Objectifs du cours

Systeme temps réel est un système parallèle

- savoir comment utiliser les primitives de synchronisation

mais aussi:

- savoir pourquoi on les utilise,
- savoir comment en les écrit

Systèmes multitâches

- Un système a un certain nombre de traitements séquentiels à effectuer (suites d'instructions)
- On découpe les traitements à faire en plusieurs fils d'executions (threads)
  - Car ils ont des rythmes (périodes) différents ou pas entièrement défini (tâches sporadiques)
  - Ou pour profiter du parallélisme (architecture multiprocesseurs, parallélisme IO/CPU)

Note: j'appelle ici thread tout fil d'execution, pas seulment les thread POSIX

- deux processus sont deux threads différents
- un processus et son signal handler sont deux threads différents
- un thread noyau et un interrupt handler sont deux threads différents
- etc..

Deux besoins primordiaux:

- La communication entre les threads: transfert de données entre les threads
- La synchronisation entre les threads (ordonnancement de traitements les uns par rapport aux autres

Pourquoi synchroniser?

- résoudre les problèmes de cohérence mémoire sur la communication de données (en mémoire partagée)
- spécifier les dépendences entre les traitements:
  - contrôler l'ordre d'execution des threads
    - Ex: producteur/consommateur: cas des listes pleines et vides, semaphores
  - Execution de section critique
    - Ex: commandes de peripheriques/du materiel (s'assurer que 2 commandes sont faites à la suite sans interruption)
- En général: régler les problèmes de concurrence sur l'accès à une ressource (logicielle ou matérielle) partagée

Primitives de synchronisation

- Mutexes (sémaphores binaires)
- Sémaphores
- Barrières
- Variables conditionelles

Mais aussi:

- Masquer les interruptions
- Atomicité du matériel (bus et instruction)
- spinlocks
- envoi de signaux..

(dépend du niveau où on se place: matériel, OS, logiciel)

Mécanismes de synchronisation

On met en oeuvre les primitives de synchronisation pour créer des mécanismes (ou patterns) de synchronisation
Exemple de mécanismes:

- La section critique:
  - Permet de s'assurer qu'un traitement est executé de manière séquentielle
  - Par rapport à un certain nombre d'autres traitements : (ex: multithread dans multiprocessus)
- Le sémaphore privé (producteur/consommateur)
  - Permet à un thread de controller quand un autre s'execute
- D'autres (les reader/writer locks, etc)

Mécanismes de communication

- Mémoire partagée
- tubes fifo
- boites aux lettres asynchrones
- buffers circulaires
- ...

Note

Toute communication entre deux threads utilise à un moment de la mémoire partagée

(Éventuellement caché dans le noyau)
$\longrightarrow$ Mécanisme principal à maîtriser

Cohérence globale

- Une liste chaînée d'éléments en mémoire partagée:
  - `début` référence le premier élément
  - `fin` référence le dernier élément
  - `debut` et `fin` valent 0 si la liste est vide
  - 2 opérations sur la liste: `ajouter` et `rechercher`
- Deux processus:
  - L'un veut ajouter
  - L'autre veut rechercher

```
ajouter(E *elem)
{ p1: if (fin != 0)
  p2:   fin->suivant = elem;
        else
  p3:   debut = elem;
  p4: fin = elem;
  p5: fin->suivant = 0;
}

rechercher(E *elem)
{ c1: E *e = debut;
  c2: for(; e != 0; e = e->suiv
            if(test(e, elem)) brea
  return e;
}
```

Autre example

Attentions aux problèmes cachés: une instruction C = plusieurs instructions assembleur!
Ex: Une variable, deux threads: l'un l'incrémente, l'autre la décrémente

`i--`

	reg	i
	?	10
`reg = i;`	10	10
`reg = reg - 1;`	9	10
`i = reg;`	9	9

En général : pour tout accès à une mémoire partagée entre deux threads, il faut réfléchir au problème de la cohérence.

Autre example

Attentions aux problèmes cachés: une instruction C = plusieurs instructions assembleur!

