Systèmes d'exploitation

Chapitre 3 : IPC et ordonnanceurs

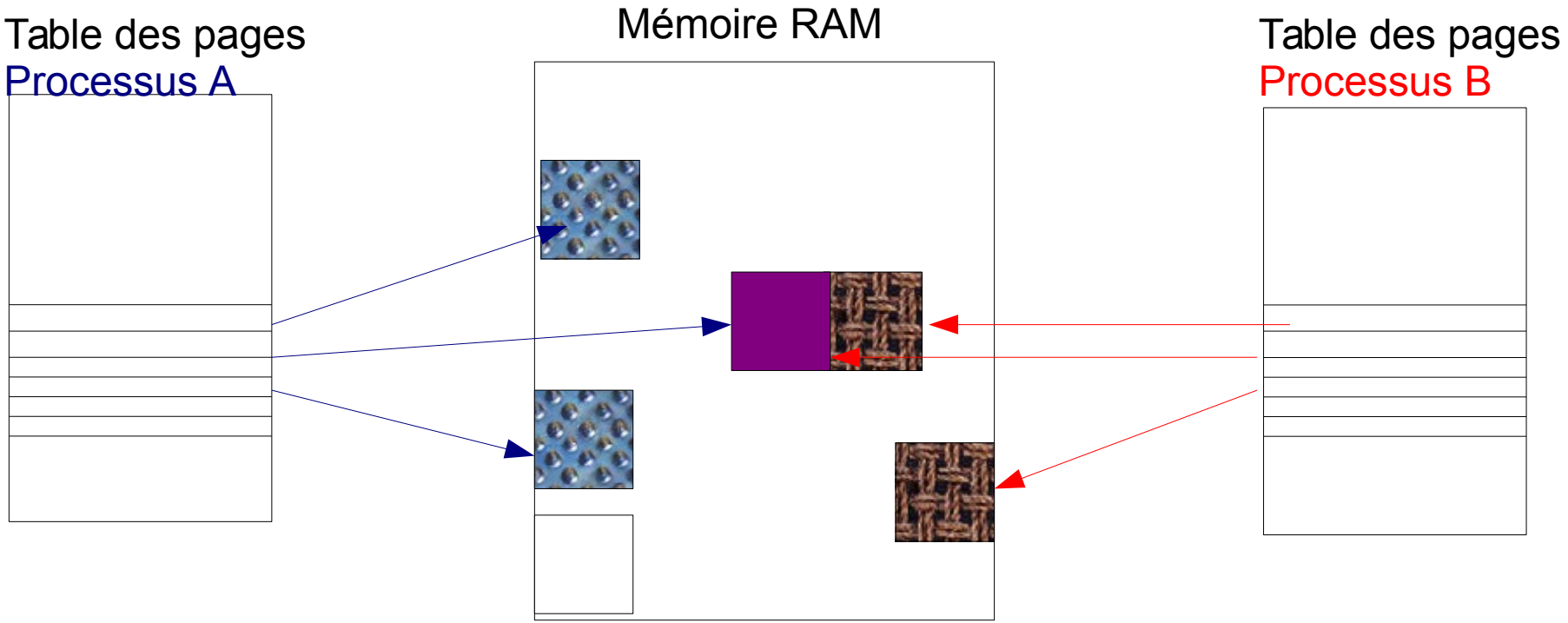
Chargé de cours :
Emery Kouassi Assogba
Tél : 95 22 20 73
Emery.assogba@uac.bj/
emery.assogba@ptgfengineering.com

Agenda

- Processus
- Threads
- Synchronisation entre processus
  - Problème de l'exclusion mutuelle
  - Algorithme de Peterson
  - Sémaphores binaires et entiers
  - Problèmes classiques
    - Dining Philosophers, Producteur consommateur, Readers/Writers
  - Moniteurs
- Echange de messages
- Schedulers

Communication entre processus
Mémoire partagée

- Principe
  - En manipulant la table des pages de deux (ou+) processus, ils peuvent avoir une page en commun



  - Chaque processus peut lire et écrire dans la zone de mémoire partagée

Une banque Naïve

```
send_money(int amount, account my,
account his)
{
     if(my.bal >= amount)
    {
              his.bal=his.bal+amount;
             my.bal=my.bal-amount;
   }
   else
   { /* not enough money */ }
}
```

- Hypothèse
  - Pour des raisons de performance, tous les comptes sont en mémoire partagée
  - Un processus est lancé pour traiter chaque demande de clients homebaking
  - Deux demandes send_money(1000,Alice,Bob) faites par Alice alors qu'elle a 1001 sur son compte

Problème de l'exclusion mutuelle

Processus A

```
…
section_critique() ;
….
```

Processus B

```
…
section_critique() ;
….
```

Processus C

```
…
section_critique() ;
….
```

- Conditions à remplir par une solution
  - Deux processus ne peuvent y être en même temps
  - Un processus se trouvant hors de sa section critique ne peut pas bloquer d'autre processus
  - Un processus ne doit pas attendre indéfiniment le droit d'entrer dans sa section critique
  - Aucune hypothèse n'est faite sur la vitesse relative des processus ou le nombre de CPUs

Comment résoudre l'exclusion mutuelle

- Sur une machine monoprocesseur
  - Un processus ne peut être interrompu que par le kernel
  - Le kernel ne tourne que si appel système ou interruption

Processus A

```
…
Désactiver les interruptions
section_critique() ;
Réactiver les interruption
….
```

- Inconvénients
  - Impossible de désactiver les interruptions dans l'application
    - Seul le kernel peut manipuler les interruptions
    - Deux appels au kernel pour chaque entrée/sortie de section critique
  - Une désactivation trop fréquente des interruptions peut réduire les performances des I/O...

Comment résoudre l'exclusion mutuelle

- Sur une machine multiprocesseur
  - Désactiver les interruptions
    - Nécessite une communication entre tous les CPUs
    - Ne garantit pas que deux processus ne seront pas dans la même section critique sur deux CPUs différents
- Est-il possible de construire une solution sur base uniquement de la mémoire partagée entre les processus/threads ?

Solution naïve

Processus A

```
…
While (turn !=0)
{/*loop*/}
section_critique() ;
Turn=1 ;
...
```

Processus B

```
…
While (turn !=1)
{/*loop*/}
section_critique() ;
Turn=0 ;
...
```

- Variables
  - Turn : partagé par les processus A et B
- Inconvénients
  - Seule une alternance stricte est possible A-B-A-B-A-B...
  - Si un des processus se plante hors de sa section critique, l'autre ne pourra plus y accéder !

Deuxième Solution

Processus A

```
…
While (flag[B]==TRUE)
{/*loop*/}
flag[A] = TRUE ;
section_critique() ;
flag[A]=FALSE ;
...
```

Processus B

```
…
While (flag[A]==TRUE)
{/*loop*/}
flag[B] = TRUE ;
section_critique() ;
flag[B] = FALSE;
...
```

- Variables
  - flag[A]
    - Initialisé à FALSE
    - lu par B et modifié par A
  - flag[B]
    - Initialisé à FALSE
    - lu par A et modifié par B
- Inconvénient
  - Section critique non respectée !

Troisième Solution

Processus A

```
…
flag[A] = TRUE ;
While (flag[B]==TRUE)
{/*loop*/}
section_critique() ;
flag[A]=FALSE ;
...
```

Processus B

```
…
flag[B] = TRUE ;
While (flag[A]==TRUE)
{/*loop*/}
section_critique() ;
flag[B] = FALSE;
...
```

- Variables
  - flag[A]
    - Initialisé à FALSE
    - lu par B et modifié par A
  - flag[B]
    - Initialisé à FALSE
    - lu par A et modifié par B
- Inconvénient
  - Deadlock possible !

Algorithme de Peterson

Processus A

```
…
flag[A] = TRUE ;
Turn = B ;
While (flag[B]==TRUE &&
     turn==B)
{/*loop*/}
section_critique() ;
flag[A]=FALSE ;
...
```

Processus B

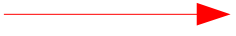
```
…
flag[B] = TRUE ;
Turn = A ;
While (flag[A]==TRUE &&
Turn == A)
{/*loop*/}
section_critique() ;
flag[B] = FALSE;
...
```

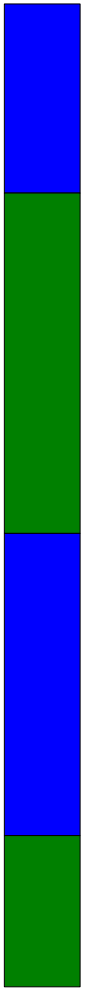
- Variables
  - flag[A]
    - Initialisé à FALSE
    - lu par B et modifié par A
  - flag[B]
    - Initialisé à FALSE
    - lu par A et modifié par B
- Turn
  - Testé et modifié par A et B !

