Architecture d’un système d’exploitation: Ordonnancement

Marc.perache@cea.fr

Bibliographie

- La programmation sous UNIX de Jean-Marie Rifflet
- Système d’exploitation d’Andrew Tanenbaum
- Inside the Linux 2.6 Completely Fair Scheduler https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/
- http://www.cse.iitm.ac.in/~chester/courses/16o_os/syllabus.html

Les processus

Correspond à l'exécution d'un programme

Un processus a son propre espace d'adressage:

- Instructions
- Données

Deux modes de fonctionnement:

- Mode utilisateur:
  - Instructions ordinaires
  - Données de son espace d'adressage
- Mode noyau:
  - Instructions privilégiées
  - Résultat d'un appel système

Les processus

La base de tout système d'exploitation UNIX

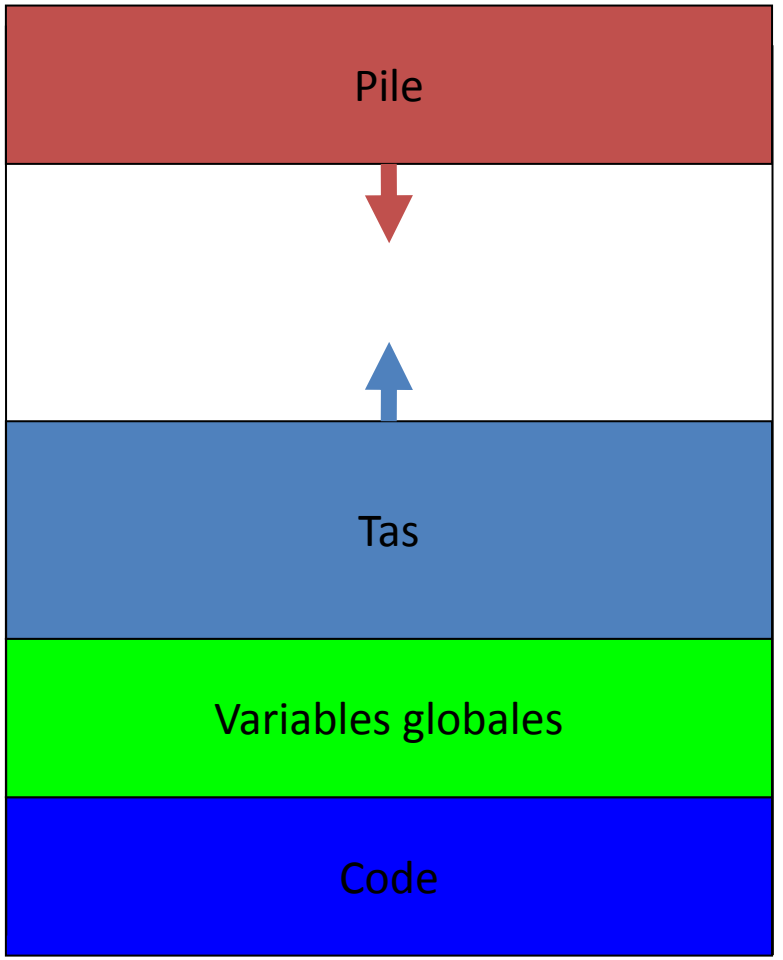
- Nécessaire pour le multitâche
- Nécessaire pour le multiutilisateur
- Assure un cloisonnement

Requiert

- Pagination mémoire
  - TLB
- Primitive de synchronisations
  - test_and_set
  - compare_and_swap
  - …

Les processus

- Registres
  - Pointeur de pile
  - Pointeur d'instruction
  - Flottants
  - Entiers
- Système
  - Table des pages
  - Descripteurs de fichiers
  - Statistiques
  - Signaux
  - Priorité
  - …



Primitives générales: identification

Include de base #include <unistd.h>

Identité du processus et celle de son père:

- Identité du processus: pid_t getpid(void);
- Identité du processus père: pid_t getppid(void);

Identité de l'utilisateur:

- Propriétaire réel (utilisateur qui lance le processus): uid_t getuid(void);
- Propriétaire effectif (utilisateur relatif aux fichiers): uid_t geteuid(void);
- Groupe propriétaire réel: gid_t getgid(void);
- Groupe propriétaire effectif: gid_t getgid(void);

Primitives générales: statistiques

Statistiques de temps: #include <sys/times.h>
clock_t times(struct tms* buf);

Processus courant

- clock_t tms_utime: nombre de tics d'horloge en mode utilisateur
- clock_t tms_stime: nombre de tics d'horloge en mode système

Processus fils et attendus (non zombies)

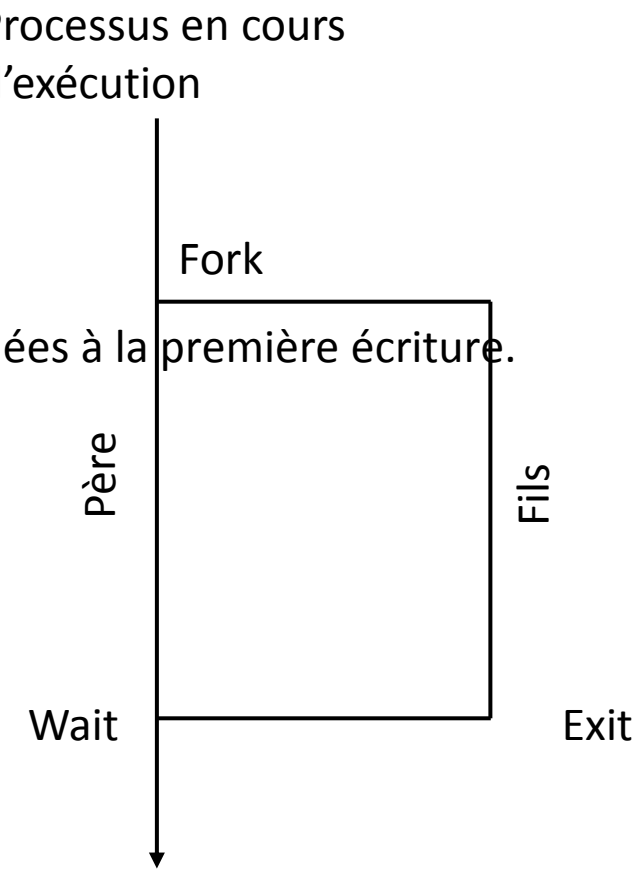
- clock_t tms_cutime: nombre de tics d'horloge en mode utilisateur des processus fils et attendus
- clock_t tms_cstime: nombre de tics d'horloge en mode système des processus fils et attendus

Danger: ne correspond pas au temps réel écoulé!

Les processus

Création
- Appel système fork
- Clone un processus existant
- Mécanisme de copy_on_write
  - Toutes les pages du processus fils sont dupliquées à la première écriture.

Destruction
- Appel exit ou fin du main
- Erreur (erreur de segmentation,…)
- Signal type SIG_KILL



Fork

Synopsis

```
#include <unistd.h>
pid_t fork(void);
```

Description

fork crée un processus fils qui diffère du processus parent uniquement par ses valeurs PID et PPID et par le fait que toutes les statistiques d'utilisation des ressources sont remises à zéro. Les verrouillages de fichiers, et les signaux en attente ne sont pas hérités.

Sous Linux, fork est implementé en utilisant une méthode de copie à l'écriture. Ceci consiste à ne faire la véritable duplication d'une page mémoire que lorsqu'un processus en modifie une instance. Tant qu'aucun des deux processus n'écrit dans une page donnée, celle-ci n'est pas vraiment dupliquée. Ainsi les seules pénalisations induites par fork sont le temps et la mémoire nécessaires à la copie de la table des pages du parent ainsi que la création d'une structure de tâche pour le fils.

Fork

Valeur Renvoyée

En cas de succès, le PID du fils est renvoyé au processus parent, et 0 est renvoyé au processus fils. En cas d'échec -1 est renvoyé dans le contexte du parent, aucun processus fils n'est créé, et errno contient le code d'erreur.

Erreurs

ENOMEM: Impossible d'allouer assez de mémoire pour copier la table des pages du père et d'allouer une structure de tâche pour le fils.
EAGAIN: Impossible de trouver un emplacement vide dans la table des processus.

Fork: exemple

```
pid_t pid;
pid = fork ();
if (pid > 0) {
/* Processus père */
} else if (pid == 0) {
/* Processus fils */
} else {
/* Traitement d'erreur */
}
```

Fork en détail

fork() crée un nouveau processus en dupliquant le processus appelant. Le nouveau processus, que l'on nomme processus fils, est l'exacte duplication du processus appelant, nomme processus parent, excepte pour les points suivants :

Le fils a son propre identifiant de processus (PID). Ce PID est unique et ne correspond a aucun autre identifiant de groupe de processus existant (setpgid(2)).
Le PPID (Parent Process ID) du fils est identique au PID du parent.
Le fils n'hérite pas des verrouillages mémoire de son parent (mlock(2), mlockall(2)).
Les statistiques d'utilisation des ressources de processus (getrusage(2)) et les compteurs de temps CPU (times(2)) sont réinitialises dans le fils.

Fork en détail

L'ensemble des signaux en attente pour le fils est initialement vide (sigpending(2)).
Le fils n'hérite pas des ajustements de semaphore de son parent (semop(2)).
Le fils n'hérite pas des verrouillages d'enregistrement de son parent (fcntl(2)).
Le fils n'hérite pas des temporisateurs de son parent (setitimer(2) alarm(3), timer_create(3)).
Le fils n'hérite pas des opérations d'E/S asynchrones en cours de son parent (aio_read(3), aio_write(3)).

Fork en détail

Les attributs de processus de la liste précédente sont tous spécifies dans POSIX.1-2001. Le parent et le fils différent également par les attributs de processus suivant, spécifiques a Linux :

Le fils n'hérite pas des notifications de changement de repertoire (dnotify) de son parent (voir la description de F_NOTIFY dans fcntl(2)).
L'attribut PR_SET_PDEATHSIG de prctl(2) est réinitialise de sorte que le fils ne reçoive pas de signal lorsque son parent se termine.
Le signal de terminaison du fils est toujours SIGCHLD (voir clone(2)).

Fork en détail

Noter les points suivants :

Le processus fils est crée avec un simple thread -- celui qui a appelé fork(2). L'espace entier d'adresses virtuelles du parent est répliqué dans le fils. Cela inclut les états des mutexes, les variables de condition et autres objets pthreads ; l'utilisation de pthread_atfork(3) peut être préférable afin de gérer les problèmes qui pourraient être provoqués.