Ex: Une variable, deux threads: l'un l'incrémente, l'autre la décrémente

i--

	reg	i
	?	10
`reg = i;`	10	10
`reg = reg - 1;`	9	11
`i = reg;`	9	9

En général : pour tout accès à une mémoire partagée entre deux threads, il faut réfléchir au problème de la cohérence.

Sources d'incohérences

- Ce n'est pas le parallélisme
- Mais les interactions entre programmes se déroulant en parallèle
  - Mémoire commune
  - Ressources partagées
  - Communications (ordre des)
  - ...

Définitions

A, B 2 traitements

- Sériabilité:
  - A//B est indépendant de l'ordonnancement
  - A//B = A, B = B, A
- Atomicité
  - A est atomique par rapport à B si
    - A ne peut pas être mis en // avec B ou
    - A ne peut pas être preempté au profit de B ou
    - B ne peut pas observer d'états intermédiaires dans A pendant l'exécution de A ou
    - A dure un temps nul pour B
- Remarques:
  - Les périodes d'atomicité diminuent le taux de parallélisme
    - Elles ne doivent pas faire rater les échéances
    - Elles doivent être brèves en multicpu
  - L'atomicité évite certaines interactions
    - Mais ne résout pas A, B = B, A

Attitude face aux risques d'incohérence

- A et B doivent être atomiques l'un vis à vis de l'autre
  - Prévention → atomicité
  - Guérison → estampilles
- Dépend de la possibilité (probabilité) d'avoir A et B actifs ensembles (estampilles: peuvent ne pas finir!)
- Guérir:
  - Copier la date (l'estampille)
  - Copier les données
  - Calculer les nouvelles valeurs
  - début atomicité
  - Copier la date actuelle
  - Si elle est inchangée
    - Modifier les données et
    - augmenter la date
  - fin atomicité
  - Si non recommencer

Rappels sur les interruptions

- Les instructions machines s'executent séquentiellement, sauf:
  - Saut
  - Exception (instruction invalide)
  - Trap ("Interruption demandée")
  - Interruption (évènement matériel)
- $\longrightarrow$ Les interruptions sont le seul moyen pour le système d'exploitation de "reprendre la main" de manière forcée sur le processeur
- $\longrightarrow$ preemption implémentée avec elles (notamment: interruption de timer)

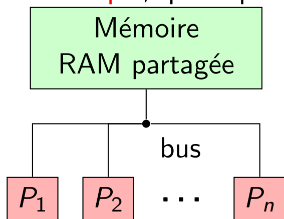
Atomicité des instructions

Principe:

- une instruction assembleur s'execute atomiquement
- i.e. une interruption ne peut survenir qu'entre deux instructions
  - (Note: réordonnancement des instructions dans le pipeline lorsqu'execution dans le désordre
- $\rightarrow$ atomicité des instructions garantie entre deux threads ordonnancés preemptivement

Rappels sur le modèle de calcul parallèle en mémoire partagée

Le modèle von neumann en multiCPU.
Bus unique, qui séquentialise les accès



Atomicité des accès bus

Principe:

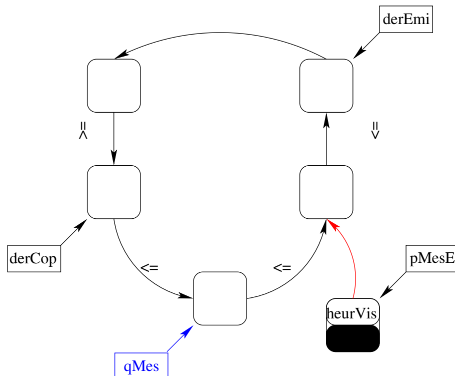
- Le bus d'interconnection mémoire/CPU ne traite qu'une requête à la fois
- → les requêtes sont séquentialisées

Ex: `a = b + c` → `addl a, b, c`
Si `a`, `b`, `c` peuvent être des addresses mémoire → 3 accès bus!
(on ne compte pas l'accès pour la lecture de l'instruction)

En général (processeurs modernes), une instruction = au plus un accès mémoire mais pas toujours!
Ex: processeurs RISC: accès mémoire se fait seulement par instruction load/store: 1 accès bus
Pentium: instruction `add a, b` (`a = a + b`) où `a` peut être une adresse mémoire (mais pas `b`) → 2 accès bus!