Algorithme de Peterson
Exemple

```
…
flag[A] = TRUE ;
Turn = B ;
```

```
while((flag[B]==TRUE) &&
    (Turn == B)){
   {/*loop*/}
Section_critique() ;
flag[A] = FALSE ;
```





```
…
Flag[B] = TRUE ;
Turn = A ;
while((flag[A]==TRUE) &&
    (Turn == A)){
   {/*loop*/}
```

```
Section_critique() ;
Flag[B] = FALSE ;
```

Algorithme de Peterson

Processus A

```
…
flag[A] = TRUE ;
Turn = B ;
While (flag[B]==TRUE &&
     turn==B)
{/*loop*/}
section_critique() ;
flag[A]=FALSE ;
...
```

Processus B

```
…
flag[B] = TRUE ;
Turn = A ;
While (flag[A]==TRUE &&
Turn == A)
{/*loop*/}
section_critique() ;
flag[B] = FALSE;
...
```

- Deadlock possible ?
  - Il faudrait : ((flag[A]==TRUE) && (flag[B]==TRUE) && (turn==A) && (turn==B))
- Violation de section critique ?
  - A est dans while -> flag[A]=TRUE et turn=B
  - Si B y est déjà, flag[B]=TRUE et A bloque
  - Si B n'y est pas, il ne pourra quitter while que si
    - ((flag[A]==TRUE) && (turn==A)) est faux

With a little help from the hardware...

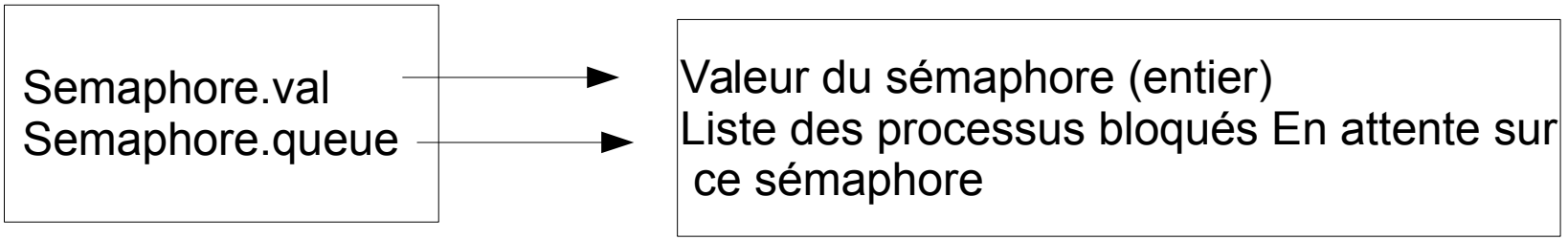
- Instruction Test and Set Lock : TSL
  - TSL Reg, Addr
    - Charge, en une instruction ininterruptible, dans le registre Reg le mot se trouvant à l'adresse Addr et stocke la valeur 1 en mémoire à l'adresse Addr
- Entrée en section critique
  - ENTERR: TSL $R1,LOCK # MEM[LOCK] contains 1 and
  - BEQ $R1,$ZERO,OK # $R1 contains previous value MEM[LOCK] #IF $R1==0 we can enter the region
  - J ENTERR #Busy wait until MEM[LOCK]=0
  - OK : JR $RA #Return to caller
- Sortie de section critique
  - EXITR: LA $T0,LOCK # Stores address of LOCK in $T0
  - SW $ZERO,0($T0) # MEM[LOCK]=0
  - JR $RA # Return to caller

Inconvénients de Peterson et TSL

- Un processus en attente d'entrée en section critique utilise inutilement le CPU
  - Raisonable si la section critique dure 1microsec
  - Inutilisable si la section critique dure longtemps
- Inversion de priorité
  - Le processus P1, de basse priorité, entre en section critique
  - Le processus P2, de haute priorité, devient ready
  - L'OS interrompt le processus P1 et exécute P2
  - P2 veut entrer en section critique et boucle en attendant que P1 quitte sa section critique...
    - Cela ne risque pas d'arriver puisque l'OS préfère P2 à P1

Une meilleure solution

- Principe
  - Faire activement participer l'OS
  - Plutôt que de faire exécuter une boucle infinie par un processus en attente d'entrée, le bloquer et le relancer uniquement lorsqu'il pourra continuer
- Sémaphore
  - Objet maintenu par le kernel



Sémaphores binaires

- Opérations
  - 3 opérations possibles sur un sémaphore binaire
  - Init(s)
    - À zéro ou un
  - Down_b(s)
  - Up_b(s)

```
down_b(binary_semaphore s)
{
  if(s.val==unlocked)
  {
     s.val=locked;
  }
  else
  {
   Place this process in s.queue;
   This process is blocked;
  }
}
```

```
up_b(binary_semaphore s)
{
     if(s.queue is empty)
     {
        s.val=unlocked;
     }
     else
    {
      Remove one process(P) from
     s.queue;  Mark Process(  P)  as ready
      to run;
    }
}
```

Implémentation des sémaphores binaires

- Via une instruction tsl
  - Peut fonctionner en userspace ou dans le kernel et même sur une machine multiprocesseur
  - Mutex : un bit en mémoire
  - Mutex=0 (unlocked)-> down peut réussir sans attente
  - Mutex=1 (locked) -> un processus qui fait down sera mis en attente

```
DOWN:   TSL $R1,MUTEX  # MEM[MUTEX] contains 1 and
                       # $R1 contains previous value MEM[MUTEX]
        BEQ $R1,$ZERO,OK # IF $R1==0 down succeeds
                # OS places this process in s.queue and starts another one
        ....
        J DOWN # Busy wait until MEM[LOCK]=0
OK:     JR $RA  # Return to caller

UP:     LA $T0,MUTEX       # Stores address of MUTEX in $T0
        SW $ZERO,0($T0)              # MEM[MUTEX]=0
        # OS removes one process from s.queue and marks it as ready
        ...
        JR $RA             # Return to caller
```

Sémaphores entiers

- Opérations
  - Seules 3 opérations possibles sur un sémaphore binaire
  - Init(s)
    - À une valeur enière non-négative
  - Down_b(s)
    - Parfois appelé wait(s) ou P(s)
  - Up_b(s)
    - Parfois appelé signal(s) ou V(s)

```
down(semaphore s)
{
      s.val=s.val-1;
     if(s.val<0)
   {
Place this process in s.queue;
This process is blocked;
   }
}
```

```
up(semaphore s)
{
    s.val=s.val+1;
    if(s.val<=0)
   {
Remove one process(P) from s.queue;
Mark Process(P) as ready to run;
    }
}
```

Implémentation des sémaphores

- Via une désactivation des interruptions par le kernel sur une machine monoprocesseur

```
down(semaphore s)
{
   disable_interrupts();
    s.val=s.val-1;
    if(s.val<0)
    {
    Place this process in s.queue;
   This process is blocked;
   re-enable_interrupts();
    }
   else
   {
       re-enable_interrupts();
    }
}
```

```
up(semaphore s)
{
    disable_interrupts();
    s.val=s.val+1;
    if(s.val<=0)
    {
      Remove one process(P) from s.queue;
      Mark Process(P) as ready to run;
    }
    re-enable_interrupts();
}
```

Implémentation des sémaphores(2)

- Via une instruction tsl
  - Peut fonctionner en userspace ou dans le kernel et même sur une machine multiprocesseur

```
down(semaphore s)
{
   while(testandset(s.flag))
   { /* Loop */ }
     s.val=s.val-1;
    if(s.val<0)
    {
      Place this process in s.queue;
     This process is blocked;
   }
   else
   {
     s.flag=0;
   }
}
```

```
up(semaphore s)
{
      while(testandset(s.flag))
     { /* Loop */ }
     s.val=s.val+1;
     if(s.val<=0)
     {
          Remove one process(P) from s.queue;
          Mark Process(P) as ready to run;
     }
     s.flag=0;
}
```

Implémentation des sémaphores entiers avec des sémaphores binaires

```
/*Initialisation*/
Binary_Semaphore mutex,waiting;
mutex=1;
Waiting=0;
Int c ;
down(semaphore s)
{
   down_b(&mutex);
   C = c-1 ;
    If(c<0)
    {
      up_b(&mutex);
      down_b(&waiting) ;
   }
   up_b(&mutex);
}
```

```
up(semaphore s)
{
       down_b(&mutex);
       C=C+1;
      if(c<=0)
      {
           up_b(&waiting);
/* the wakedup process will do up_b(&mutex) */
       }
      Else{
         up_b(&mutex);
    }
}
```