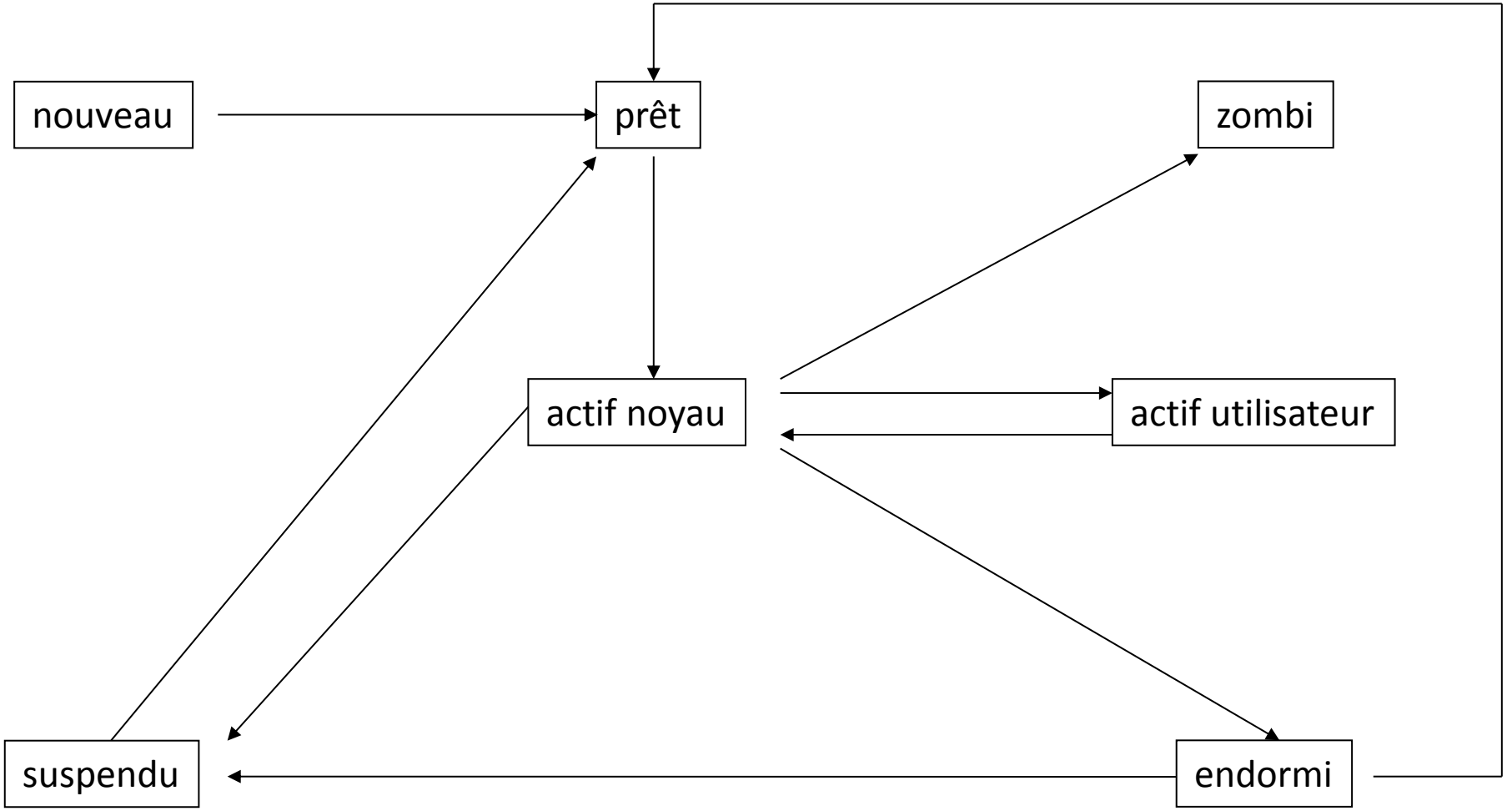
Le fils hérite d'une copie de l'ensemble des descripteurs de fichiers ouverts du parent. Chaque descripteur de fichier du fils fait référence à la même description de fichier ouvert (voir open(2)) que le descripteur de fichier correspondant du parent.

Cela signifie que les deux descripteurs partagent les attributs d'état de fichier ouvert, la position de la tète de lecture et les attributs d'E/S pilotées par signaux (voir la description de F_SETOWN et F_SETSIG dans fcntl(2)).

Fork en détail

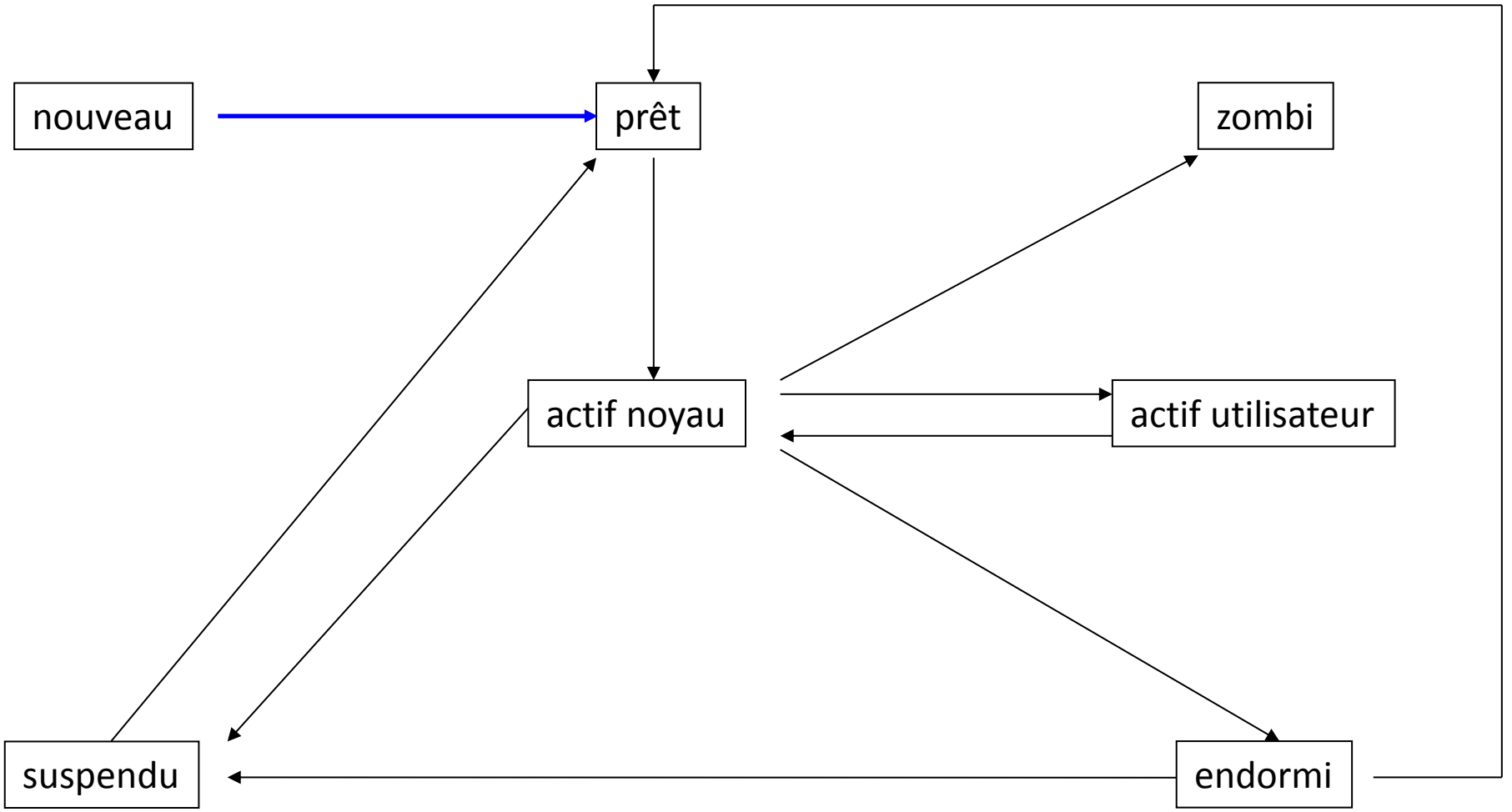
Le fils hérite d'une copie de l'ensemble des descripteurs de files de messages ouverts (voir mq_overview(7)). Chaque descripteur dans le fils fait référence à la même description de file de messages ouverte que le descripteur de file de message ouvert dans le parent. Cela signifie que les deux descripteurs partagent les mêmes drapeaux (mq_flags).

