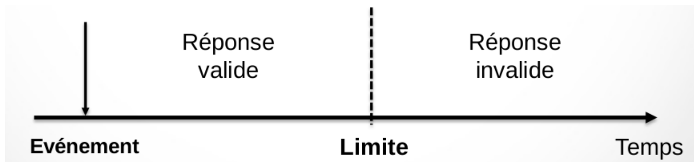


Figure – La correction d'un système temps réel

Domaines d'applications

- Installations industrielles (chimique, nucléaire, automobile, ...)
- Contrôle et régulation de trafic en milieu urbain
- Systèmes embarqués (voitures,(voitures trains,trains ...))
- Télécommunications
- Avionique et aérospatial
- Domaine militaire
- Multimédia et Web (téléconférences, télé-achat, ...)
- Domotique, Jeux
- Médecine, Télé-médecine
- Réalité virtuelle, Travail coopératif
- Autres

Exemple

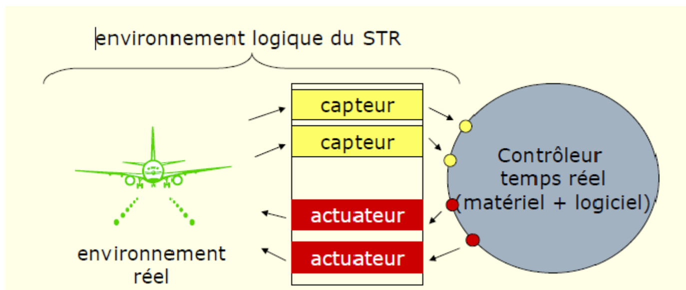


Figure – Exemple d'un système embarqué temps réel

Domaines d'applications

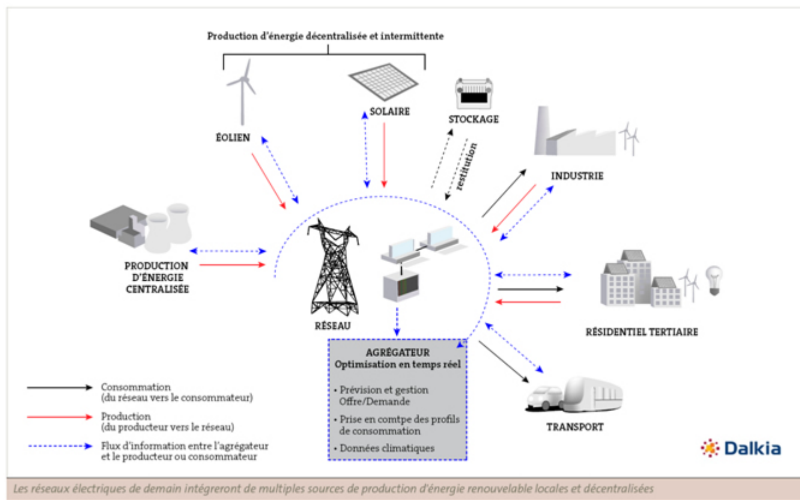


Figure – Un système embarqué temps réel (domaine des énergies)

Caractéristiques

- Large, complexe et fiable :
  - Un système temps réel interagit avec un environnement extérieur souvent complexe et en évolution,
  - Il doit garantir une fiabilité importante
- Utilisation du temps concret :
  - Au sein d'une application ou d'un système temps-réel il faut pouvoir manipuler le temps concret (horloge) pour limiter la **date début** et la **date fin**,
  - Il peut être nécessaire de pouvoir modifier ces paramètres en cours d'exécution et de pouvoir préciser les actions à prendre en cas de faute temporelle

Caractéristiques

- Découpé en tâches ou en processus Concurrents :
  - Dans le monde réel les périphériques et l'environnement du système évoluent simultanément (en parallèle ou concurrence),
  - Le modèle utilisé en programmation des systèmes temps réel est un modèle basé sur la concurrence (applications concurrentes)
- Respect des échéances temporelles :
  - Vue la limitation des ressources et afin de respecter en permanence les échéances, il faut gérer efficacement la pénurie et tenter de favoriser les processus (Ordonnancement) dont l'avancement est le plus "urgent",

Rappel :Système d'exploitation temps réel

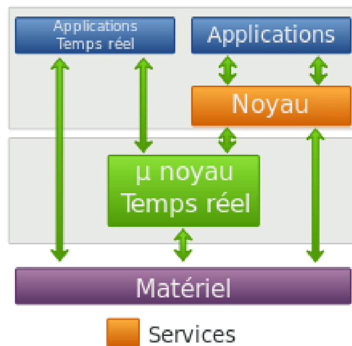


Figure – Organisation d'un système d'exploitation temps réel

Intégration du technologie (Plate-formes, SE, langages de programmation)