- Rôle des sémaphores binaires
  - mutex : exclusion mutuelle, manipulation de la variable C
  - waiting : sert à bloquer les processus en attente d'une valeur positive du sémaphore entier

Exclusion mutuelle avec sémaphores

Processus A

```
Semaphore s;
...
down(s);
section_critique();
up(s);
...
```

Processus B

```
Semaphore s;
...
down(s);
section_critique();
up(s);
...
```

- Exemple
  - s est initialisé à 1 et accessible aux deux processus
  - Le processus A démarre et A exécute down(s);
    - Le processus A entre en section critique
    - s.val vaut 0
  - L'OS interrompt le processus A et exécute B
  - Le processus B exécute down(s);
    - s.val devient négatif et le processus B est bloqué dans s.queue
  - Le processus A s'exécute et quitte la section critique
  - Lors de l'appel à up(s);, l'OS va retirer le processus B de la queue s.queue et le mettre dans l'état ready

Synchronisation simple avec sémaphores

- Problème
  - Comment faire pour que le processus X ne démarre un traitement qu'après que le processus Y aie fini d'exécuter une procédure particulière ?

```
/* Initialisation */
Semaphore wait=0;
```

Processus X

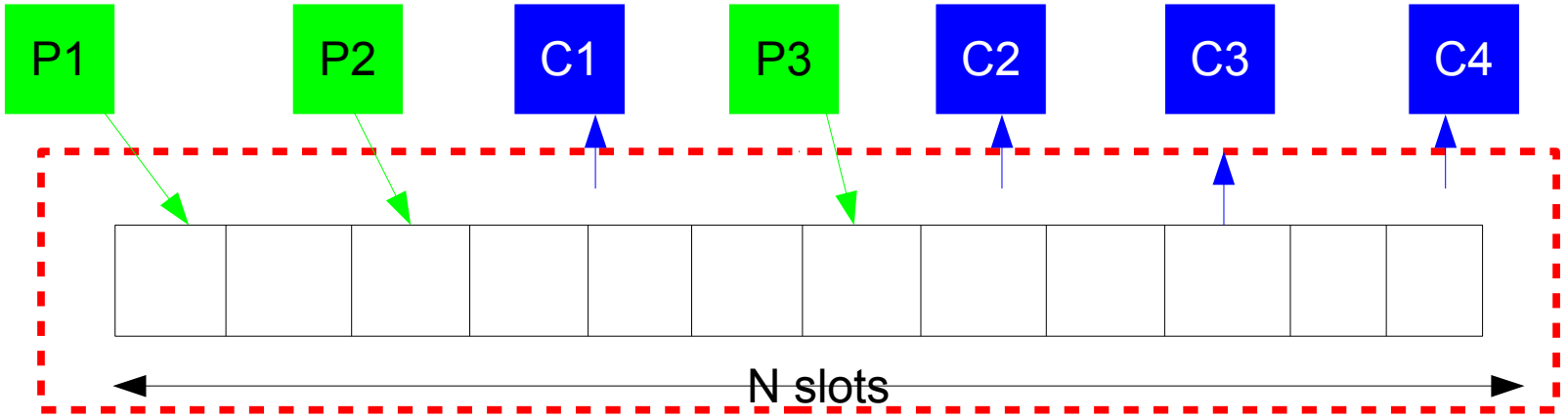
```
….
/* do something*/
…
/* only start after
 * Y has finished calc()
*/
down(&wait) ;
/**continue processing/
...
```

Processus Y

```
...
/* do something */
calc();
/* inform X */
up(&wait);
/* continue processing */
....
```

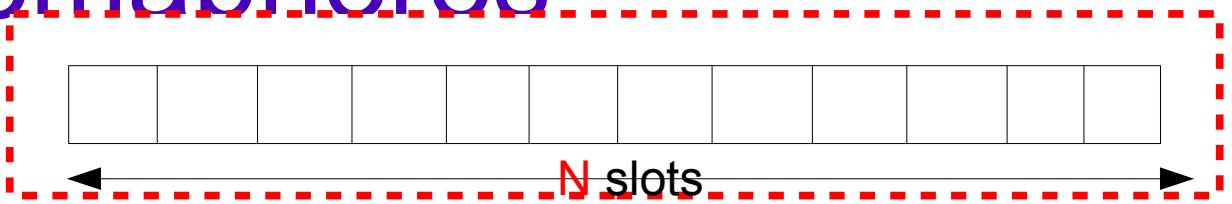
Producteurs consommateurs

- Problème
  - Deux classes de processus
    - Producteurs produisent de l'information et la place dans un buffer partagé
    - Consommateurs traitent de l'information en allant la chercher dans un buffer partagé
    - Comment coordonner ces processus ?



Producteurs consommateurs avec sémaphores

```
/**Initialisation **/
Semaphore mutex = 1 ;
Semaphore empty = N ;
Semaphore full = 0 ;
```



Producteur

```
Void producer(void)
{
   int item;
  while(TRUE)
 {
    item=produce(item);
    down(&empty);
    down(&mutex);
    insert_item();
    up(&mutex);
    up(&full);
  }
}
```

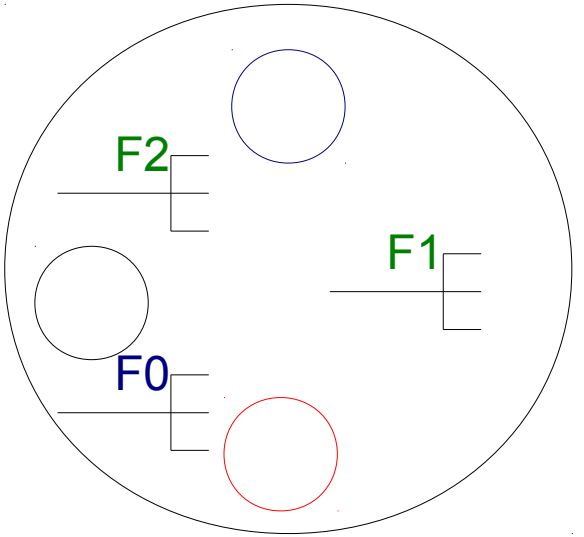
Consommateur

```
void consumer(void)
{
    int item;
    while(TRUE)
   {
     down(&full);
     down(&mutex);
    item=remove(item);
    up(&mutex);
    up(&empty);
   }
```

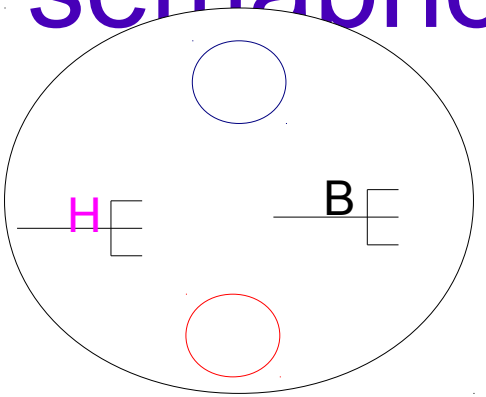
- Mutex : exclusion mutuelle pour l'accès au buffer
- Empty : nombre de places libres dans le buffer
- Full : nombre de places occupées dans le buffer

Le problème des « Dining philosophers »

- Problème
  - N philosophes doivent se partager un repas dans une salle de méditation
  - La table contient N fourchettes et N assiettes
  - Chaque philosophe a un place réservée et a besoin pour manger de
    - La fourchette à sa gauche
    - La fourchette à sa droite
- Comment coordonner
  - efficacement l'accès à la table ?



Exemple avec deux philosophes et des sémaphores



Philosophe 1

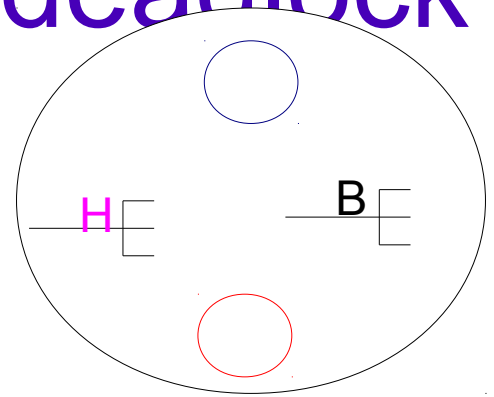
```
while(TRUE)
{
    think();
    down(H); /* fourchette H */
    down(B); /* fourchette B */
    eat();
    up(B); /* fourchette B */
    up(H); /* fourchette H */
}
```

Philosophe 2

```
while(TRUE)
{
think();
    down(B); /* fourchette B */
     down(H); /* fourchette H */
     eat();
     up(H); /* fourchette H */
    up(B); /* fourchette B */
}
```

Deadlock possible !