Les états d’un processus



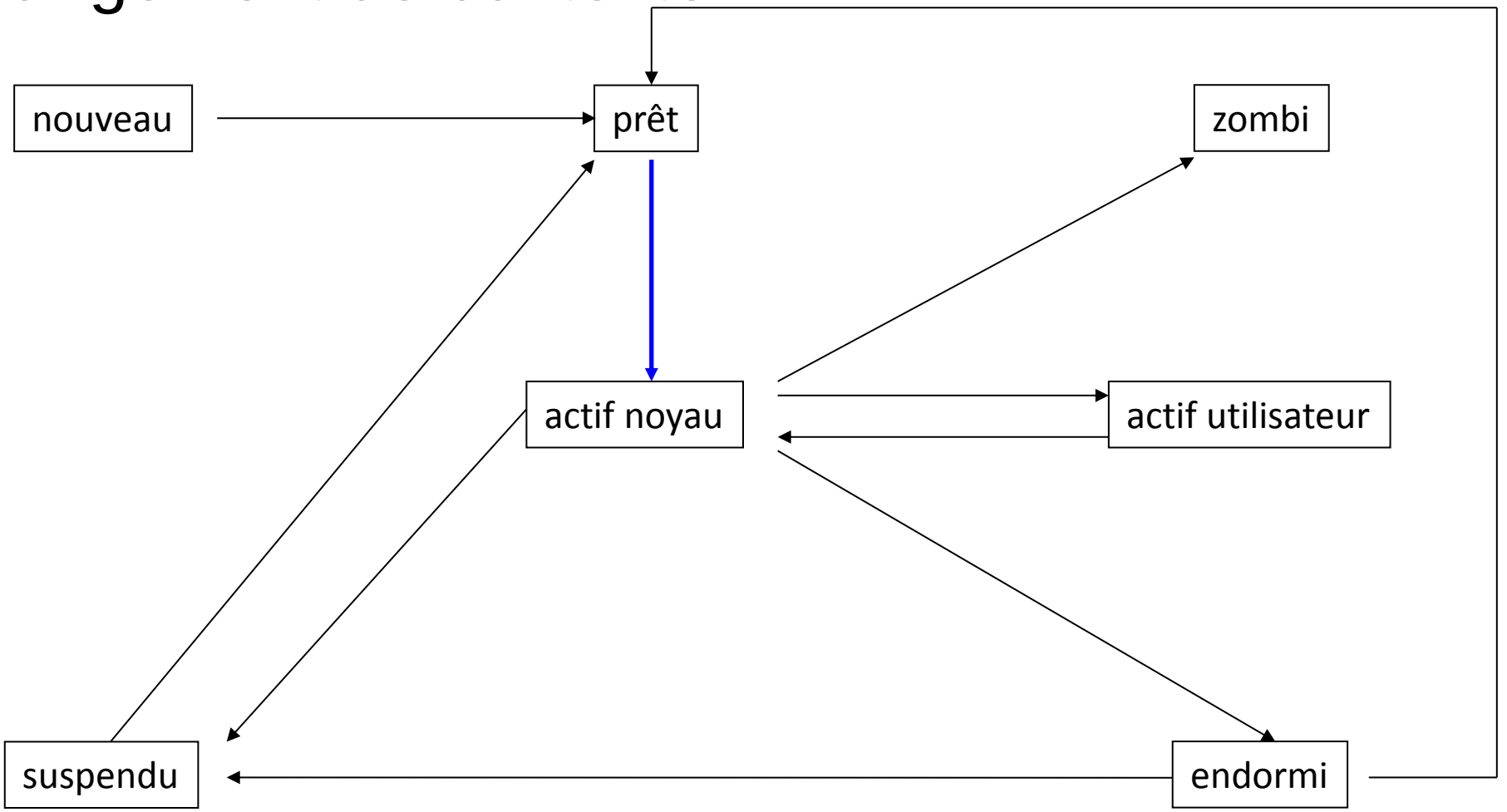
Les états d’un processus

Le processus a acquis les ressources nécessaires à son exécution



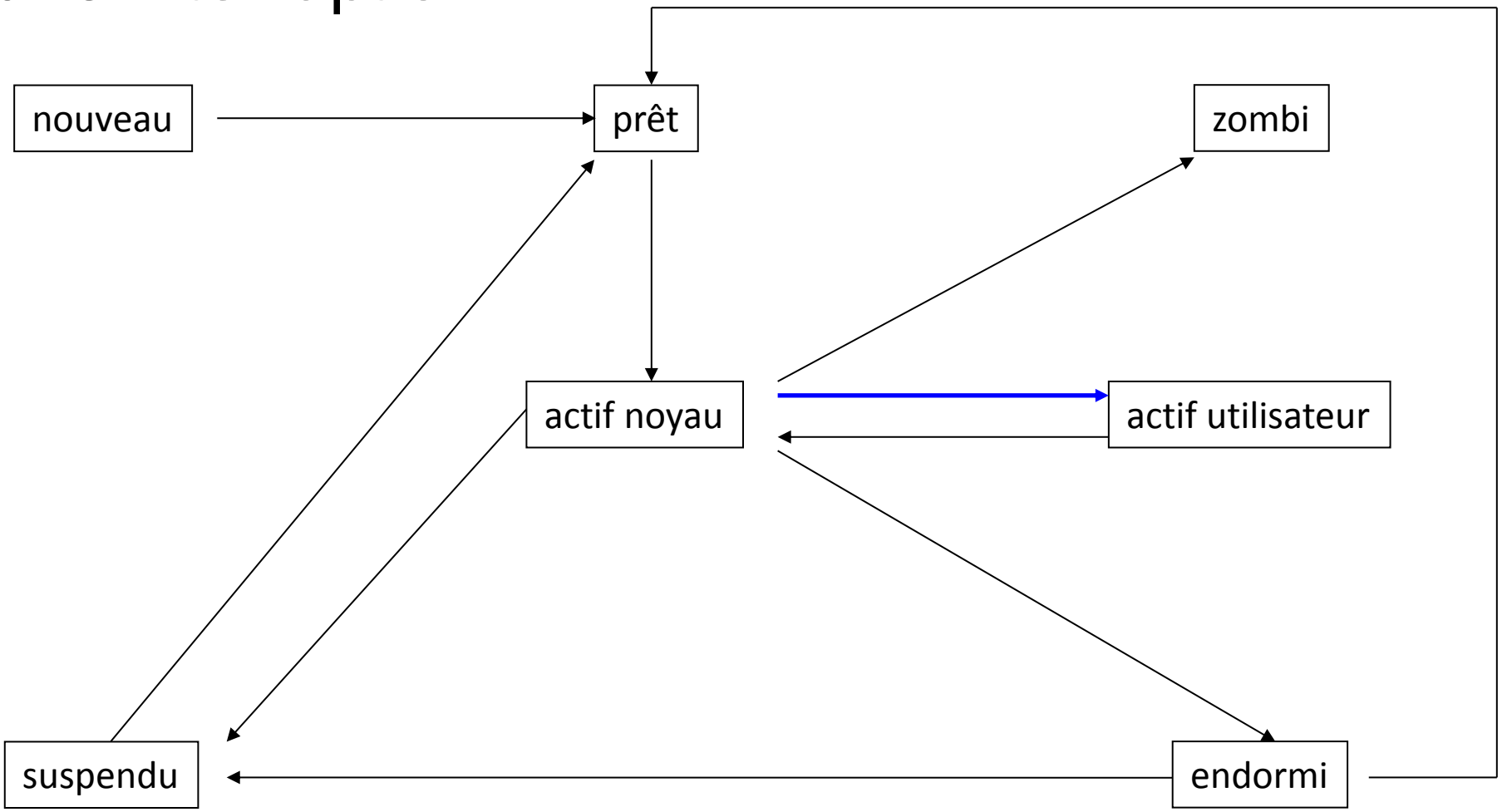
Les états d'un processus

Vient d'être élu par l'ordonnanceur: il y a alors changement de contexte



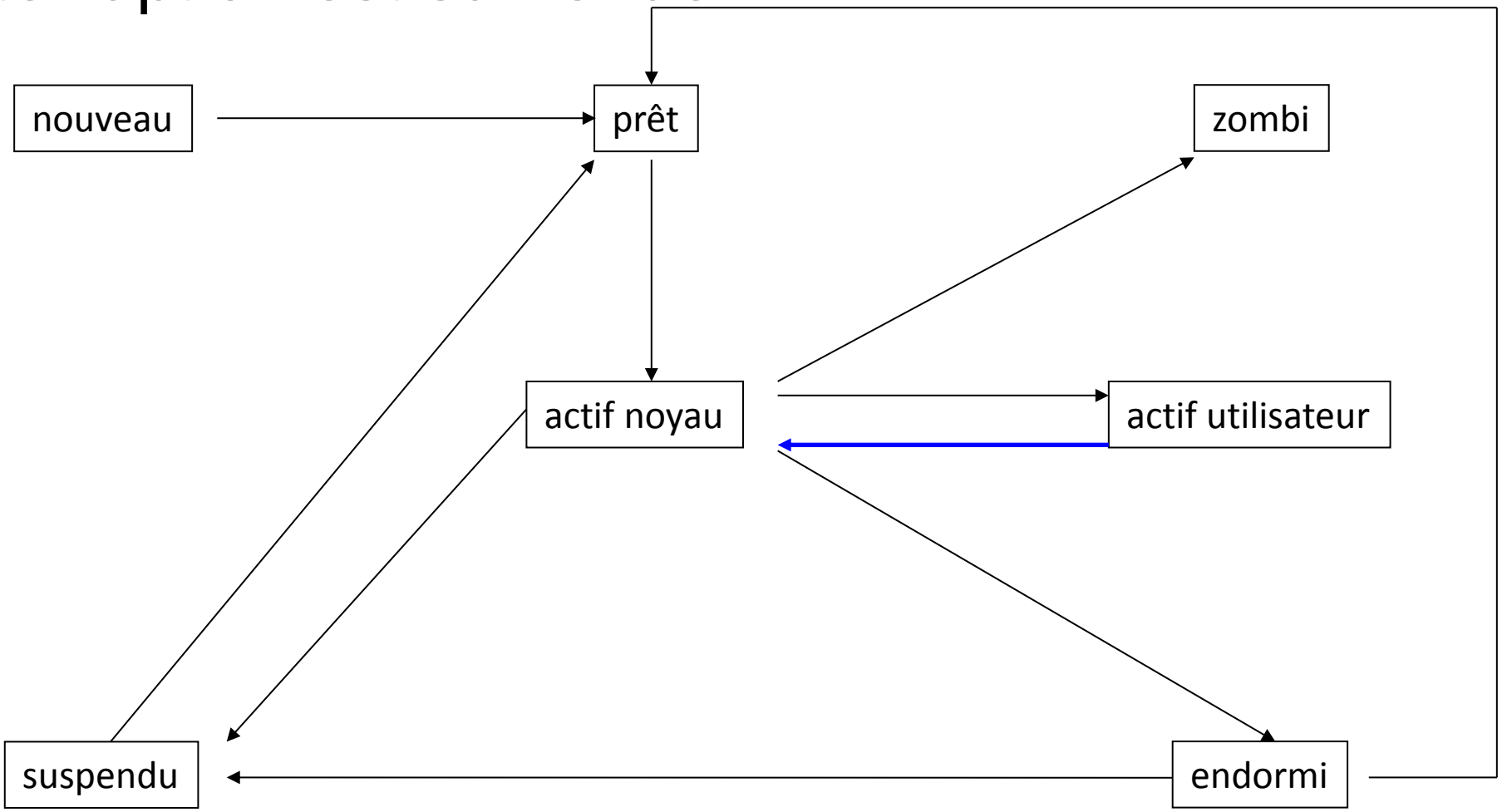
Les états d’un processus

Le processus revient d’un appel système ou d’une intérruption



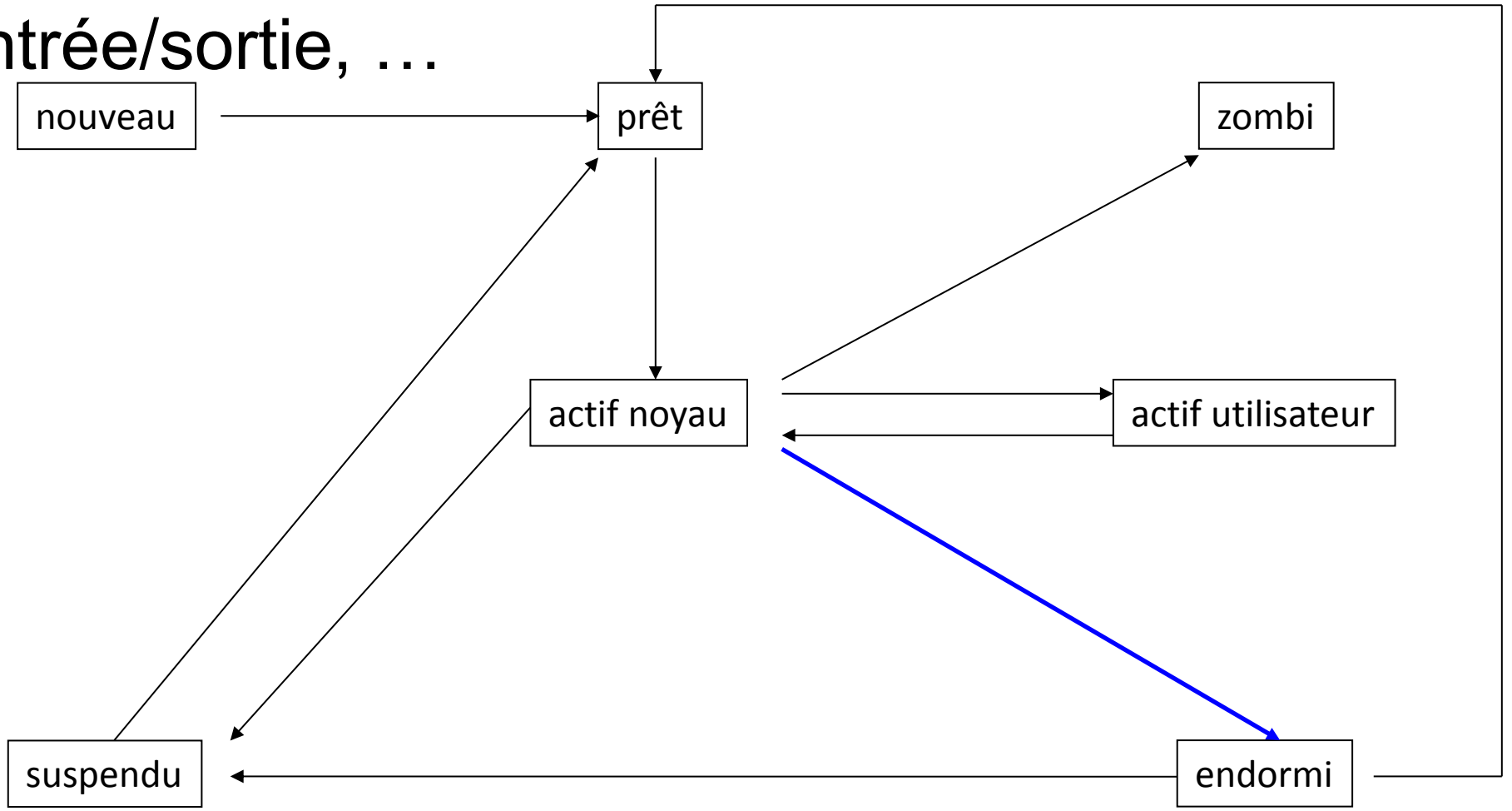
Les états d’un processus

Le processus a réalisé un appel système ou une interruption est survenue



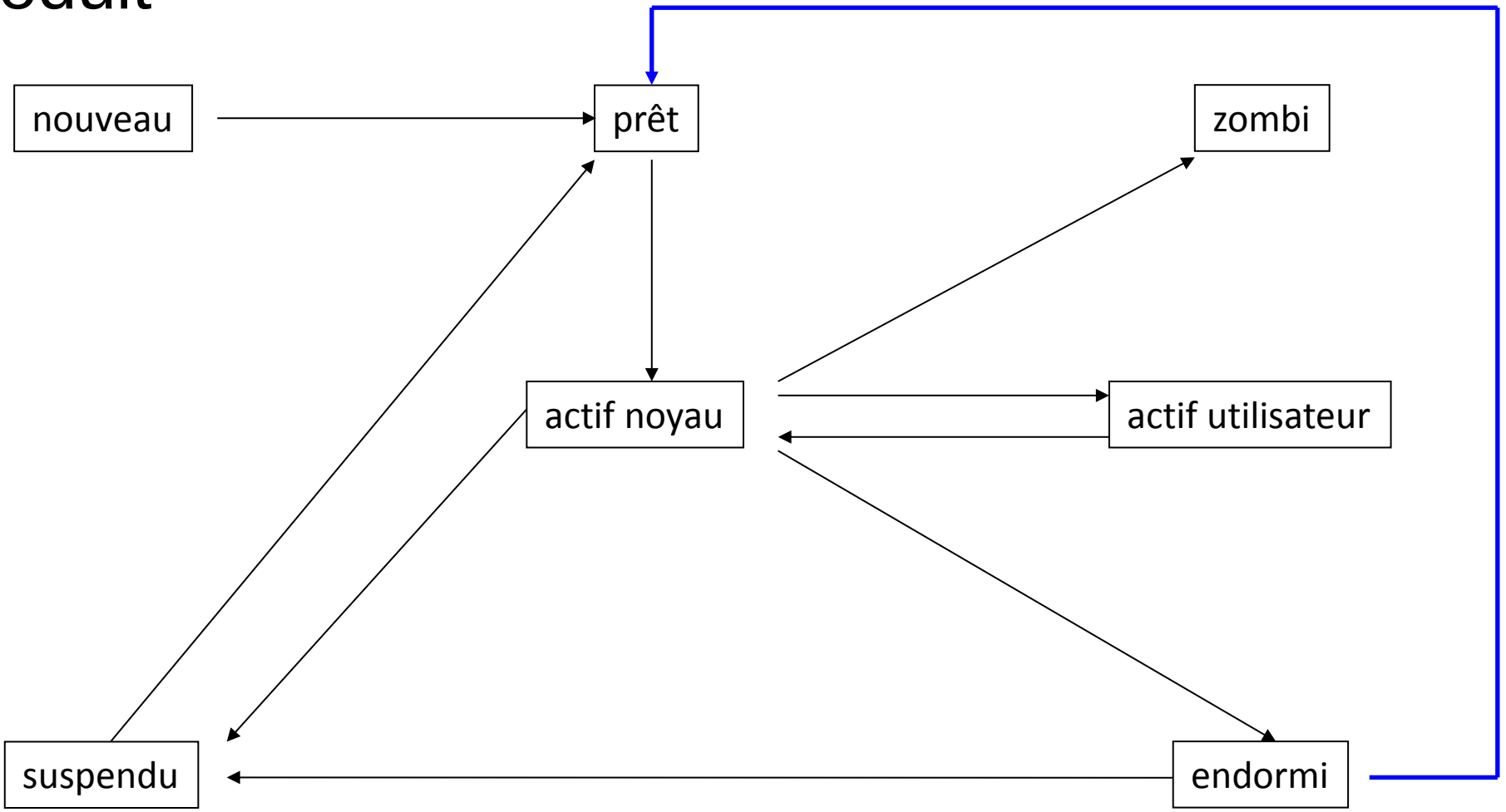
Les états d'un processus

Le processus se met en attente d'un événement: appel système bloqué, entrée/sortie, …