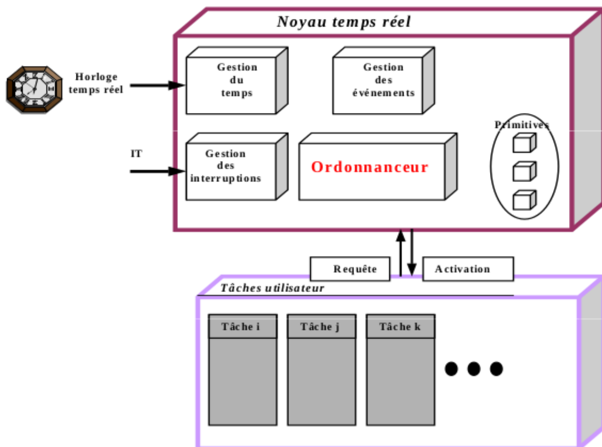


Figure – Organisation d'un noyau temps réel

Exemples de Système d'exploitation temps réel

- Produits commerciaux
  - Lynx-OS : système Unix à base de thread noyau - compatible avec Linux http://www.lynx.com/
  - QNX : système Unix www.qnx.com/
  - Windows Embedded : système Microsoft temps réel https://www.microsoft.com/windowsembedded/en-us/products-solutions-overview.aspx
  - VxWorks et pSos : Exécutif de Wind river https://www.windriver.com/products/vxworks/
  - Nucleus Real-Time Operating System https://www.mentor.com/embedded-software/nucleus/
- Open sources – extensions de Lunix
  - RT-Linux
  - KUSP : www.ittc.ku.edu/kurt
  - RTAI : https://www.rtai.org/

Intégration du technologie (Plate-formes, SE, langages de programmation)

Langage de programmation

- Langages assembleurs
  - Historiquement, ces langages furent longtemps les seuls à être utilisés dans le contexte des STR
  - Dépendant par nature de l'architecture cible (matériel et système d'exploitation)
  - Aucune abstraction possible et grande difficulté de développement, de maintenance et d'évolution
  - Langages à proscrire sauf pour l'implémentation de petites fonctionnalités très spécifiques et apportant une grande amélioration des performances.

Intégration du technologie (Plate-formes, SE, langages de programmation)

Langage de programmation

- Langages séquentiels liés à des librairies système
  - Les plus connus sont le C, le C++, Fortran
  - Apportent un plus grand pouvoir d'abstraction et une certaine indépendance du matériel
  - Mais, doivent faire appel à des librairies systèmes spécifiques pour la manipulations des processus
  - Ils posent le problème de la standardisation des appels systèmes mais sont quelques fois le seul choix possible à cause de la spécificité d'une cible et des outils de développement sur celle-ci.

Intégration du technologie (Plate-formes, SE, langages de programmation)

Langage de programmation

- Langages concurrents de haut niveau
  - Langages généralistes incluant de plus la notion de tâches et des primitives de synchronisation
  - Haut pouvoir d'abstraction, indépendance des architecture et des systèmes cibles
  - Parmi ces langages, Ada et Java sont des plus aboutis
  - Ces langages sont à privilégier lorsque d'autres contraintes ne rendent pas leur choix impossible

Contraintes temporelles et modélisation (Gantt)

Sources des contraintes de temps

Origines externes

- Caractéristiques physiques (vitesse, durée de réaction chimique, ...)
- Caractéristiques liées aux lois de commande du système physique
- Qualité de service requise en terme de délai de réponse tolérable (qualité audio/vidéo)
- Perception sensorielle et le temps de réaction de l'homme
- Contraintes à caractère commercial
- Autres

Sources des contraintes de temps (Suite)

Origines internes

- Choix de conception
  - architecture centralisée ou répartie (données, traitements, contrôle)
  - actions périodiques (avec les périodes adéquates) ou apériodiques
  - choix d'une structure d'application (interface entre composants, ...)
  - autres
- Choix d'architecture matérielle et logicielle
  - processeurs avec des vitesses particulières
  - système d'exploitation (intégrant des algorithmes d'ordonnancement)
  - réseau avec des débits et des temps de réponse donnés
  - autres

4 types de liens entre les tâches :

- Date au plus tôt,

Contraintes temporelles et modélisation (Gantt)