2 philosophes : solution sans deadlock



Philosophe 1

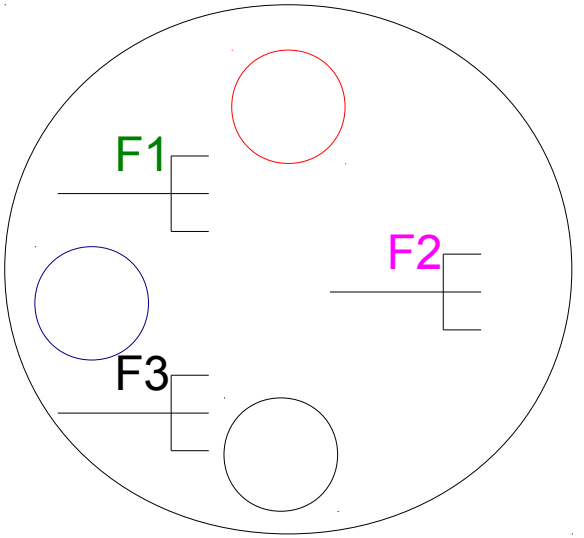
```
while(TRUE)
{
think();
down(B); /* fourchette B */
down(H); /* fourchette H */
eat();
up(B); /* fourchette B */
up(H); /* fourchette H */
}
```

Philosophe 2

```
while(TRUE)
{
think();
    down(B); /* fourchette B */
     down(H); /* fourchette H */
     eat();
     up(H); /* fourchette H */
    up(B); /* fourchette B */
}
```

Différence entre les 2 solutions : ordre de prise des fourchettes

3 philosophes



Philosophe 1

```
while(TRUE)
{
    think();
    down(F1) /*left fork*/;
    down(F3); /*right fork*/
    eat();
   up(F1); /*left fork*/
up(F3); /*right fork*/
}
```

Philosophe 2

```
while(TRUE)
{
think();
down(F2);  /* left fork */
down(F1); /* right fork */
eat();
up(F2);  /* left fork */
up(F1); /* right fork */
}
```

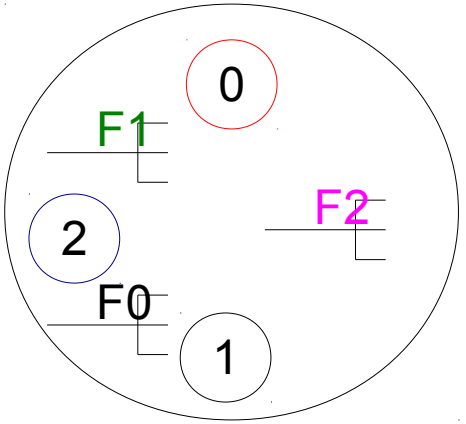
Philosophe 3

```
while(TRUE)
{
think();
down(F3); /*left fork*/
down(F2); /*right fork*/
eat();
up(F3); /*left fork*/
up(F2); /*right fork*/
}
```

Deadlock possible !

Dining philosophers : une solution san deadlock

```
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
/* Initialisation */
Semaphore fork[N]=1;
```



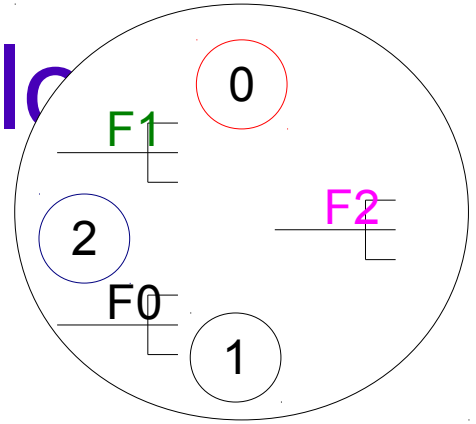
Philosophe[i]

```
while(TRUE)
{
    think();
   /* take forks */
   take_forks(LEFT,RIGHT);
  /* forks taken */
  eat();
  /* release forks */
  up(&fork[LEFT]);
  up(&fork[RIGHT]);
  /* forks released */
}
```

```
Void take_forks(int i, int j)
{
    if(i<j)
   {
     down(&fork[i]);
     down(&fork[j]);
   }
   else
  {
     down(&fork[j]);
    down(&fork[i]);
   }
}
```

Les fourchettes sont ordonnées. Tous les philosophes prennent leurs fourchettes dans cet ordre

Dining philosophers : une autre solution sans deadlo




```
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
/* Initialisation */
int state[N]=THINKING;
Semaphore mutex=1;
Semaphore s[N]=0
```

Philosophe[i]

```
while(TRUE)
{
   think();
   /* take forks */
   down(&mutex);
   state[i]=HUNGRY;
   test(i);
   up(&mutex);
   down(&s[i]);
   /* forks taken */
   eat();
  /* release forks */
  down(&mutex);
   state[i]=THINKING;
   test(LEFT);
   test(RIGHT);
   up(&mutex);
   /* forks released */
}
```

```
Void test(int i)
{
    if(state[i]==HUNGRY &&
    state[LEFT]!=EATING &&
     state[RIGHT]!=EATING)
     {
        state[i]=EATING;
        up(&s[i]);
     }
}
```

Readers/Writers

- Problème
  - Deux classes de processus doivent partager l'accès à une base de données dans des conditions particulières
  - Writers
    - Accès exclusif. Lorsqu'un writer accède à la base de données, aucun autre processus ne peut y accéder
  - Readers
    - Accès partagé entre readers. Un nombre quelconque de processus readers peut accéder en même temps à la base de données.
  - Comment réguler l'accès à la base de données sans risquer de deadlock ni que les readers n'empêchent tout accès des writers ?

Readers/Writers

Reader

```
void reader(void)
{
       while(TRUE)
      {
           down(&mutex);
            readcount=readcount+1;
           if (readcount==1)
           { /* first reader */
                down(&db);
          }
          up(&mutex);
         read_data_base();
        down(&mutex);
       readcount=readcount-1;
       if(readcount==0)
       { /* last reader left */
          up(&db);
        }
       up(&mutex);
        use_data_read();
    }
}
```

```
/* Initialisation */
typedef int semaphore;
semaphore mutex=1; /* for readcount */
semaphore db=1;
/* db : mutual exclusion for database
*either (many) readers use database
*or a single writer uses database
*/
int readcount=0;
```

```
void writer(void)
{
   while(TRUE)
   {
        think_up_data();
        down(&db);
         write_data_base();
        up(&db);
    }
}
```

Les Readers ont priorité sur les Writers

Moniteurs

- Extension de haut niveau à certains langages
  - permet de cacher la complexité des sémaphores au programmeur pour rendre le code plus sûr

```
monitor Nom
Condition a,...;
/* variables declaration */
procedure P1(...);
begin ... end;
procedure P2(...);
begin ... end;
begin
/* initialisation */
end
end monitor;
```

Moniteurs (2)

- Propriétés
  - Exclusion mutuelle
    - A tout moment, un seul processus au maximum peut exécuter du code du moniteur
    - le compilateur peut "entourer" les appels aux procédures du moniteur par des opérations down et up sur un sémaphore mutex
  - Conditions
    - Un moniteur contient plusieurs conditions
    - Un processus peut attendre que la condition x soit vérifiée
    - en faisant wait(x)
      - Si la condition n'est pas vérifiée à ce moment, le processus sera bloqué jusqu'à ce qu'elle le devienne
    - Un processus peut informer les autres processus que la condition y est vérifiée via signal(y)
    - signal(y) réveillera un seul processus en attente sur cette condition
    - Après exécution de signal(y), le processus doit quitter le moniteur immédiatement

Moniteurs : exemple

- Allocation d'une ressource unique

```
monitor Printer
Condition x;
bool busy;
procedure Acquire;
   begin
      if busy then wait(x);
      busy:=TRUE;
  End;
procedure Release;
  begin
     busy:=FALSE;
     signal(x);
   End;
   begin
   /* initialisation */
       busy:=FALSE;
   end
end monitor;
```

Processus[i]

```
...
Printer.Acquire;
/* print somehting */
Printer.Release;
...
```

Producteurs-consommateurs avec moniteurs

```
monitor PC
     condition full,empty;
     integer count;
procedure insert(item:integer)
begin
    if count=N then wait(full);
    insert_item(item);
    count := count+1:
   if count=1 then signal(empty)
End;

function remove:integer
begin
      if count=0 then wait(empty);
      remove=remove_item;
      count:=count-1;
      if count=N-1 then signal(full);
end;
begin
     /* Initialisation */
    count:=0;mutex=1;
end;
```