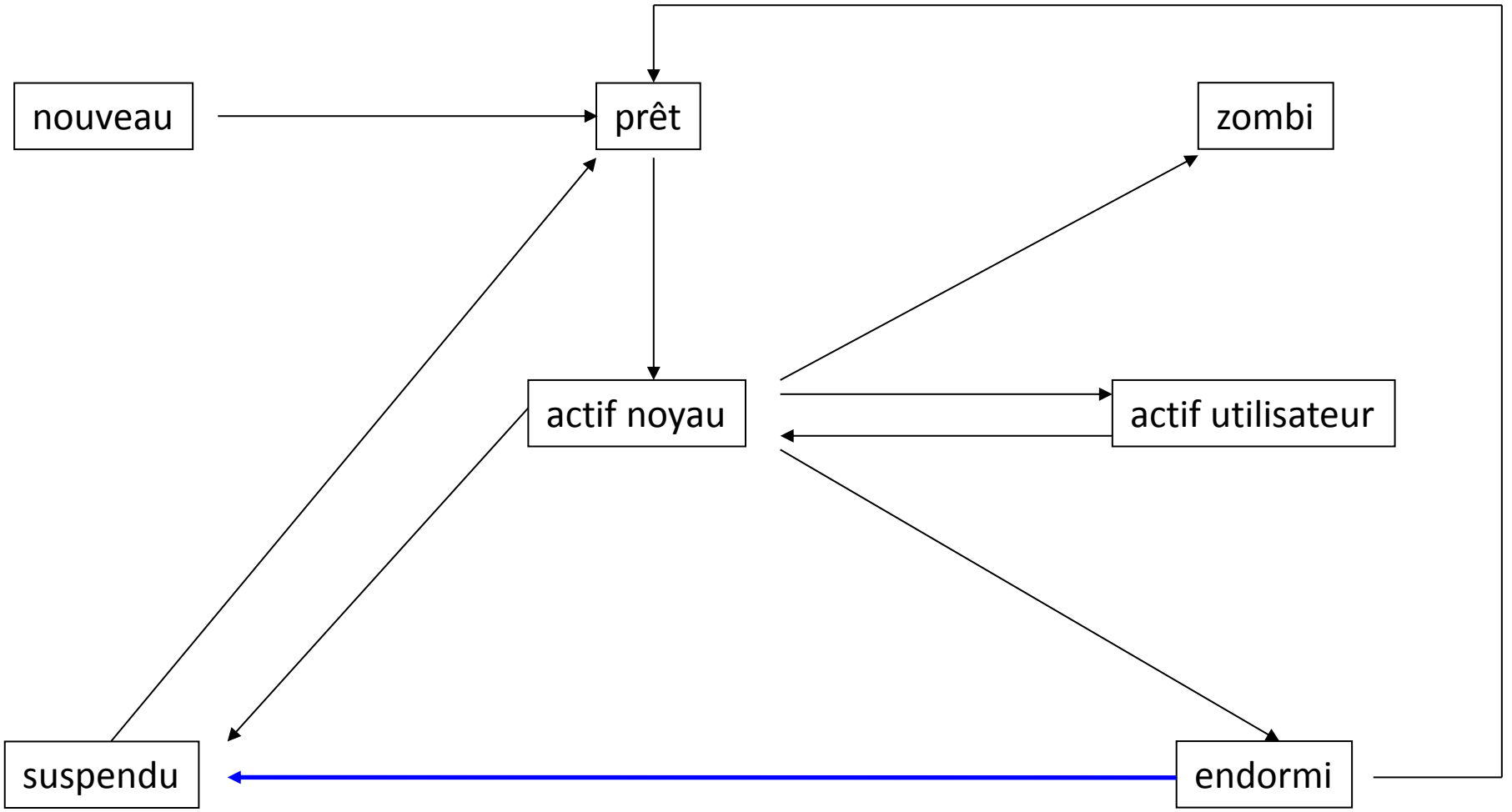
Les états d’un processus

L’événement attendu par le processus s’est produit



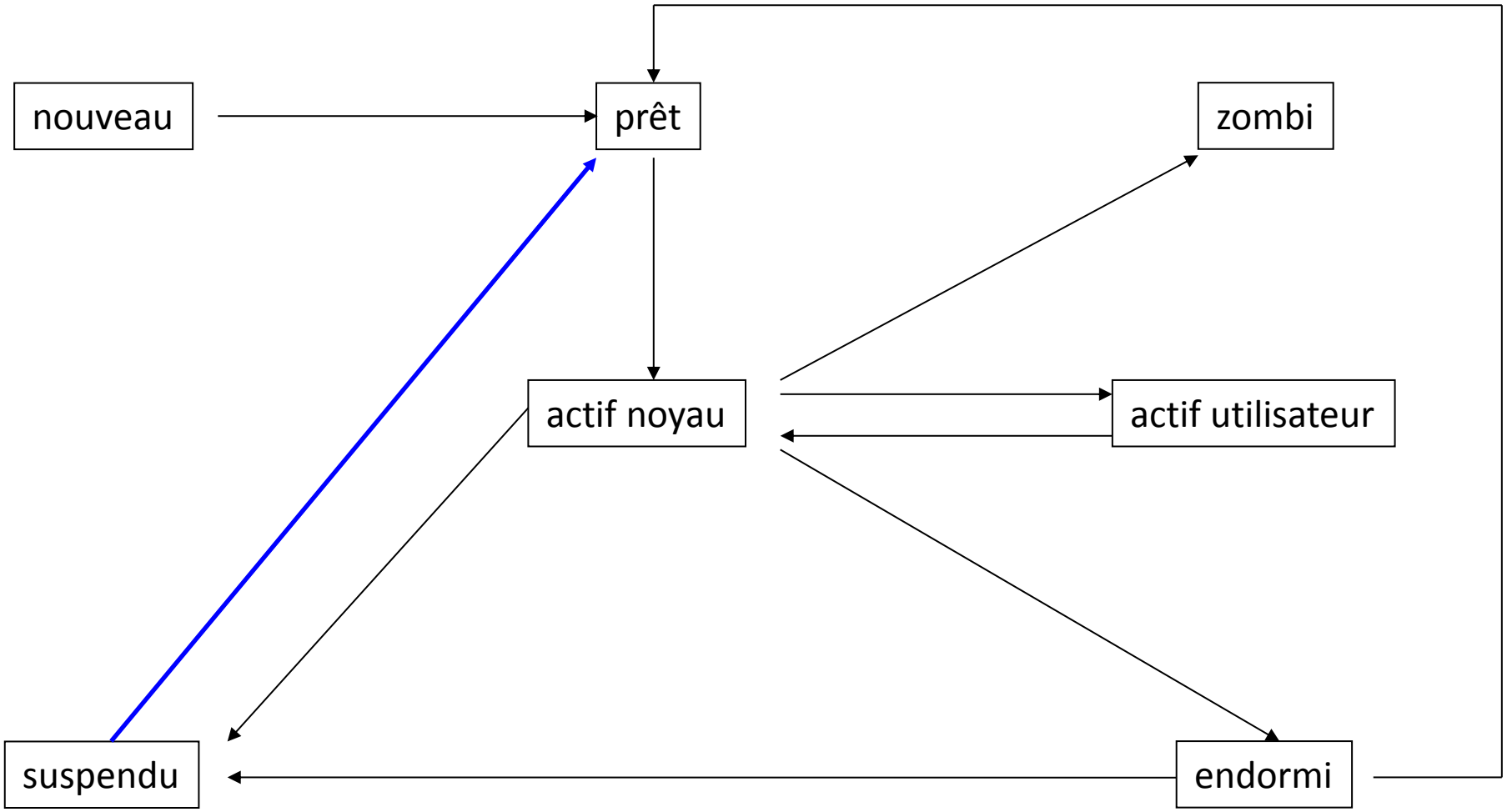
Les états d’un processus

Signal SIGSTOP ou SIGTSTP



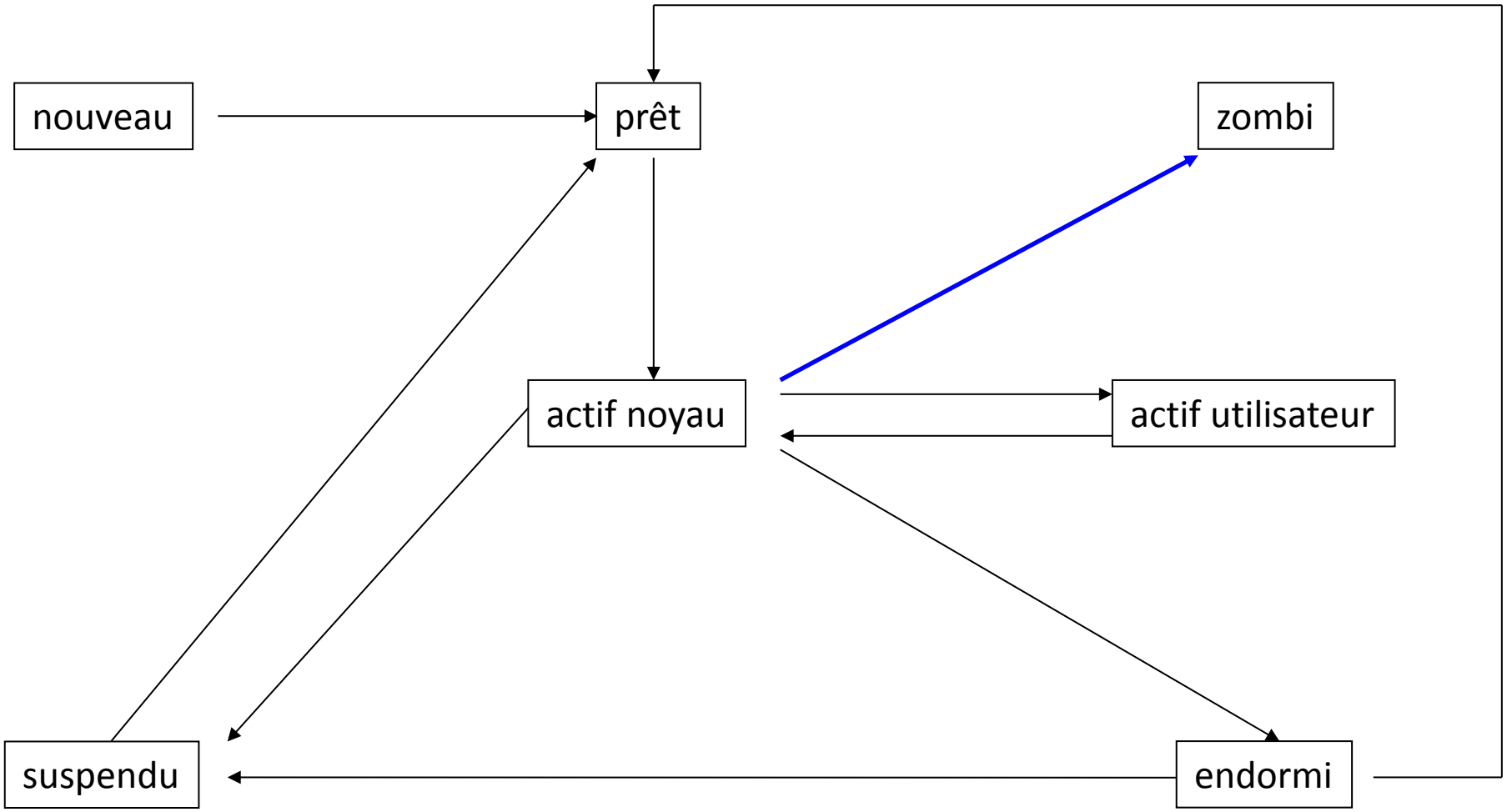
Les états d’un processus

Réveil du processus par le signal SIGCONT



Les états d’un processus

Le processus se termine



Wait

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int* pointer_status);
```

Le processus ne possède aucun fils, la primitive renvoie -1 et errno a la valeur ECHILD.
Le processus possède au moins un fils zombi, la primitive renvoie l'identité du fils et pointer_status fournit des informations sur la terminaison.
Le processus appelant est bloqué jusqu'à:

- Un de ces fils ce termine
- L'appel système est interrompu par un signal. Dans ce cas, la primitive renvoie -1 et errno EINTR.

Waitpid

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int* pointer_status, int options);
```