4 types de liens entre les tâches :

- Date au plus tôt,
- Date Début,

Avec la possibilité de :

Contraintes temporelles et modélisation (Gantt)

4 types de liens entre les tâches :

- Date au plus tôt,
- Date Début,
- Date fin,

Avec la possibilité de :

Contraintes temporelles et modélisation (Gantt)

4 types de liens entre les tâches :

- Date au plus tôt,
- Date Début,
- Date fin,
- Date au plus tard,

Avec la possibilité de :

Contraintes temporelles et modélisation (Gantt)

4 types de liens entre les tâches :

- Date au plus tôt,
- Date Début,
- Date fin,
- Date au plus tard,

Avec la possibilité de :

- Avance,

Contraintes temporelles et modélisation (Gantt)

4 types de liens entre les tâches :

- Date au plus tôt,
- Date Début,
- Date fin,
- Date au plus tard,

Avec la possibilité de :

- Avance,
- Retard,

Contraintes temporelles et modélisation (Gantt)

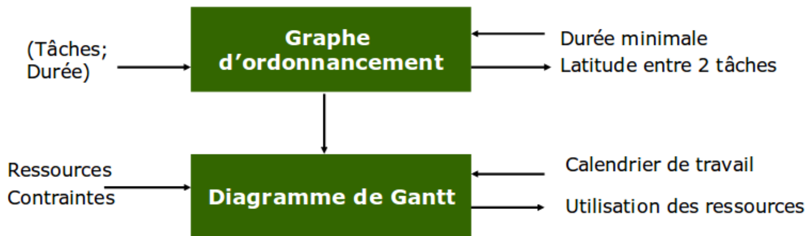
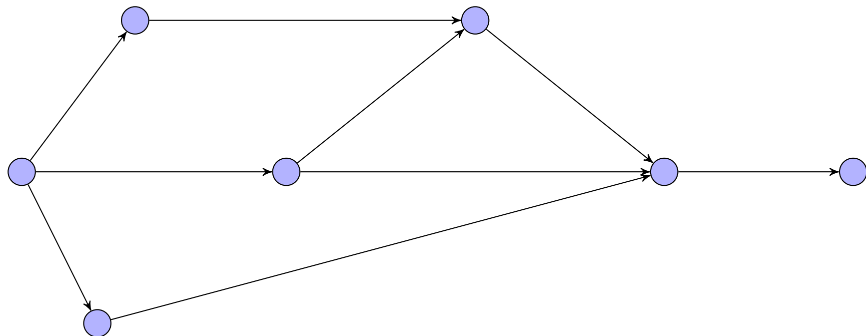


Figure – Les techniques de planification

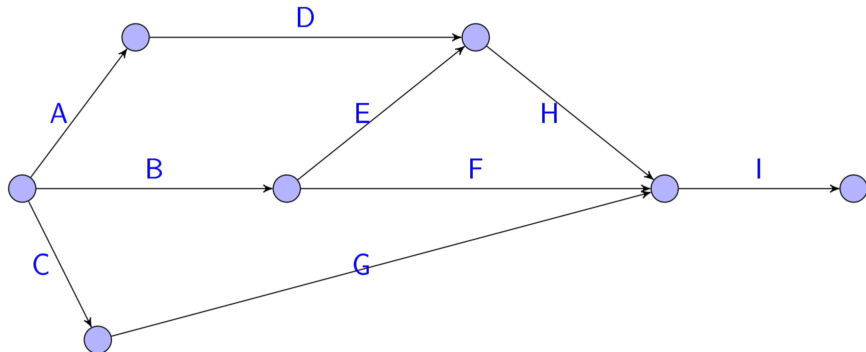
Contraintes temporelles et modélisation (Pert)

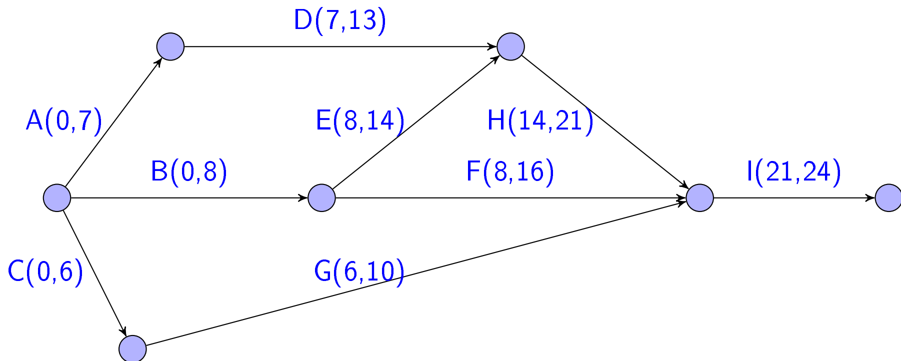
Tâche	Durée	Dépendances
A	7	
B	8	
C	6	
D	6	A
E	6	B
F	8	B
G	4	C
H	7	D, E
I	3	G, H, F