Producteur

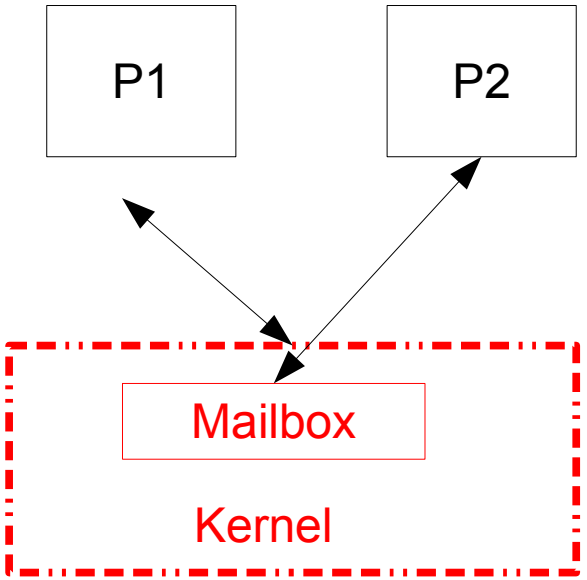
```
procedure producer;
begin
      while true do
          begin
             item=produce_item;
             PC.insert(item);
          end
end;
```

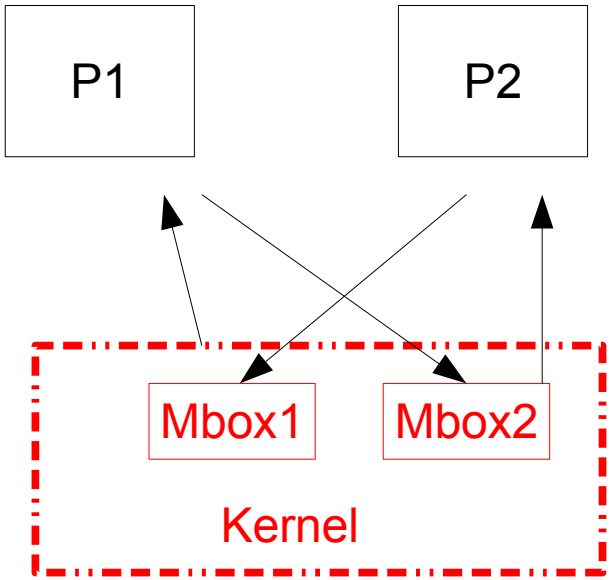
Consommateur

```
procedure consummer;
begin
while true do
    begin
       item=PC.remove;
       consume_item(item);
    end
end;
```

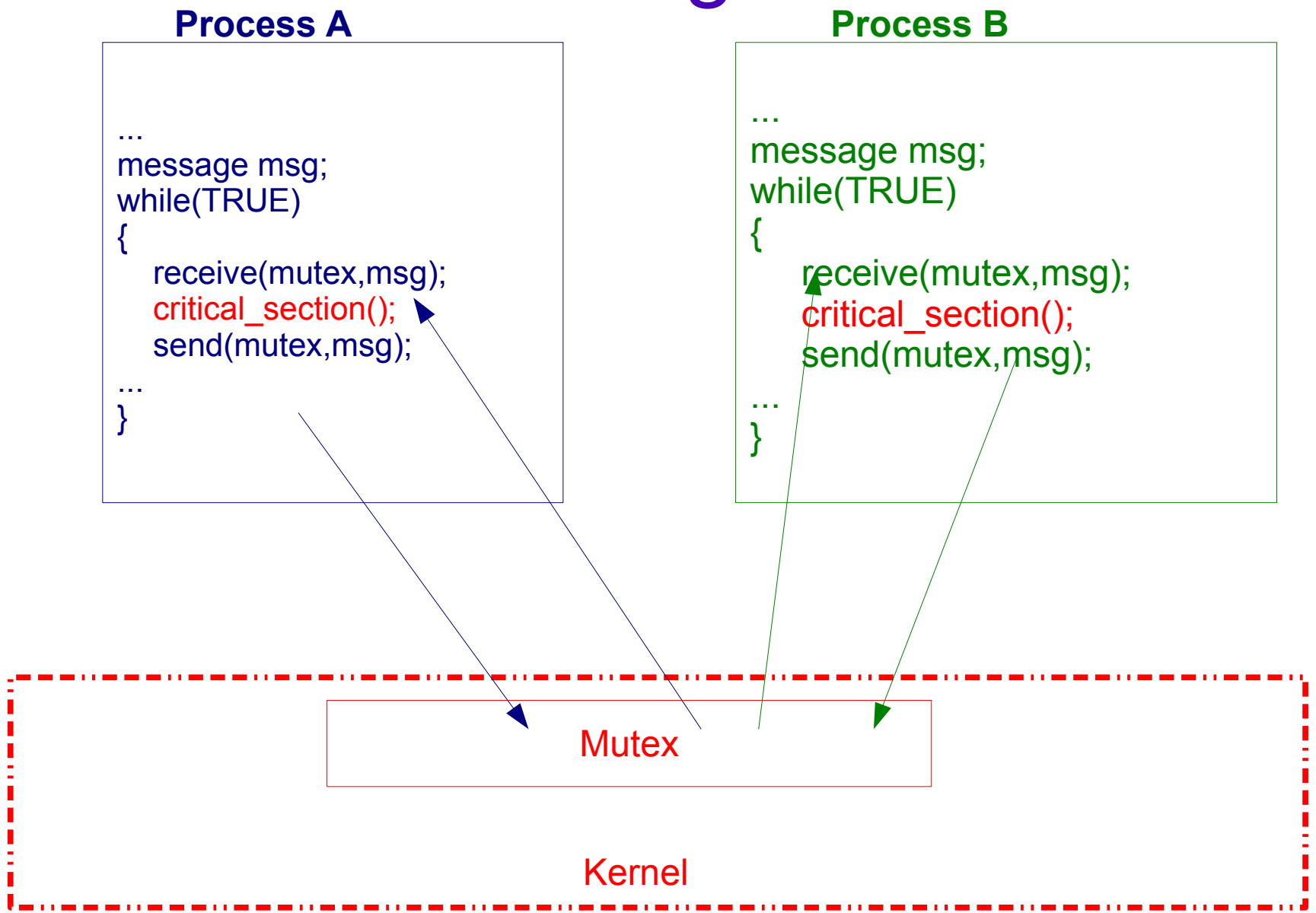
Synchronisation par échange de messages

- Principe
  - Des processus peuvent s'échanger des messages par l'intermédiaire du système d'exploitation
    - Send(destination,message)
      - Envoi d'un message à une destination
    - Receive(source,message)
      - Réception d'un message provenant d'une source





Exclusion mutuelle par échange de messages



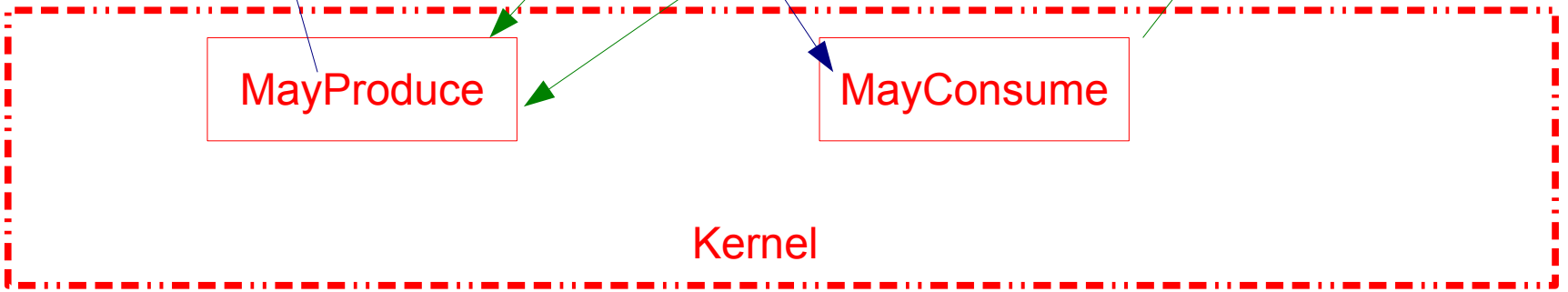
Producteurs/Consommateurs par échange de message

Producteur

```
Int item;
message m;
while(TRUE)
{
     item=produce_item();
    receive(mayproduce,&m);
    build_message(&m,item);
    send(mayconsume,&m);
}
```