Sélection du processus attendu :

- pid<-1: tout processus fils dans le groupe |pid|
- pid=-1: tout processus fils
- pid=0: tout processus du même groups que l'appelant
- pid>0: processus fils d'identité pid

Valeur de retour:

- -1 erreure
- 0 echec: processus pid ni terminé ni stoppé en mode non-bloquant
- Le numéro du processus fils pris en compte

Execve: Exécuter un programme

Synopsis

#include <unistd.h>

int execve (const char *fichier, char * constargv [], char * constenvp[]);

Description

execve() exécute le programme correspondant au fichier. Celui-ci doit être un exécutable binaire ou bien un script commençant par une ligne du type "#! interpreteur [arg]". Dans ce dernier cas, l'interpréteur doit être indiqué par un nom complet, avec son chemin d'accès, et qui sera invoqué sous la forme interprèteur [arg] fichier.

Execve: Exécuter un programme

argv est un tableau de chaînes d'arguments passées au nouveau programme. envp est un tableau de chaînes, ayant par convention la forme cle=valeur, qui sont passées au nouveau programme comme environnement. argv ainsi que envp doivent se terminer par un pointeur NULL. Les arguments et l'environnement sont accessibles par le nouveau programme dans sa fonction principale, lorsqu'elle est définie comme int main (int argc, char * argv [], char * envp []).

En cas de réussite, execve() ne revient pas à l'appelant, et les segments de texte, de données, ainsi que la pile du processus appelant sont remplacés par ceux du programme chargé. Le programme invoqué hérite du PID du processus appelant, et de tous les descripteurs de fichiers qui ne possèdent pas le drapeau Close-on-Exec. Les signaux en attente destinés au processus parent sont effacés. Les signaux prêts à être intercepté par le processus appelant reprennent leur comportement par défaut.

Execve: Exécuter un programme

Si l'on effectuait un ptrace(2) sur le programme appelant, un signal SIGTRAP est envoyé après la réussite de execve().

Si le bit Set-UID est positionné sur le fichier du programme, l'UID effectif du processus appelant est modifié pour prendre celui du propriétaire du fichier. De même, lorsque le bit Set-GID est positionné, le GID effectif est modifié pour correspondre à celui du groupe du fichier. Si l'exécutable est un fichier binaire a.out lié dynamiquement, et contenant des appels aux bibliothèques partagées, le linker dynamique de Linux ld.so(1) est appelé avant l'exécution, afin de charger les bibliothèques partagées nécessaires en mémoire, et d'effectuer l'édition des liens de l'exécutable.

Execve: Exécuter un programme

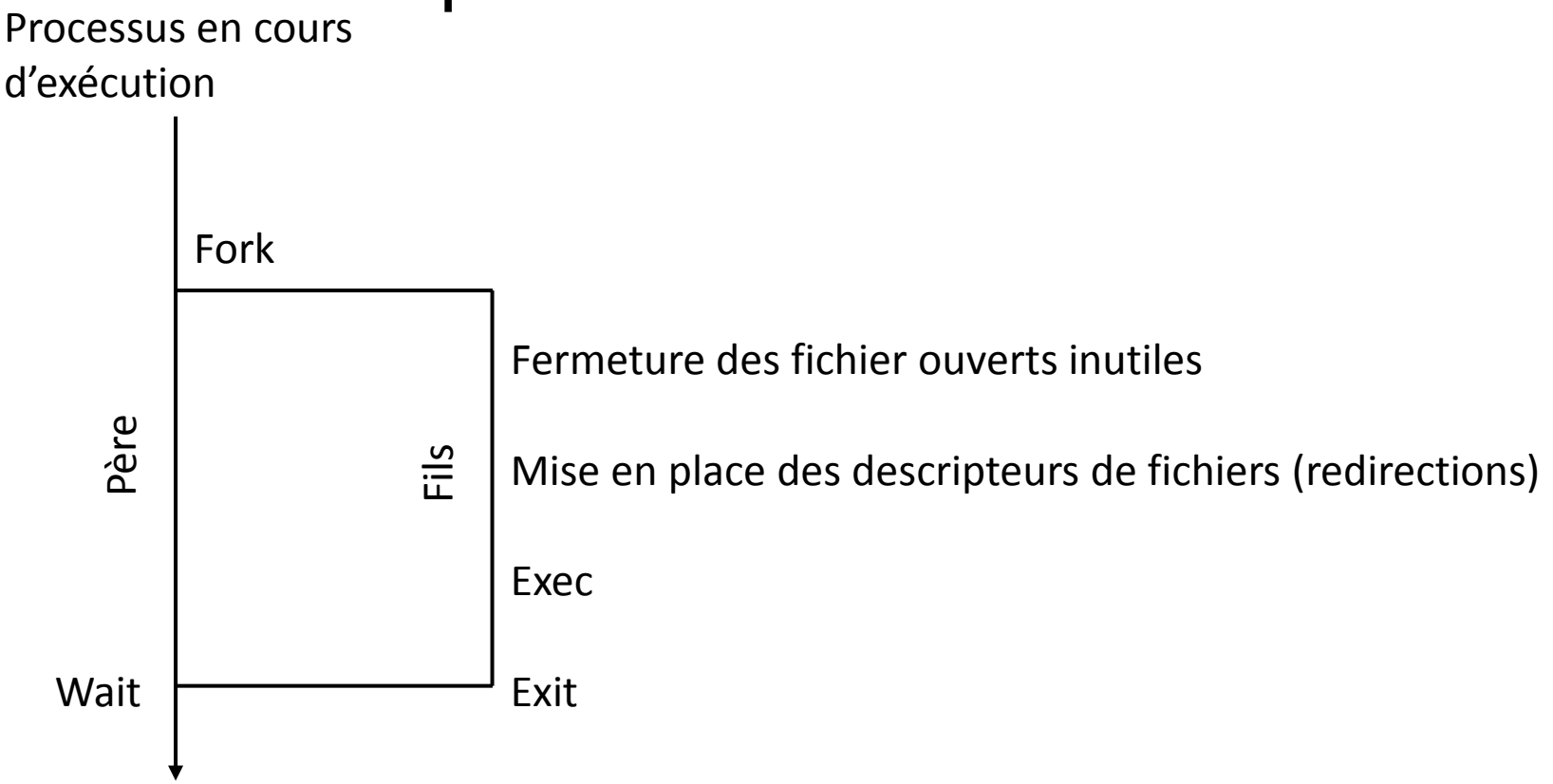
Si l'éxécutable est au format ELF lié dynamiquement, l'interprèteur indiqué dans le segment PT_INTERP sera invoqué pour charger les bibliothèques partagées. Cet interprèteur est généralement /lib/ld-linux.so.1 pour les fichiers binaires liés avec la libc Linux version 5, ou /lib/ld-linux.so.2 pour ceux liés avec la GNU libc version 2.

Valeur Renvoyée

En cas de réussite, execve() ne revient pas, en cas d'échec il renvoie -1 et errno contient le code d'erreur.

Fork/Exec

Utilisation classique

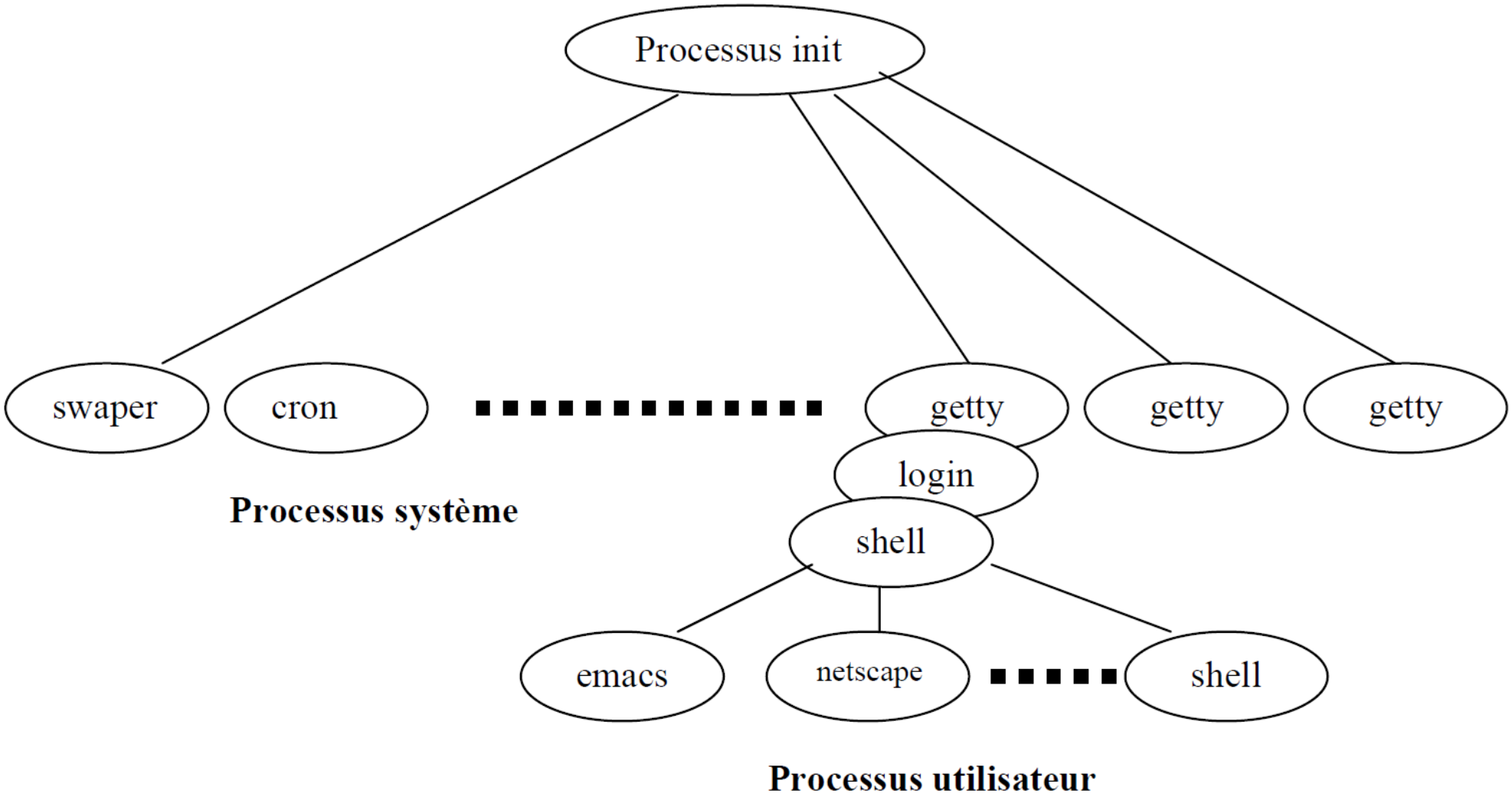


Exemple de fork/exec

```
main (int argc, *argv[])

  { int pid ;
     char *args[2] ;
     pid =  fork() ;
     if (pid ==0)
          { args[0] = "./a.out" ;          All executed by
            args[1] = NULL ;               child process
           i = execv("./aout", args) ;
           printf("OOOpppssss.\n") ;
          }
    else
          printf("Not a problem!\n") ; Executed by parent
    }
```

Arborescence de processus UNIX



Les IPC

Les IPC (Inter-Process Communications)

Signaux

Via kill(), raise().