Contraintes temporelles et modélisation (Pert)

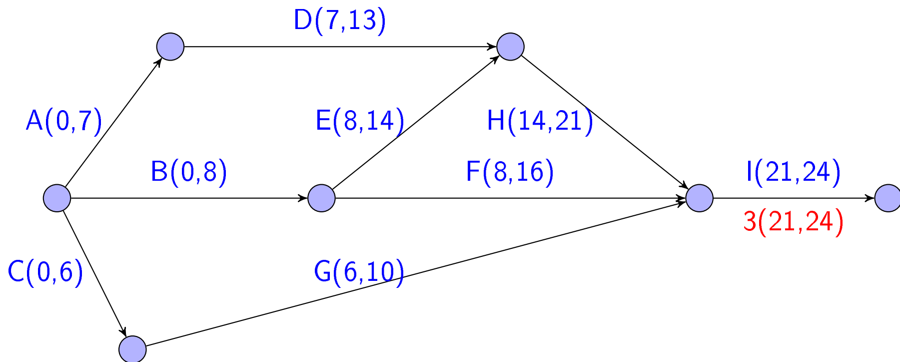


Contraintes temporelles et modélisation (Pert)

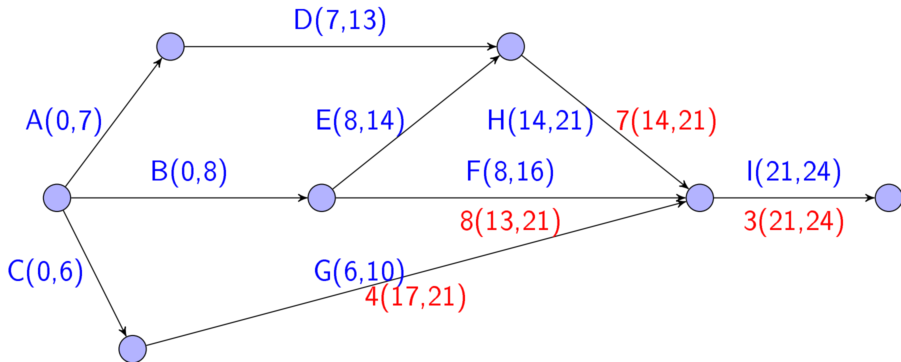



D(7,13)
A(0,7)
E(8,14)
H(14,21)
B(0,8)
F(8,16)
I(21,24)
C(0,6)
G(6,10)

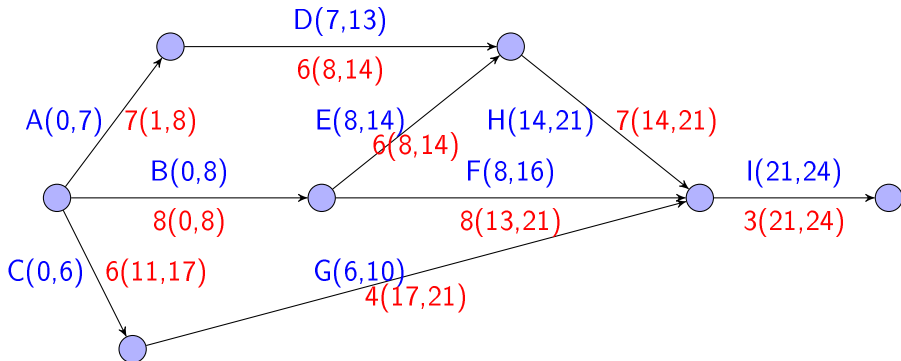
Contraintes temporelles et modélisation (Pert)



Contraintes temporelles et modélisation (Pert)



Contraintes temporelles et modélisation (Pert)



Contraintes temporelles et modélisation (Gantt)