Consommateur

```
Int item,i;
message m=NULL;
for(i=0;i<N;i++)
{
     send(mayproduce,&m)
}
while(TRUE)
{
     receive(mayconsume,&m);
     item=extract_item(&m);
     send(mayproduce,&m);
     consume_item(item);
}
```



Agenda

- Processus
- Threads
- Synchronisation entre processus
- Schedulers
  - Schedulers pour systèmes interactifs
  - Schedulers temps-réel
  - Schedulers Unix

Types de systèmes

- Batch processing
  - exécute un grand nombre de jobs dont les caractéristiques sont connues
- Système interactif
  - Plusieurs utilisateurs actifs en même temps sur la machine qui doivent éviter de perdre du temps à cause du système
- Système temps réel
  - certaines tâches doivent s'effectuer dans un délai précis

Scheduling

- Problème
  - Comment efficacemment répartir les ressources de l'ordinateur (CPU, mémoire, I/O) entre les différents processus ?
  - Choix à faire pour répartir ces ressources
    - Lancer ou un pas lancer un nouveau processus
      - fait par l'utilisateur dans les systèmes interactifs
      - fait par un scheduler dans les systèmes batch
    - Charger un processus lancé en mémoire RAM
      - réalisé par le swapper
        - si la mémoire est insuffisante, le swapper peut « geler » un processus et le mettre entièrement sur disque
    - Faire exécuter un processus sur un CPU
      - fait par le scheduler

Scheduling(2)

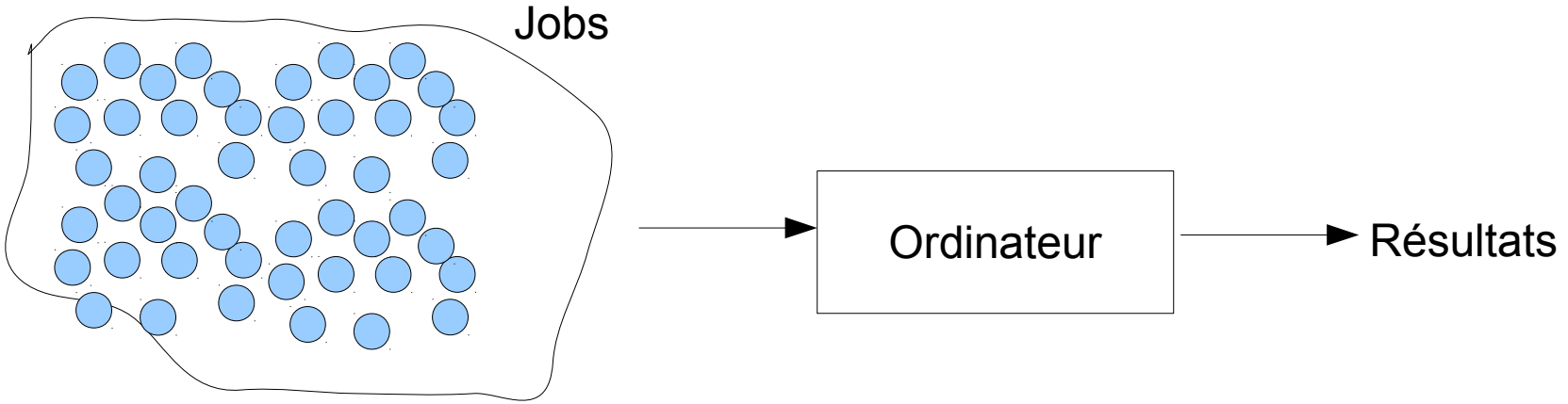
- Pourquoi introduire un scheduler ?
  - Pour répartir le CPU entre plusieurs processus
    - Permet de donner l'illusion à plusieurs utilisateurs qu'ils ont chacun une machine à leur disposition
    - Permet de partager efficacemment et de façon transparente les CPUs sur une machine multi-CPU
    - Permet d'éviter qu'un processus ne puisse consommer toutes les ressources de la machine
  - Permet d'améliorer les performances
    - pendant qu'un processus attend le résultat d'une opération d'I/O, un autre peut s'exécuter
  - Types de schedulers
    - **non-préemptif**
    - scheduler lancé uniquement lors d'accès I/O ou explicitement
    - **préemptif**
    - le scheduler peut s'exécuter sans demande explicite ou implicite d'un processus

Critères d'évaluation des schedulers

- Comment évaluer les performances ?
  - Débit
    - nombre de jobs traités durant une période de temps
  - Temps d'exécution moyen
    - temps moyen d'exécution d'un job
  - Temps de réponse
    - pour les systèmes interactifs
  - Utilisation
    - du CPU
      - pourcentage du temps pendant lequel le CPU exécute des processus utilisateurs
    - des autres ressources (disque, mémoire, ...)
      - il faut éviter qu'un ressource ne soit saturée à un moment et complètement libre à d'autres moments
    - Equité
      - tous les processus doivent pouvoir s'exécuter

Batch processing

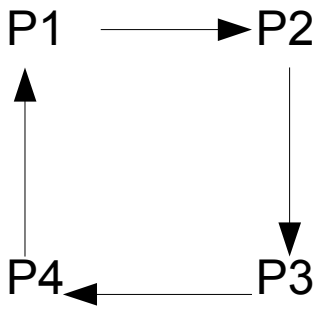
- Problème
  - Comment faire efficacemment exécuter par un ordinateur un grand nombre de traitements ?
  - calcul scientifique
  - gestion de comptes bancaires (ex. calcul intérêts, ...)



- Solutions naïves
  - charger un seul job à la fois et l'exécuter
  - charger tous les jobs en même temps et les exécuter

Schedulers pour systèmes interactifs

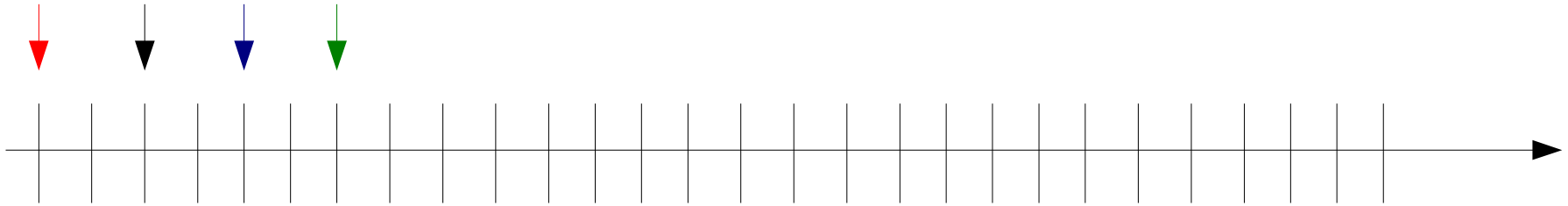
- Round Robin
  - Maintenir une liste circulaire des processus dans l'état Ready avec un pointeur courant
- A chaque activation du scheduler
  - changer de contexte et remplacer le processus courant par celui qui le suit immédiatement dans la liste
- A chaque arrivée d'un processus (nouveau processus ou processus bloqué devenant Ready)
  - insérer le processus dans la liste circulaire



Avantage
Equité
Inconvenient
Equité

Round-Robin Exemple

- Round-Robin scheduling
  - quantum fixé à une unité CPU



B(6unités CPU)
C(3unités CPU)
D(5unités CPU)

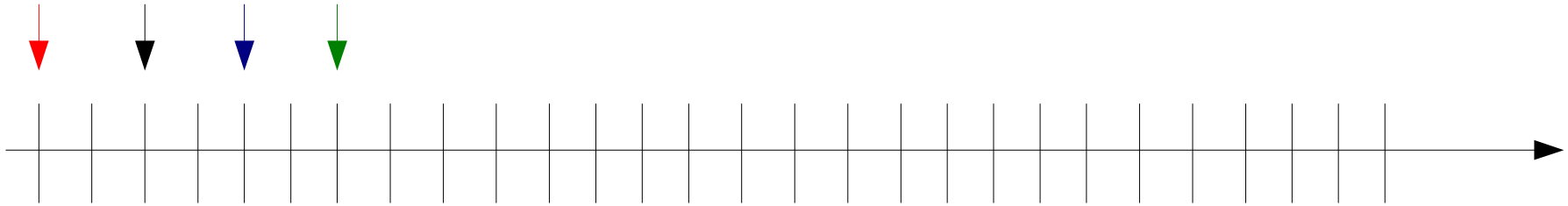
B
A
B
C
D
B
C
D
B
C
D
B
D
B

A terminé

C terminé

Round-Robin Exemple (2)