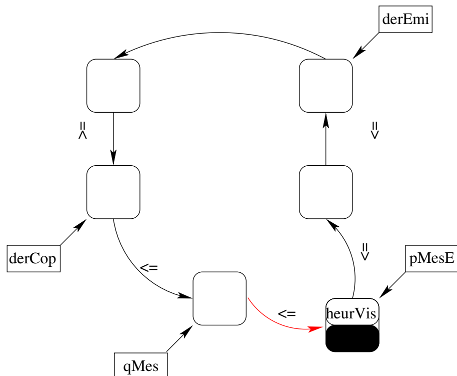
Exemple: insertion atomique dans une liste (1)

- `pMesE->arri = qMes->arri`



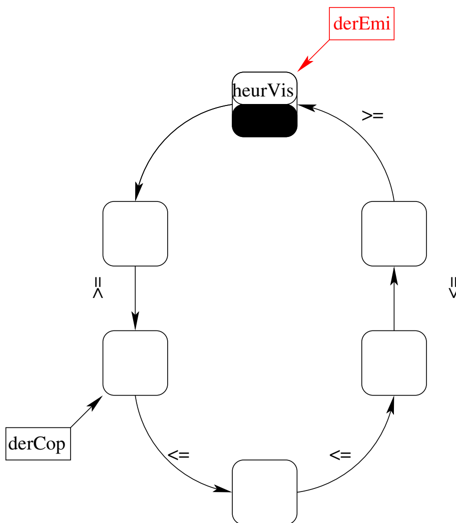
Exemple: insertion atomique dans une liste (2)

- `SFENCE`
- `ATOMIC(qMesE->arri = pMesE)`



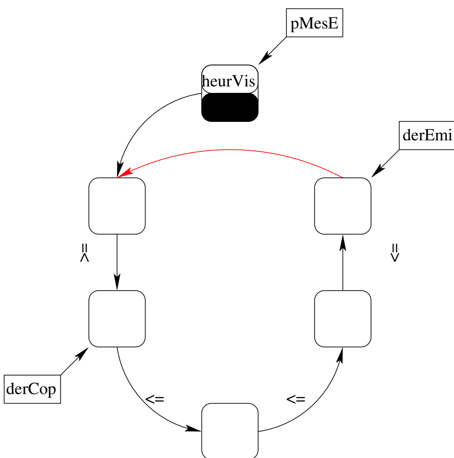
Exemple: deletion atomique d'une liste (2)

- Si `pMesE >= derEmi` (plus récent)
- `SFENCE`
- `ATOMIC(derEmi = pMesE)`



Exemple: deletion atomique d'une liste (2)

- Retire le message de la liste :
- `ATOMIC(pMesE->suiv->arri = pMesE->arri;)`
- `pMesE->arri->suiv = pMesE->suiv`



Remarques

- Pas d'instruction spéciale, juste des read et des writes!
- Dernier exemple: il faut retirer un élément quand il n'y a plus de lecteur
- Attention à l'indéterminisme!
- On peut aussi implémenter avec des read/writes:
  - Des buffers circulaires
  - Des sémaphores!

Instructions à accès mémoire multiple

Exemple :

C:

```
int i = 0;
void handler(void) { i++;}
void main(void) { i--;}
```

Assembleur Pentium:

```
handler: add i, $1
ret

main: add i, $-1
ret
```

- Handler ininterruptible (interruptions masquées)
- Main interruptible

→ marche en monoprocesseur (atomicité des instructions)
→ ne marche pas en multiprocesseur (atomicité du bus)

Instructions à accès mémoire multiple

Exemple :

C:

```
int i = 0;
void handler(void) { i++;}
void main(void) { i--;}
```

La traduction pourrait aussi être:

handler: add i, $1
ret

main: mov %eax, i
add %eax, $-1
mov i, %eax
ret

Ne marche plus non plus en monoprocesseur
→ Attention à la compilation!