Mémoire partagée

Via shmget, …: partage de mémoire entre deux processus existants.

Via mmap: partage de mémoire père/fils.

Synchronisations:

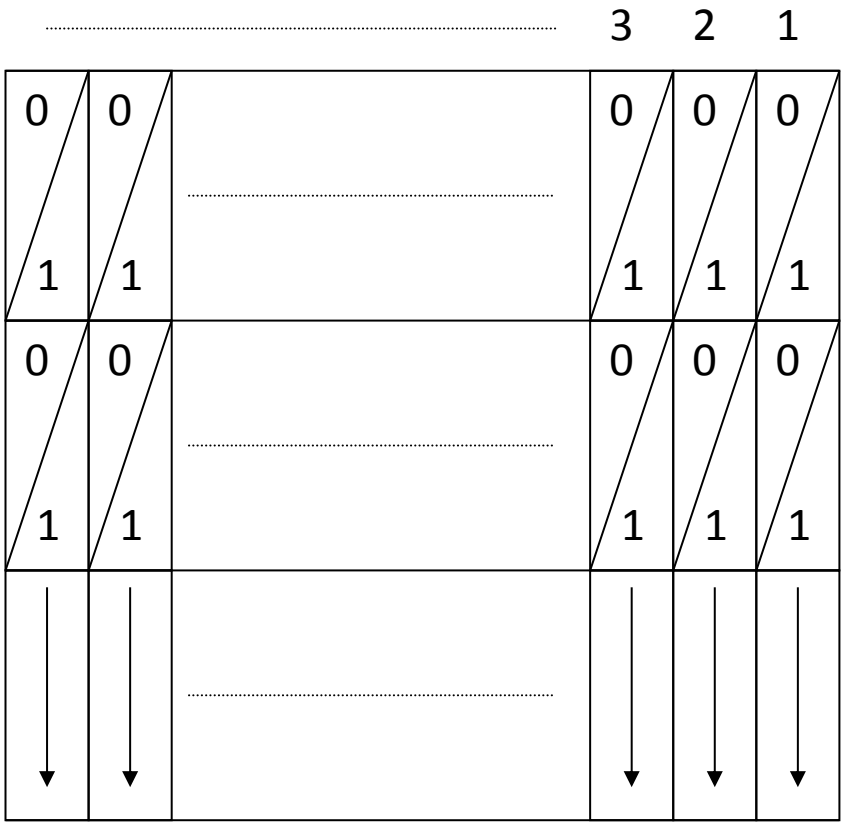
Mutex (nouvelle API POSIX commune avec les threads).

Sémaphores

Les signaux

Interruptions logicielles
Quelles informations un signal véhicule-t-il?
Comment les signaux sont-ils émis?
Quand les signaux sont-ils pris en compte?
Que fait un processus à la prise en compte d’un signal?

Les signaux



Indicateur des signaux pendants

Indicateur des signaux bloqués

Indicateur des comportements

Les signaux

Définitions:

Un signal est dit **pendant** s'il a été envoyé mais pas encore pris en compte

Un signal est dit **délivré** si le processus le prend en compte lors de son exécution.

Un signal peut être perdu: s'il est émis que le précédent signal du même identifiant est toujours pendant;

La prise en compte d'un signal entraîne l'exécution d'une fonction particulière

Toutes les structures et fonctions sont disponibles dans <signal.h>

Les signaux

SIGABRT	Terminaison anormal	A
SIGALRM	Fin de temporisation	T
SIGBUS	Erreur sur le bus	A
SIGCHLD	Envoyer au père en fin d'exécution	I
SIGCONT	Continuer l'exécution	C
SIGFPE	Erreur arithmétique	A
SIGHUP	Déconnexion d'un terminal	T
SIGILL	Instruction illégale	A
SIGINT	Ctrl-C	T
SIGKILL	Tuer un processus	T
SIGPIPE	Erreur écriture sur un tube sans lecteur	T
SIGQUIT	Caractère <QUIT> frappé au clavier	A
SIGSEGV	Violation de segment mémoire	A
SIGSTOP	Arrêter l'exécution	S
SIGTERM	Signal de Terminaison	T
SIGUSR1	Réservé à l'utilisateur	T
SIGUSR2	Réservé à l'utilisateur	T

T = terminaison; A = terminaison + core; I = ignore; S = processus stoppé; C = continuation

Les signaux

L'envoie de signaux:

int kill(pid_t pid, int sig);

pid > 0 processus d'identité pid

pid = 0 processus du même groupe

pid = -1 non POSIX, mais tous les processus sauf 0 et 1 pour système V

pid < -1 tous les processus du même groupe que |pid|

int raise(int sig);

Envoie le signal sig au processus courant

Les signaux

Le comportement à la prise en compte:

Comportement par défaut:

1. Terminaison du processus
2. Terminaison du processus et génération d'un fichier core
3. Signal ignoré
4. Suspension du processus
5. Continuation: reprise d'un processus stoppé et ignoré sinon

Fonction utilisateur:

Toute fonction utilisateur du type void handler(int sig) qui sera exécuté à la délivrance du signal correspondant

Les signaux

Manipulation des handlers

La structure sigaction

```
struct sigaction{
        void (*sa_handler)(); /* Handler de signal*/
        sigset_t sa_mask;     /* Signaux à bloquer */
        int sa_flags                /* options */
}
```

La primitive sigaction

int sigaction(int sig, const struct sigaction* p_action, struct sigaction* p_action_anc);

p_action nouveau handler de signal

p_action_anc ancien handler de signal

Exemple d’utilisation des handlers

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static void hdl (int sig, siginfo_t *siginfo, void *context)
{
		printf ("Sending PID: %ld, UID: %ld\n",
				(long)siginfo->si_pid, (long)siginfo->si_uid);
}

int main (int argc, char *argv[])
{
		struct sigaction act;

		memset (&act, '\0', sizeof(act));

		/* Use the sa_sigaction field because the handles has two additional parameters */
		act.sa_sigaction = &hdl;

		/* The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field, not sa_handler. */
		act.sa_flags = SA_SIGINFO;

		if (sigaction(SIGTERM, &act, NULL) < 0) {
			perror ("sigaction");
			return 1;
		}

		while (1)
			sleep (10);