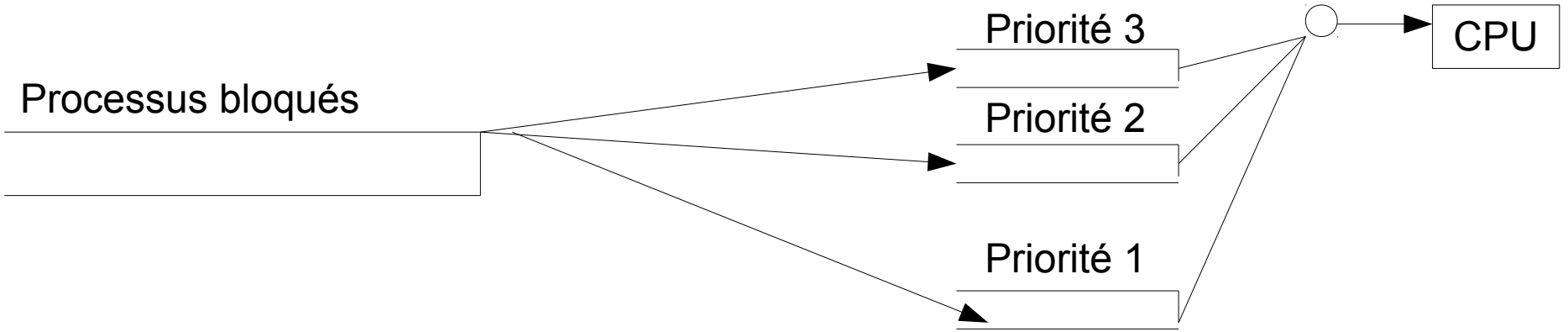
- Round-Robin scheduling
  - quantum fixé à trois unité CPU



Time	Arrivée processus	CPU execute	
0	A(3unités CPU)	A	
1	B(6unités CPU)	A	
2	C(3unités CPU)	A	
3	D(5unités CPU)	B	
4		B	
5		B	
6		C	
7		C	C terminé
8		C	
9		D	
10		D	
11		D	
12		B	
13		B	
14		B	

Schedulers pour systèmes interactifs

- Priority scheduling
  - Associer une priorité à chaque processus
  - A chaque activation du scheduler, sélectionner le processus de plus haute priorité



- Avantage
  - Les processus « importants » s'exécutent plus rapidement que les processus normaux
- Inconvénient
  - Un processus « important » peut complètement empêcher l'accès au CPU pour d'autres processus

Scheduler pour systèmes temps réel

- Problème
  - Parfois, il est nécessaire qu'un processus soit terminé avant un instant précis
    - contrôle de processus industriels
    - digitalisation/compression d'informations multimedia
    - transmission d'informations multimedia
  - Comment ordonnancer des processus sachant que tous les processus ont des échéances strictes à respecter ?
    - processus périodiques
    - processus non-périodiques

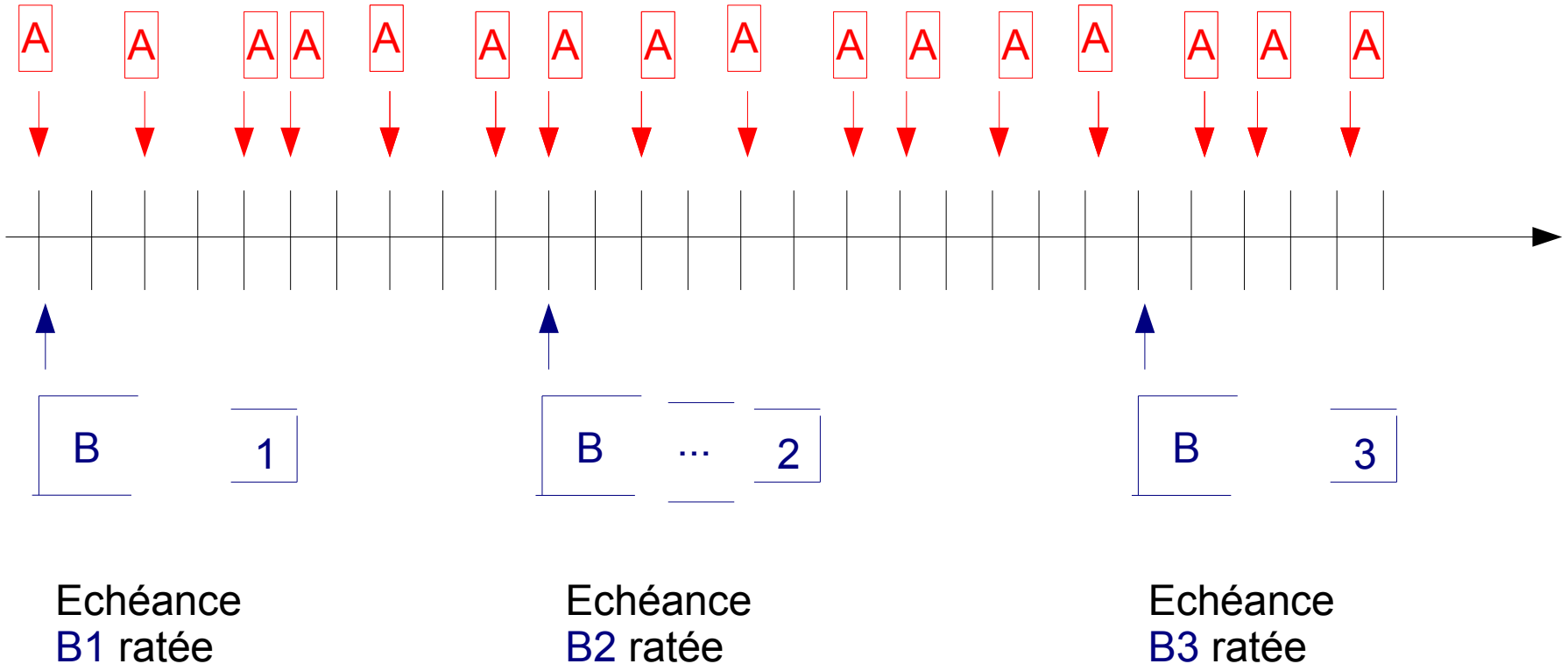
Scheduler temps réel

- Considérons un ensemble de i processus
  - période du ième processus : P i
  - temps CPU par période du i ème processus : Ci
  - Si la condition suivante n'est pas respectée, aucun scheduler ne pourra ordonnancer ces processus

$$\sum_{1}^{m} \frac{Ci}{Pi} \leqslant 1$$

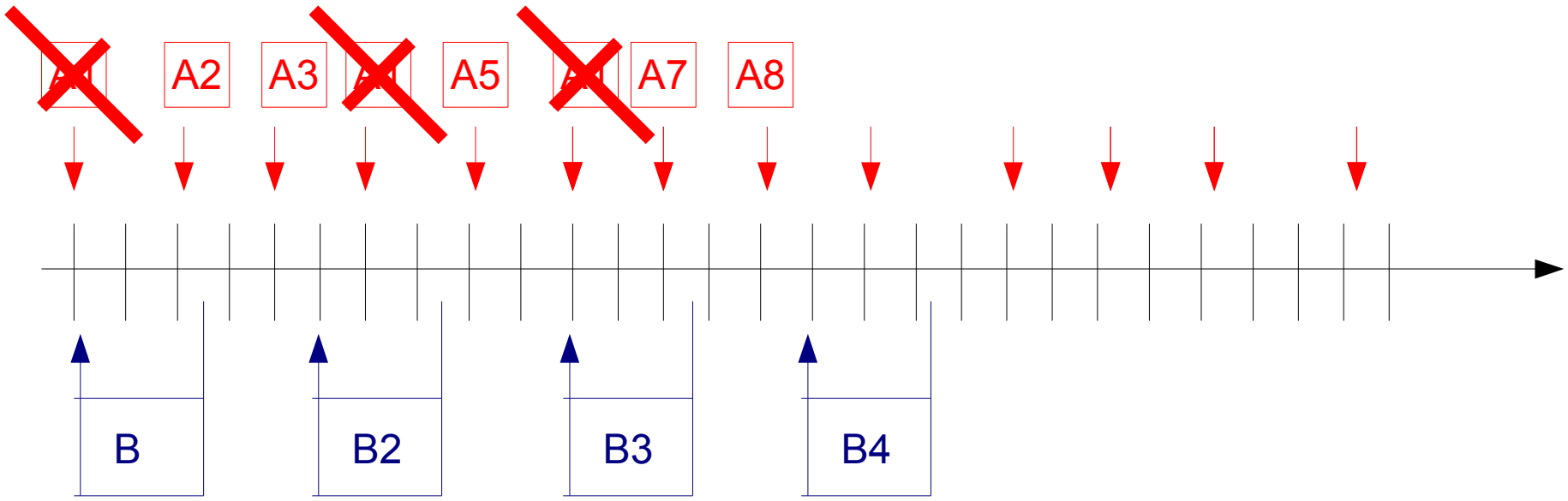
Scheduler temps réel Exemple

- Deux processus, A prioritaire
  - A : période = 20 et temps d'exécution = 10
  - B : période = 50 et temps d'exécution = 25