Verrouillage du bus

- Extension de l'interface entre le bus et le CPU pour le SMP
- Permet de monopoliser le bus pour l'execution d'une instruction
- $\longrightarrow$ Permet l'atomicité des instructions à accès mémoire multiple en multiprocesseur

Note: sur certains processeurs, ce verrouillage est accessible au programmeur explicitement. Exemple, le Pentium, prefixe “lock”.
Note: ralentit l'execution des autres processeurs

Programme correct en SMP:

```
 handler:  lock add i, $1
ret

main:  lock add i, $-1
ret
```

Instructions spéciales

Instructions spéciales qui profitent du verrouillage du bus pour offrir des primitives pratiques:

- Test&Set
- Compare&Swap
- Fetch&Add (Fetch&Op)

Permettent en particulier l'implémentation efficace de primitives de plus haut niveau (mutex, semaphores, spinlocks)

Sémantique des instructions spéciales

Sémantique: fonctions suivantes executées atomiquement

```
bool tas(bool *addr)
{if(*addr == false)
     { *addr = true; return true;}
else { return false;}}

bool cas(int *addr, int val_attendue, int newval)
{if(*addr == val_attendue)
     { *addr = newval; return false;}
else { return false;}}

int fetch_and_add(int *addr, int increment)
{ int oldval = *addr;
  *addr += increment;
  return oldval;
```

Example: work-stealing avec fetch_and_add

```
struct { int x; int y;} params[300];
int done = 0;

thread_function()
{
 int i;

 while((i = fetch_and_add(&done, 1)) < 300)
 {
   func(params[i].x, params[i].y);
 }
 exit_thread();
}
```

→ Synchronisation wait-free

Example: work-stealing avec compare_and_swap

```
struct { int x; int y;} params[300];
int done = 0;

thread_function()
{ int i;
 while(1)
 {
 do {
     i = done;
     if(i >= 300) goto end;
    }   while(compare_and_swap(&done, i, i+1));

   func(params[i].x, params[i].y);  }
 end:
 exit_thread(); }
```

Example: work-stealing avec test_and_set

```
struct { int x; int y;} params[300];
int done = 0;
int done_lock = 0;

thread_function()
{
 int i;
 while(1)
 { while(test_and_set(&done_lock));
   i = done;
   done = done + 1;
   done_lock = 0;
   if(i >= 300) break;
   func(params[i].x, params[i].y);
 }  exit_thread(); }
```

Section critique

Definition (Section critique)

Une section critique est une portion de code dans laquelle un thread utilise des ressources qui peuvent être utilisées par d'autres threads, mais qui ne peuvent pas être utilisées par eux au même moment.

Un exemple commun de ressource partagée est un ensemble de blocs mémoire.

Attention: les sections critiques diminuent le taux de parallélisme
$\longrightarrow$ il faut les minimiser

Masquage des interruptions

Principe:

- Utilisation d'instructions speciales `mask_interrupts/unmask_interrupts`
- Les interruptions ne sont plus recues, mais sauvegardées jusqu'à ce qu'on les démasque
- $\longrightarrow$ pas de preemption
- $\longrightarrow$ section critique simple.

Avantages:

- Simple, rapide, efficace

Inconvénients:

- Si on reçoit plusieurs interruptions, on peut en perdre une $\longrightarrow$ reservé à des codes de section court
- Réservé à du code de confiance ($\longrightarrow$ instruction privilégiée)
- Exceptions peuvent encore arriver (défaut de page, etc...)
- Ne constitue pas une section critique en multiprocesseur

Spinlocks

Principe:

- Une variable indique si le lock est utilisé (1) ou non (0)
- On vérifie si on peut rentrer en section critique
- On met le lock à 1 quand on y rentre
- On le met à 0 quand on sort

Utilisation:

```
spin_lock(lockvar);
traitement();
spin_unlock(lockvar);
```

Implémentation: Naïve:

```
while(lockvar); lockvar = 1; traitement(); lockvar = 0;
```

Réelle:

```
while(!cas(lockvar, 0, 1)); traitement(); lockvar = 0;
```

Spinlocks

Principe:

- Une variable indique si le lock est utilisé (1) ou non (0)
- On vérifie si on peut rentrer en section critique
- On met le lock à 1 quand on y rentre
- On le met à 0 quand on sort

Utilisation:

```
spin_lock(lockvar);
traitement();
spin_unlock(lockvar);
```

Implémentation: Naïve:

```
while(lockvar); lockvar = 1; traitement(); lockvar = 0;
```

Réelle:

```
while(!cas(lockvar, 0, 1)); traitement(); lockvar = 0;
```

Avantages:

- Simple, se fait rapidement dans le cas ou il n'y a pas de contention

Inconvénients:

- Attente active: temps processeur perdu $\longrightarrow$ limiter à des sections de code courtes
- Si un thread est preempté pendant qu'il détient un spinlock?
- Risque de famine d'un des threads

Note: ne sert à rien en monoprocesseur!