		return 0;
}
```

Sémaphores

Les sémaphores POSIX permettent aux processus et aux threads de se synchroniser.

Un sémaphore est un entier dont la valeur ne peut jamais être négative. Deux opérations peuvent être effectuées : incrémenter la valeur du sémaphore de 1 (sem_post(3)), ou décrémenter la valeur du sémaphore de 1 (sem_wait(3)). Si la valeur courante est 0, une opération sem_wait(3) bloque jusqu'à ce que la valeur devienne strictement positive.

Les sémaphores POSIX sont de deux types : les sémaphores nommés et les sémaphores anonymes.

Sémaphores

Sémaphores nommés

Un sémaphore nommé est identifié par un nom de la forme /un_nom. Deux processus peuvent utiliser un même sémaphore nommé en passant le même nom à sem_open(3). La fonction sem_open(3) crée un nouveau sémaphore nommé ou en ouvre un existant. Après l'ouverture de ce sémaphore, il peut être utilisé avec sem_post(3) et sem_wait(3). Lorsqu'un processus a fini d'utiliser le sémaphore, il peut utiliser sem_close(3) pour le fermer. Lorsque tous les processus ont terminé de l'utiliser, il peut être supprimé du système avec sem_unlink(3).

Sémaphores

Sémaphores anonymes (sémaphores en mémoire)

Un sémaphore anonyme n'a pas de nom. Il est placé dans une région de la mémoire qui est partagée entre plusieurs threads (sémaphore partagée par des threads) ou processus (sémaphore partagée par des processus).
Un sémaphore partagé par des threads est placé dans une région de la mémoire partagée entre les threads d'un processus, par exemple une variable globale.
Un sémaphore partagé par des processus doit être placé dans une région de mémoire partagée (par exemple un segment de mémoire partagée System V créé avec semget(2), ou un objet de mémoire partagée POSIX créé avec shm_open(3)).
Avant son utilisation, un sémaphore anonyme doit être initialisé avec sem_init(3). Il peut ensuite être utilisé avec sem_post(3) et sem_wait(3). Lorsque le sémaphore n'est plus nécessaire, et avant que la mémoire où il est placé ne soit libérée, le sémaphore doit être détruit avec sem_destroy(3).

Les threads

Thread = processus léger

Éléments d'un thread:

- Une pile
- Un contexte: ensembles de registres

Éléments d'un processus multithread:

- Une table de pages
- Un ensemble de threads

Deux processus différents ont des espaces d’adressages différents

Deux threads différents d’un même processus ont le même espace d’adressage

Priority Based Scheduling Algorithms

Chester Rebeiro
IIT Madras

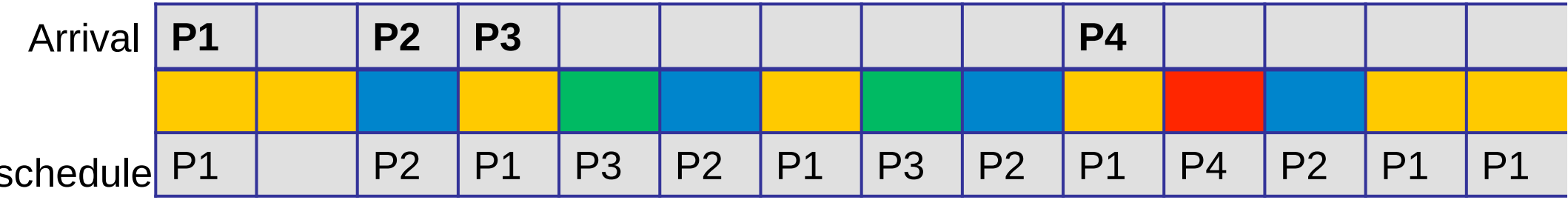
Relook at Round Robin Scheduling

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 1

Process P2 is a critical process while process P1, P3, and P4 are less critical

Process P2 is delayed considerably



Priorities

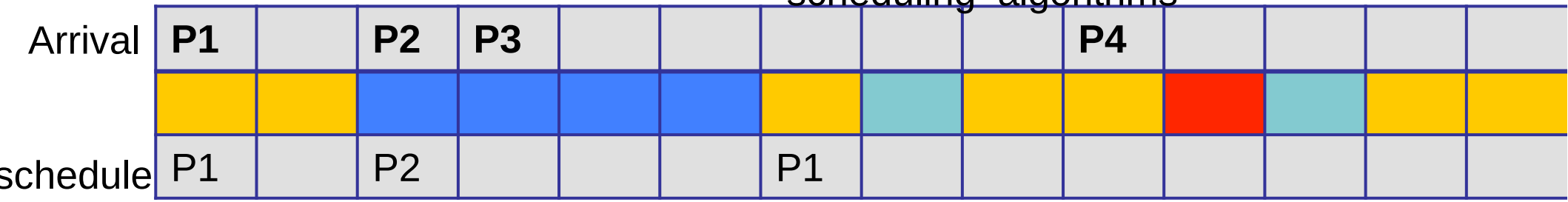
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 1

Process P2 is a critical process while process P1, P3, and P4 are less critical

We need a higher priority for P2, compared with the other processes

This leads to priority based scheduling algorithms



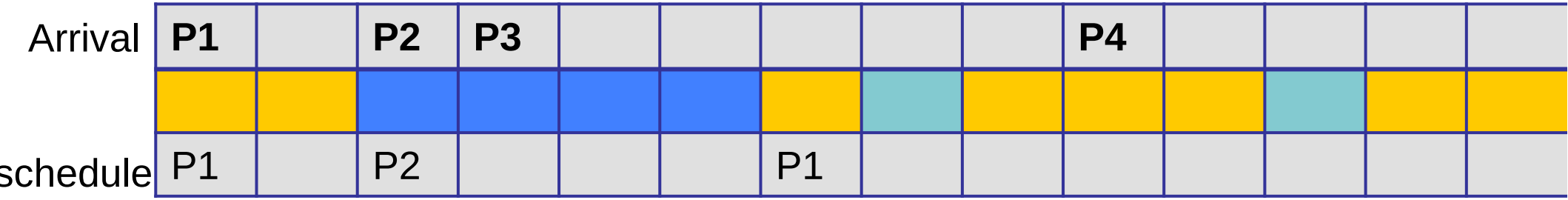
Starvation

Process	Arrival Time	Burst Time
P1	0	8
P2	2	4
P3	3	2
P4	9	1

Time slice = 1

Low priority process may never get a chance to execute.

P4 is a low priority process



Priority based Scheduling

- Priority based Scheduling
  - Each process is assigned a priority
    - A priority is a number in a range (for instance between 0 and 255)
    - A small number would mean high priority while a large number would mean low priority
  - Scheduling policy : pick the process in the ready queue having the highest priority
  - Advantage : mechanism to provide relative importance to processes
  - Disadvantage : could lead to starvation of low priority processes

Dealing with Starvation

- Scheduler adjusts priority of processes to ensure that they all eventually execute
- Several techniques possible. For example,
  - Every process is given a base priority
  - After every time slot increment the priority of all other process
    - This ensures that even a low priority process will eventually execute