Scheduler temps réel Exemple(2)

- Deux processus, B prioritaire
  - A : période = 20 et temps d'exécution = 10
  - B : période = 50 et temps d'exécution = 25



Échéances de B respectées

Scheduler temps réel Rate Monotonic Scheduling

- Conditions d'utilisation
  - Processus périodiques qui terminent leur exécution durant chaque période
  - Pas d'interdépendance entre processus
  - Durant chaque période, chaque processus utilise le même temps CPU
  - Les processus non-périodiques n'ont pas d'échéances
  - La préemption d'un processus est instantannée
- Fonctionnement
  - scheduler à priorité
    - priorité de chaque processus = fréquence de ce processus
- Exemple
  - Processus P1 à 10Hz et processus P2 à 50Hz
    - P1 aura une priorité de 10
    - P2 aura une priorité de 50

Scheduler temps réel Earliest Deadline First

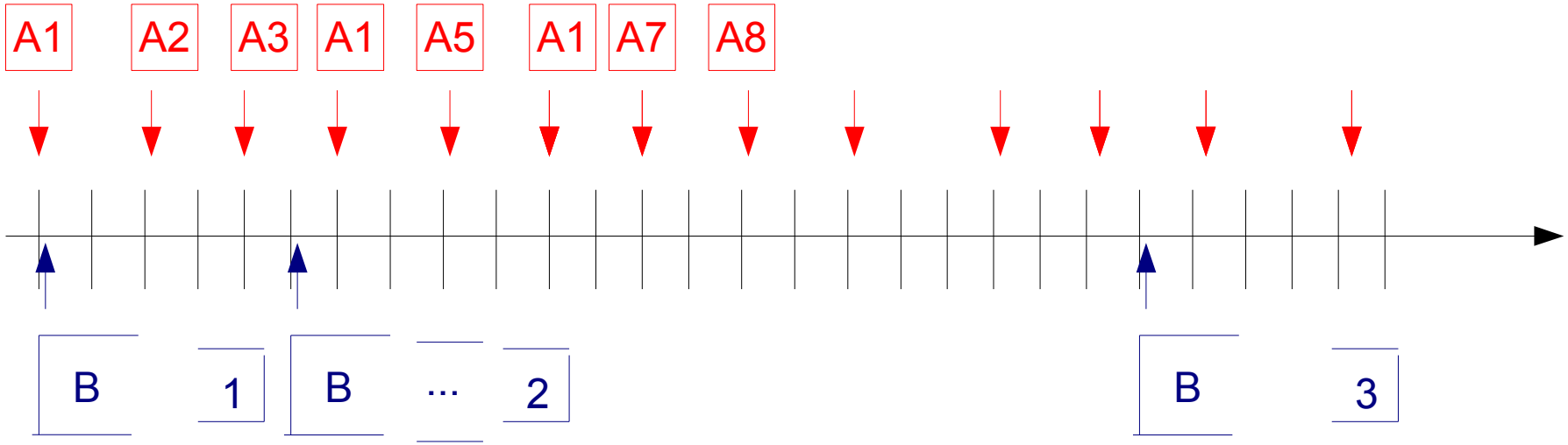
- Problème de RMS
  - Le choix de la priorité dans RMS ne prend pas en compte le temps d'exécution de chaque processus
  - RMS fonctionne si

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \leq m\left(2^{\frac{1}{m}} - 1\right)$$

- Earliest Deadline First
  - A chaque exécution, chaque processus annonce sa prochaine échéance
  - Le scheduler exécute toujours le processus qui a l'échéance la plus proche
    - si plusieurs processus ont la même échéance la plus proche, choix non-déterministe

Earliest Deadline First Exemple

- Deux processus, B prioritaire
  - A : période = 20 et temps d'exécution = 10
  - B : période = 50 et temps d'exécution = 25



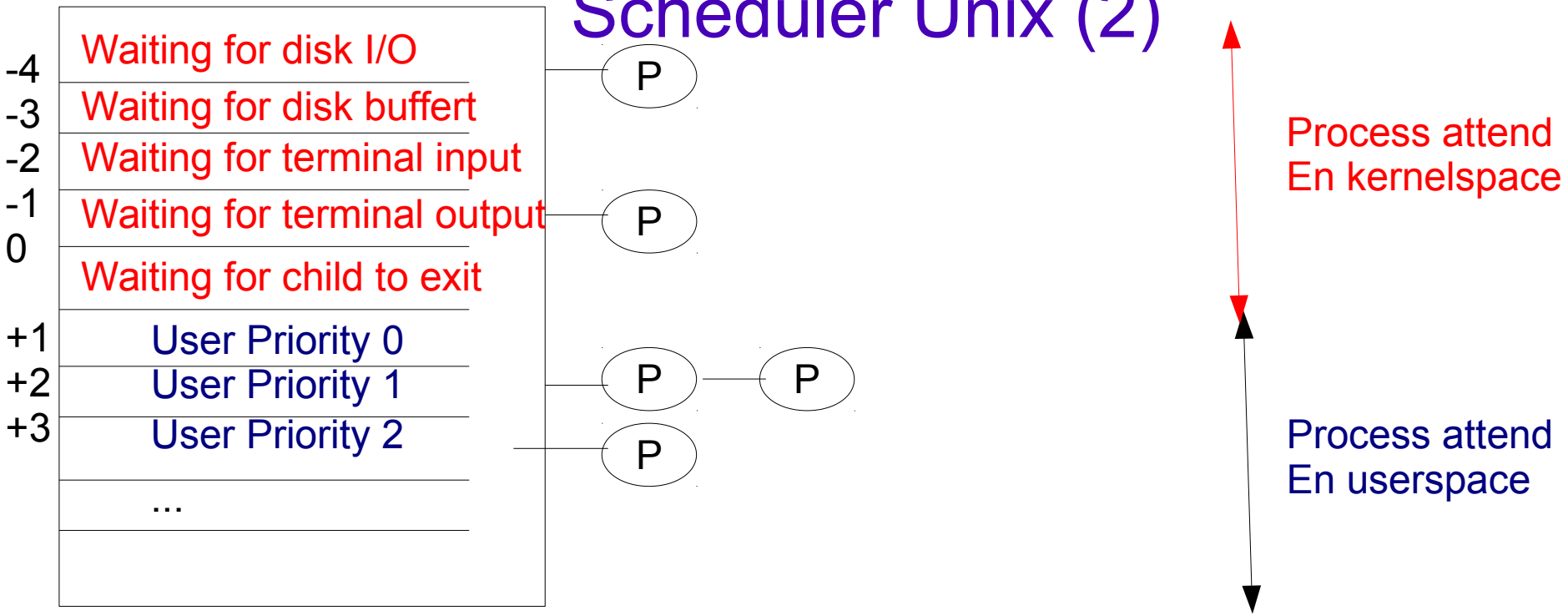
Échéances de B respectées

- EDF trouve la solution si $\sum_{1}^{m} \frac{Ci}{Pi} \leqslant 1$

Scheduler Unix

- Objectifs
  - Minimiser temps de réponse des processus interactifs
    - permettre aux processus qui font des I/O d'avancer rapidement
  - Permettre de favoriser certains processus par rapport à d'autres
    - certains processus systèmes sont plus importants que des processus utilisateurs
  - Permettre à chaque utilisateur d'indiquer que certains de ses processus sont peu importants
  - Assurer une certaine équité en permettant à tout processus de tourner

Scheduler Unix (2)



- Principe
  - A chaque activation, sélectionner processus avec priorité la plus faible
  - A la fin du quantum, remettre processus en fin de queue
  - Toutes les secondes, recalculer pour chaque processus

$$Priorité = CPU_{usage} + nice + base$$

Scheduler Linux

- Scheduler opère sur des threads
  - Classes de scheduling
    - Real-time FIFO
      - les threads dans cette classe ne sont pas préemptés par l'horloge, uniquement par processus de cette classe
    - Real-time round-robin
      - cette classe peut être préemptée par l'horloge
      - après avoir utilisé un quantum, le processus est remis en fin de queue
    - Timesharing
      - processus normaux, scheduling sur base de
        - priorité de chaque processus
        - temps CPU consommé par chaque processus
        - valeur nice de chaque processus