Exemple diagramme de Gantt

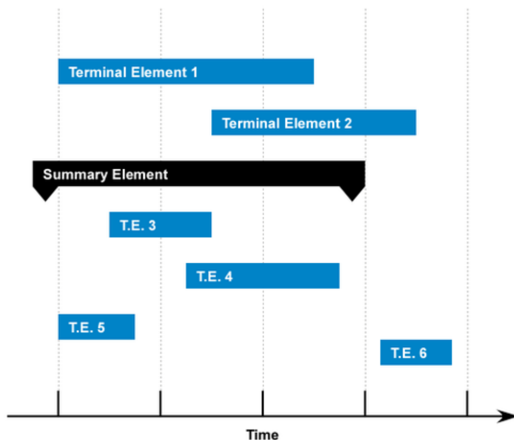


Figure – Exemple du diagramme de Gantt

Criticité

- Tâches Critiques
  - Contrôle airbag, contrôle ABS
  - Contrôle injection
  - injection-moteur
  - Contrôle pneumatique...
- Tâches non Critiques :
  - Contrôle climatisation
  - Contrôle vitres
  - Radio ...

Classification

Classement selon le respect des contraintes temporelles :

- Temps-réel dur ou critique (hard real-time) :
  - Le non respect des contraintes temporelles entraîne de graves défaillances du système
  - Ex : Système de pilotage d'un avion, guidage d'un missile, etc.
- Temps-réel mou ou souple (soft real-time) :
  - Le respect des échéances temporelles est importante mais la violation de celles-ci ne provoque pas de graves défaillances du système
  - Ex : vidéoconférence, etc.

  - Temps-réel ferme (firm real-time) : les échéances peuvent occasionnellement être manquées sans mettre le système en erreur. Le résultat ne sert plus à rien après l'échéance
  - Ex : Internet

Exercice

tâches	antécédents	durée
A	/	3
B	A	1
C	A	5
D	C,I	6
E	B,D	4
F	C,I	2
G	E,F	9
H	/	5
I	H	8
J	H	2
K	I	3
L	K,J	7

Figure – Les techniques de planification

Vue métier

- Chimiste :Formes, durées ... des réactions chimiques
- Médecin :Types de maladie, paramètres biologiques...
- Pilote :Décollage, atterrissage, manette, altitude...

Vue de l'automaticien

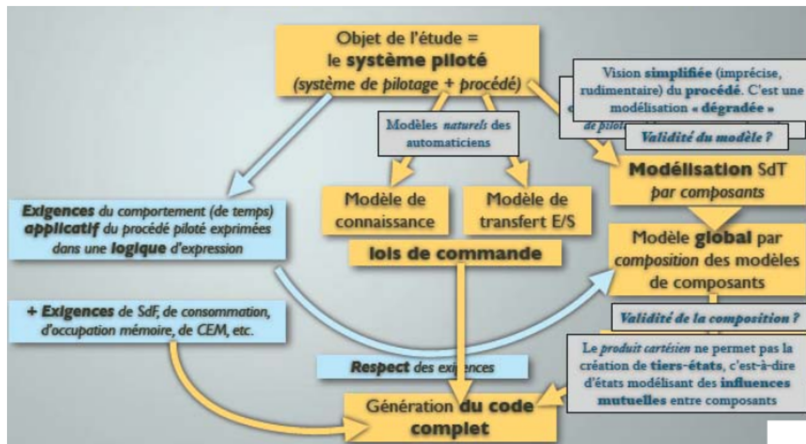


Figure – vue d'un automaticien

Vue de l'informaticien

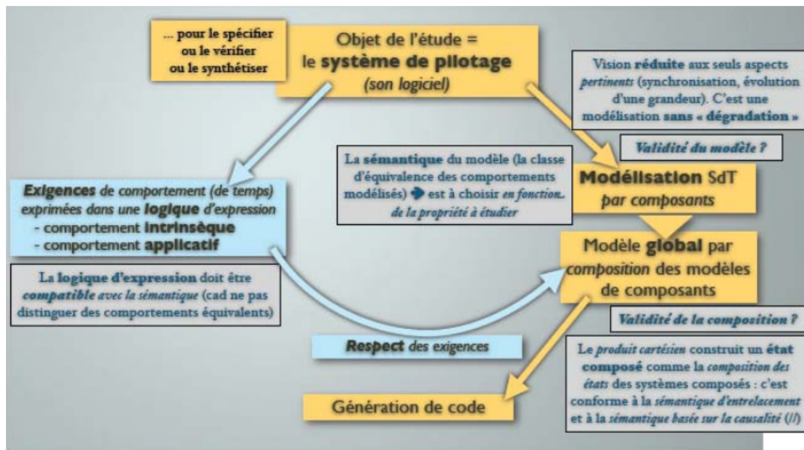


Figure – Vue d'informaticien