Spinlocks + masquage des interruptions

Principe: on masque les interruptions + on acquiert un spinlock
Implémentation:

```
mask_interrupts(); spin_lock(var); traitement(); spin_unlo
```

Noter l'ordre des opérations!

Avantage: On ne peut plus être preempté en détenant le spinlock
Inconvénient:

- Toujours attente active
- Toujours risque de famine
- Augmente le temps maximum pour prendre une interruption (+ temps d'attente d'acquisition du spinlock)

En pratique, les spinlocks sont quasiment toujours executés avec interruptions masquées
→ technique reservée au code privilégié

Alternative au spinlock : le ticket lock

Principe: On prend un tiquet et on attend qu'il s'affiche

Implémentation:

```
int panneau = 0;
int distributeur = 0;

void thread(void)
{
  int i = fetch_and_add(&distributeur, 1);
  while (i != panneau);
  traitement();
  panneau++;
}
```

Note: marche même si overflow des compteurs

Avantage sur le spin lock: Équité (évite la famine)

Rappels : sémaphores

- Sémaphores (Dijkstra)
  - M un entier positif ou nul
  - deux fonctions P et V
  - Soient n_P et n_V le nombre d'appels faits à P et V
    1. P et V sont sérialisables
    2. P ne fait `return` que si $n_P - n_V \leq M$
    3. V débloque un processus bloqué dans P (s'il y en a)
- Non spécifié:
  - Quand la fonction V débloque un processus
  - $\rightarrow$ Le choix du processus réveillé est laissé à l'implémentation
- Interprétation:
  - M est le nombre initial de ressources
  - n_P est le nombre de ressources consommées
  - n_V le nombre de ressources produites
- Cas spécial: le sémaphore binaire

Ex d'implémentation des sémaphores dans du code privilégié

Principe: On prend un tiquet et on attend qu'il s'affiche
Implémentation:

```
int panneau = M; /* panneau = nV + M */
int distributeur = 0; /* distributeur = nP  */

void thread(void)
{
  int i = fetch_and_add(&distributeur, 1);
  while (i < panneau);
  traitement();
  panneau++;
}
```

Note: ne marche plus si overmarche même si overflow des compteurs

Motivation

- Les techniques précédentes implémentent des sémaphores en attente active, et masquent les IT
  - → temps CPU gâché pour des sections de code longue
  - Temps de prise d'interruption augmenté
  - Ne s'applique qu'au code privilégié
- → Interaction des sémaphores avec l'ordonnanceur :
  - Si un lock est pris, l'ordonnanceur fait "autre chose"
  - "sleeplock"
- → Nouveaux usages des sémaphores: exemple producteur/consommateur
  - Mise en attente lorsque la file est vide

Exemple d'implémentation

```
P()
{spinlock_mask_interrupt();
 cpt = cpt - 1;
 if( cpt <= 0)
   {
    ajouter(attente,
       processeur_courant());
    bloquer_return();
   }
 spinunlock_unmask_interrupt();
}
```

```
int cpt = M;
/* cpt = M - nP + nV */
ensemble attente = VIDE;

V()
{ spinlock_mask_interrupt();
  cpt = cpt + 1;
  if (cpt <= 0)
  { debloquer(extraire(attente)
  }
  spinlock_unmask_interrupt();
}
```

Pour usage dans du code non privilégié: appels systèmes correspondants à `P()` et `V()`

Propriétés

Propriétés:

1. cpt peut devenir négatif
2. $cpt = M - n_P + n_V$
3. Soit n_S le nombre de processus sortis de P:

$$n_S < n_P$$

Théorème:

$$n_S = \min(n_P, M + n_V)$$

Démonstration:

1. à l'initialisation $n_S = 0; n_P = 0; n_V = 0; cpt = M \geq 0$
2. par récurrence: execution de P ou V

Famine

Définition:

- Un thread sera dit en famine de CPU quand il n'évolue plus dans son code

Exemples:

- Dans la fonction V la gestion de l'ensemble des threads bloqués est non spécifiée
- $\rightarrow$ avec une stratégie dernier arrivé premier servi, on aurait une famine
- Avec HPF et une gestion par estampille:
  Le thread le moins prioritaire peut boucler indéfiniment
- Avec un spinlock basique: si contention continue, un thread peut se retrouver continuellement bloqué

Interblocage

- Exemple:
  - Deux personnes, Guy et Daniel
  - un CE disposant de forets et d'une perceuse
- Lundi:
  - Guy emprunte les forets
  - Daniel emprunte la perceuse
- Mardi:
  - Guy demande la perceuse
  - Daniel demande les forets
- Question:
  - Quel jour Guy et Daniel pourront-ils faire leur trou?

Problème avec les sémaphores

- Deux processus A et B
- Deux sémaphores S_1 et S_2 avec $M(S_1) = M(S_2) = 1$
- La chronologie suivante:
  - $A : P(S_1)$
  - $B : P(S_2)$
  - $A : P(S_2)$
  - $B : P(S_1)$

Remarque:

- C'est un problème global:
  - A est bloqué et c'est B qui peut faire évoluer la situation
  - B est bloqué et c'est A qui peut faire évoluer la situation
  - $\rightarrow$ la situation ne peut plus évoluer

Note : on peut s'interbloquer avec un seul sémaphore! (interrupt handler dans le TP)

Solutions

- Remède:
  - Annuler l'un des appels à P
  - Il faut faire remonter le processus dans son code
  - Il faut pouvoir restaurer les données du processus
  - et annuler toutes les opérations qu'il à faite
  - En pratique: détection, et destruction des processus concernés

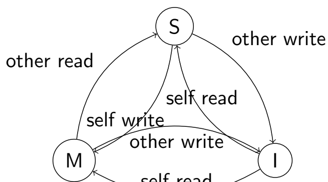
- Prévention
  - Problème compliqué mais
  - dans des cas particuliers, il y a des solutions simples
- Annonce des besoins:
  - Le processus demande en une seule fois toutes les ressources dont il a besoin

- Ressources ordonnées partiellement:
  - Le processus peut faire des demandes successives, pourvu
  - qu'elles portent sur des ressources comparables
  - qu'elles soient faites en suivant la relation d'ordre
- Démonstration:
  - Le processus qui occupe la ressource de plus grand rang, ne peut pas être bloqué
- La condition n'est pas nécessaire:
  - $A : P(R_1), P(R_2), P(R_3)$
  - $B : P(R_1), P(R_3), P(R_2)$

- Sémaphore privé
  - Quand chaque processus n'est autorisé qu'à faire une seule des primitives P ou V
  - Le sémaphore est dit privé (c'est un emploi particulier)
- Interprétation:
  - Le(s) processus qui fait P attend un signal du(s) processus qui fait V
- Propriétés:
  - Si le processus récepteur est en avance, il est bloqué
  - Si le signal est émis en avance, il est mémorisé

Cohérence des caches

- Chaque processeur ayant un cache propre, il dispose de “sa” version de la mémoire, différente des autres
- Les changements peuvent ne pas être répercutés immédiatement
- Pour vision de la mémoire unique: $\rightarrow$ mécanisme de cohérence de cache
- Exemple de protocole: MSI (modified, shared, invalid)
- Implémentation : communication entre les caches + espionnage du bus
- $\rightarrow$ Permet les primitives LL/SC



Note: effet sur les performances

Il faut eviter les invalidations.
Par exemple, ticket lock plus performant que spin lock classique!

Consistance des caches

- à cause des caches, toutes les lectures/écritures ne sont pas faites dans l'ordre des instructions
- + les processeurs font des executions d'instrucions dans le desordre
- $\rightarrow$ ordre des lectures/écritures à l'exécution $\neq$ ordre dans le programme
- $\rightarrow$ besoin de barrières mémoires