  - After a process executes, its priority is reset

Priorities are of two types

- Static priority : typically set at start of execution
  - If not set by user, there is a default value (base priority)
- Dynamic priority : scheduler can change the process priority during execution in order to achieve scheduling goals
  - eg1. decrease priority of a process to give another process a chance to execute
  - eg.2. increase priority for I/O bound processes

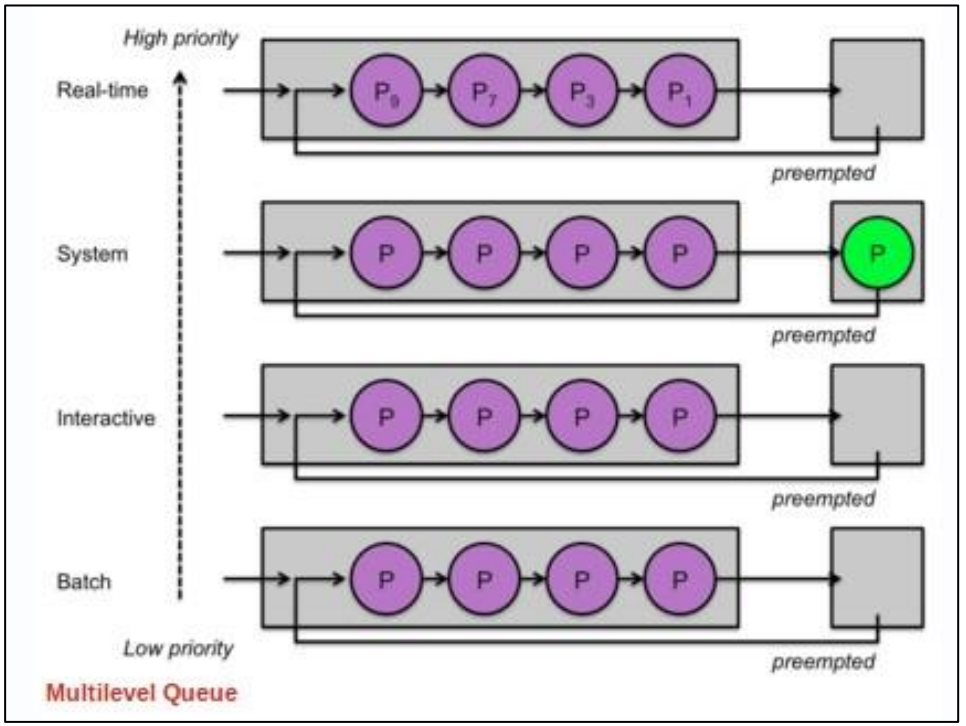
Priority based Scheduling with large number of processes

- Several processes get assigned the same base priority
  - Scheduling begins to behave more like round robin

Process	Arrival Time	Burst Time	Priority
P1	0	8	1
P2	2	4	1
P3	3	2	1
P4	9	1	1

Multilevel Queues

- Processes assigned to a priority classes
- Each class has its own ready queue
- Scheduler picks the highest priority queue (class) which has at least one ready process
- Selection of a process within the class could have its own policy
  - Typically round robin (but can be changed)
  - High priority classes can implement first come first serve in order to ensure quick response time for critical tasks



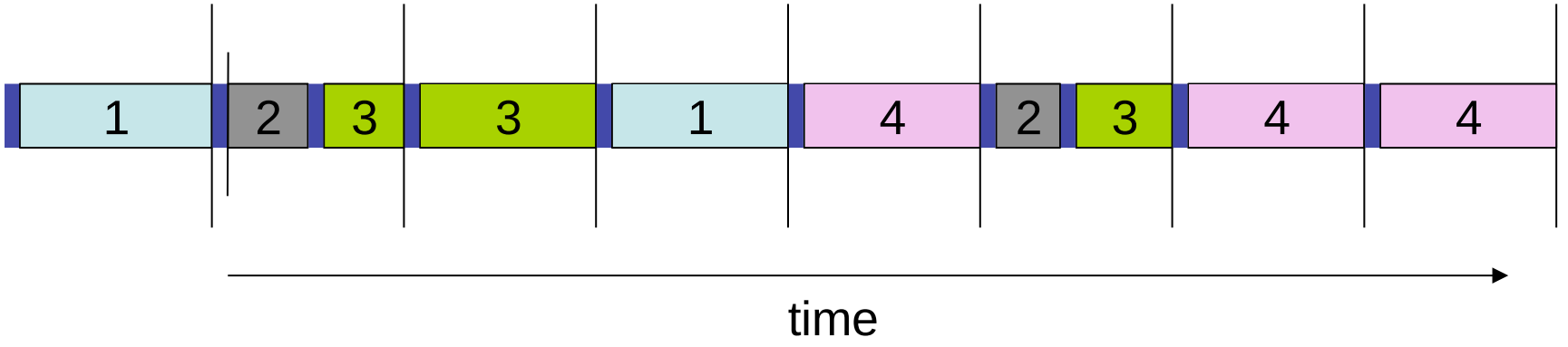
More on Multilevel Queues

- Scheduler can adjust time slice based on the queue class picked
  - I/O bound process can be assigned to higher priority classes with longer time slice
  - CPU bound processes can be assigned to lower priority classes with shorter time slices
- Disadvantage :
  - Class of a process must be assigned apriori

    (not the most efficient way to do things!)

Multilevel **feedback** Queues

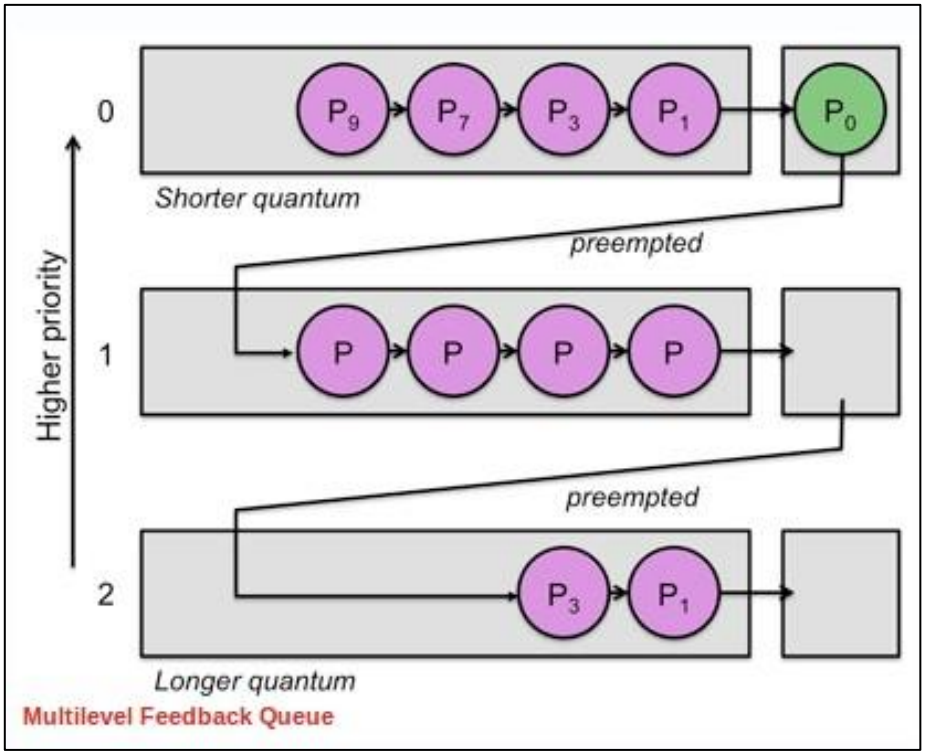
- Process dynamically moves between priority classes based on its CPU/ IO activity
- Basic observation
  - CPU bound process' likely to complete its entire timeslice
  - IO bound process' may not complete the entire time slice



Process 1 and 4 likely CPU bound Process 2 likely IO bound

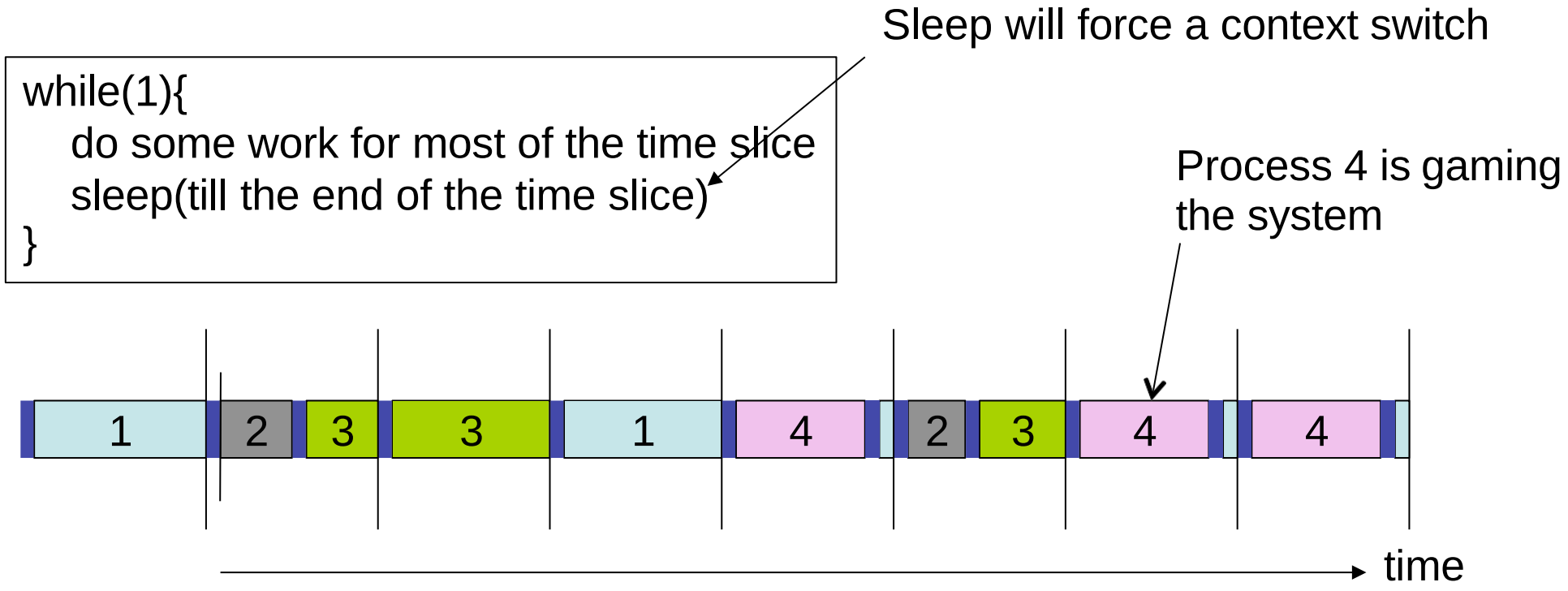
Multilevel feedback Queues (basic Idea)

- All processes start in the highest priority class
- If it finishes its time slice (likely CPU bound)
  - Move to the next lower priority class
- If it does not finish its time slice (likely IO bound)
  - Keep it on the same priority class
- As with any other priority based scheduling scheme, starvation needs to be dealt with

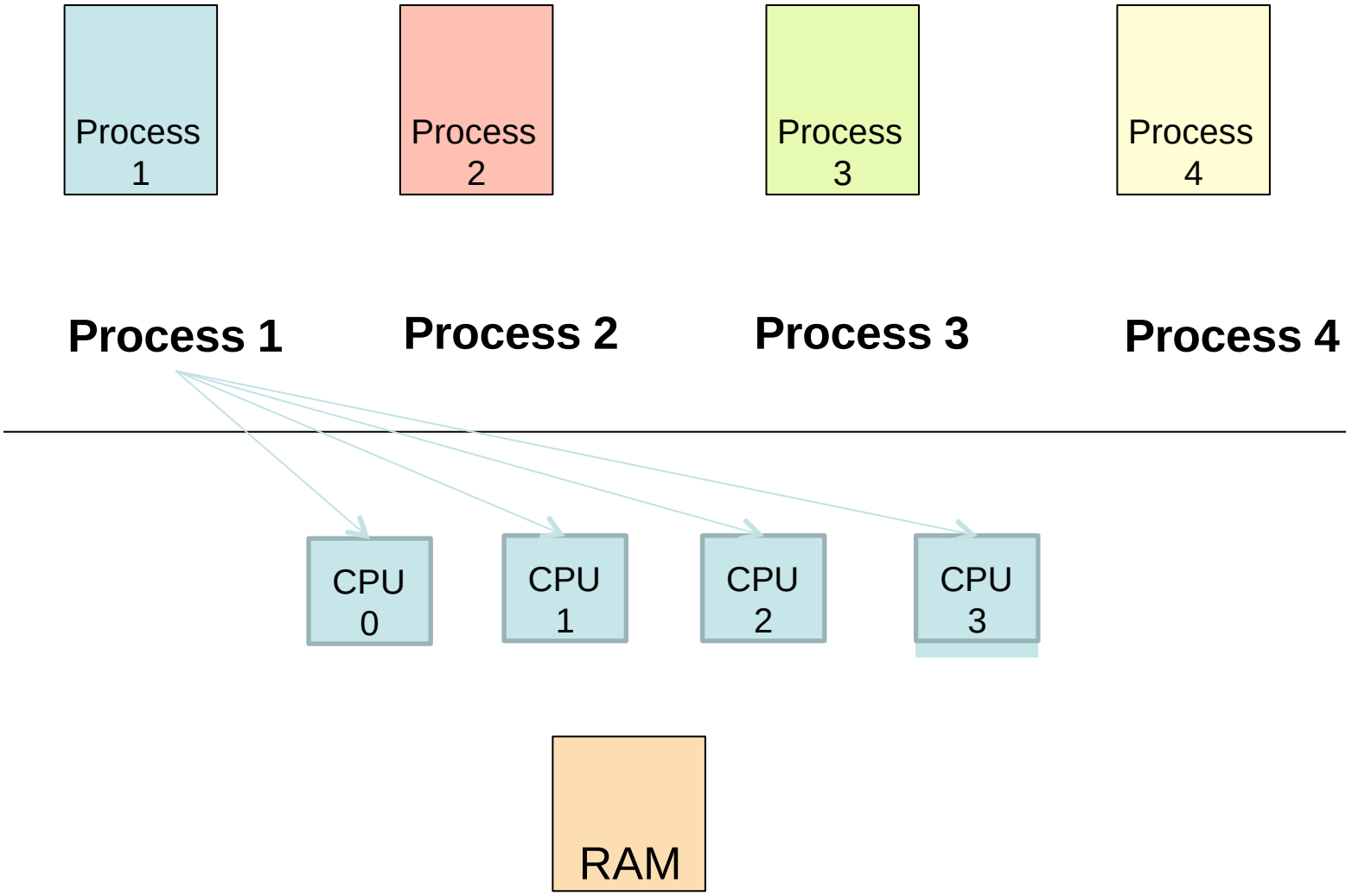


Gaming the System

- A compute intensive process can trick the scheduler and remain in the high priority queue (class)



Multiprocessor Scheduling

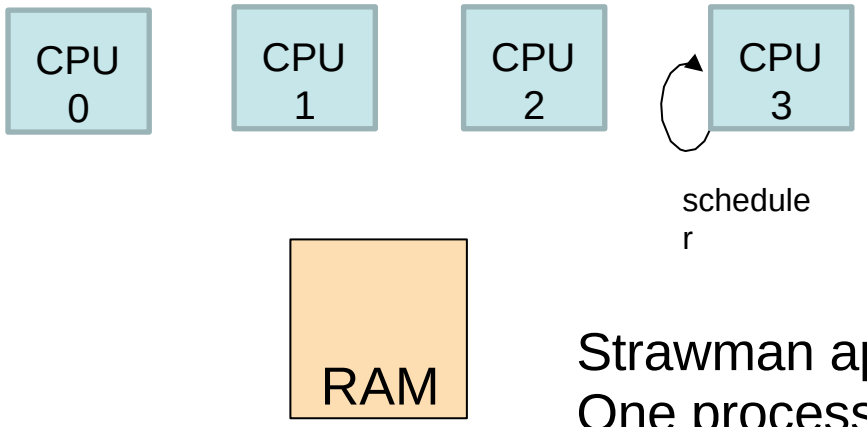


Process Migration

- As a result of symmetrical multiprocessing
  - A process may execute in a processor in one timeslice and another processor in the next time slice
  - This leads to process migration
    - Migration is expensive, it requires all memories to be repopulated
- Processor affinity
  - Process has a bitmask that tells what processors it can run on
    - Two types of processor affinity
      - Hard affinity – strict affinity to specific processors
      - Soft affinity

Multiprocessor Scheduling with a single scheduler

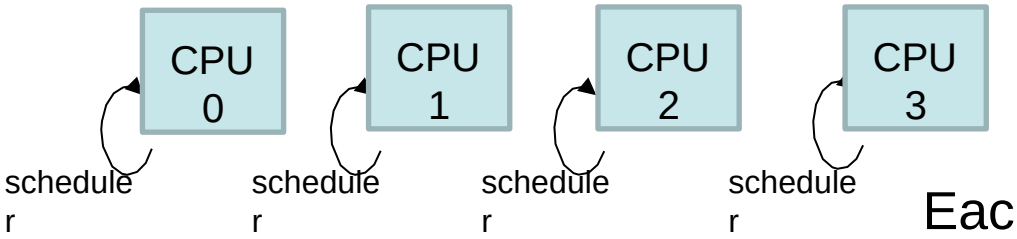
Process 1 **Process 2** **Process 3** **Process 4**

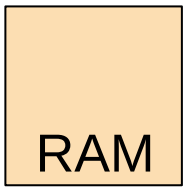


Strawman approach!!
One processor decides for everyone

Multiprocessor Scheduling (Symmetical Scheduling)

Process 1 **Process 2** **Process 3** **Process 4**





Two variants,

- Global queues
- Per CPU queues

Each processor runs a scheduler independently to select the process to execute

Requires locking to access the queues

Symmetrical Scheduling (with global queues)

Advantages

Good CPU Utilization

Fair to all processes

Disadvantages

Not scalable

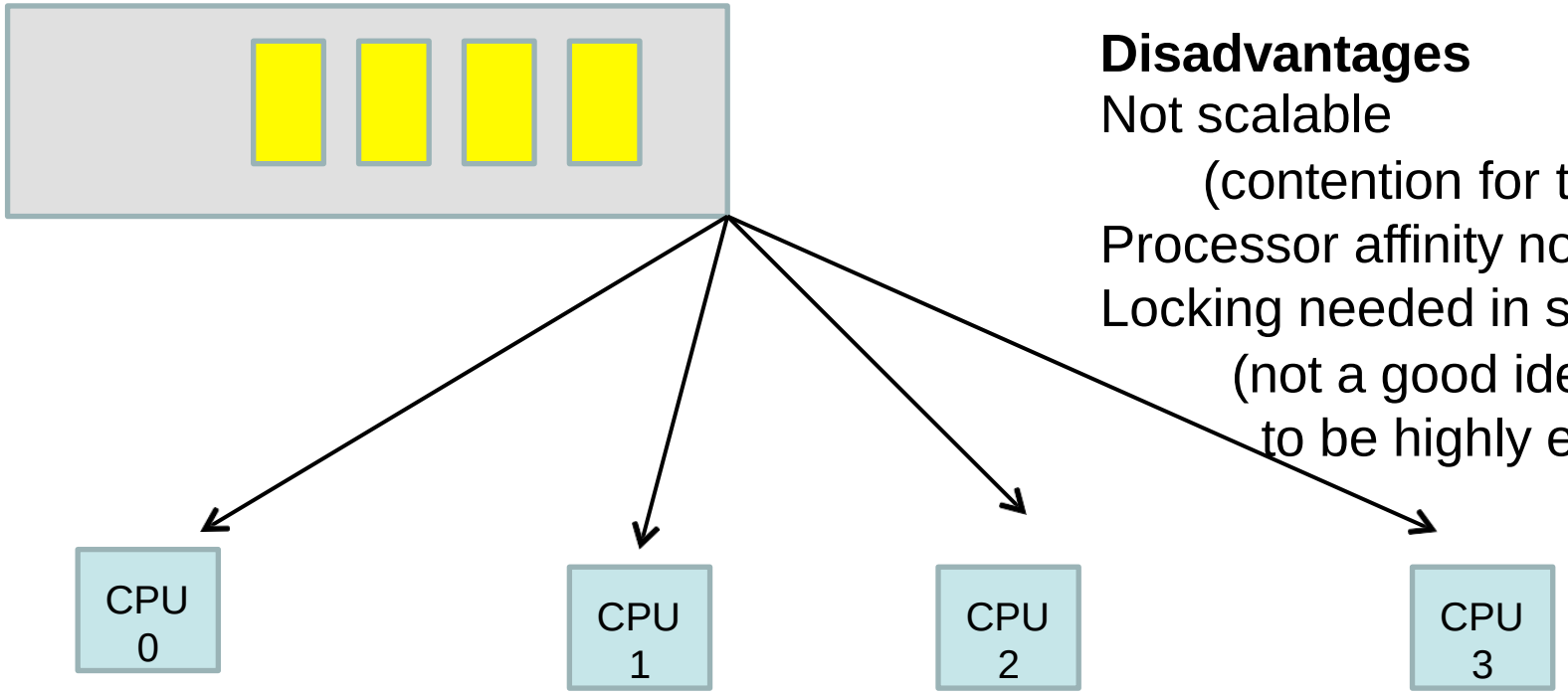
(contention for the global queue)

Processor affinity not easily achieved

Locking needed in scheduler

(not a good idea. Schedulers need to be highly efficient)

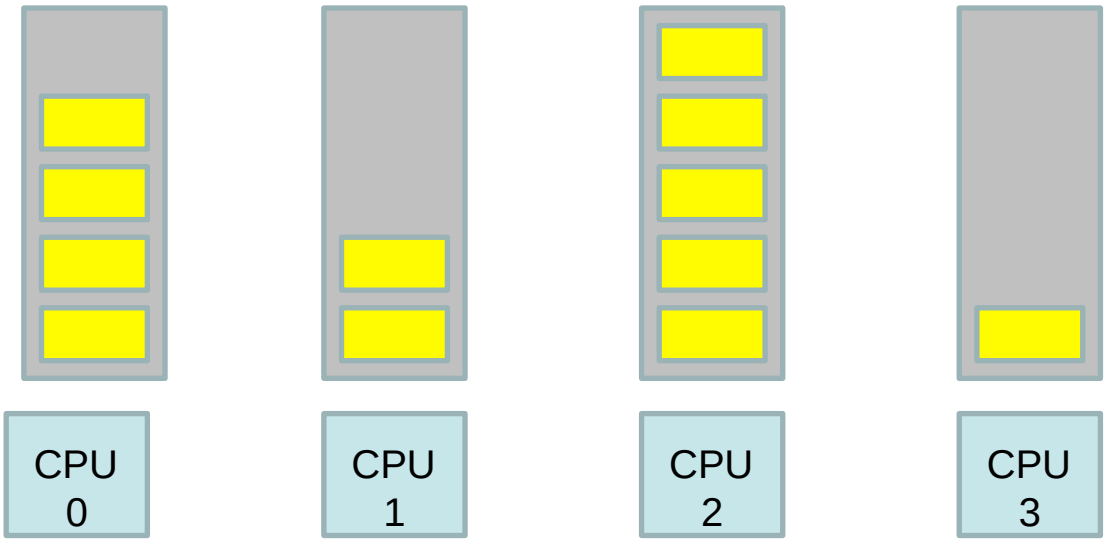
Global queues of runnable processes



Used in Linux 2.4, xv6

Symmetrical Scheduling (with per CPU queues)

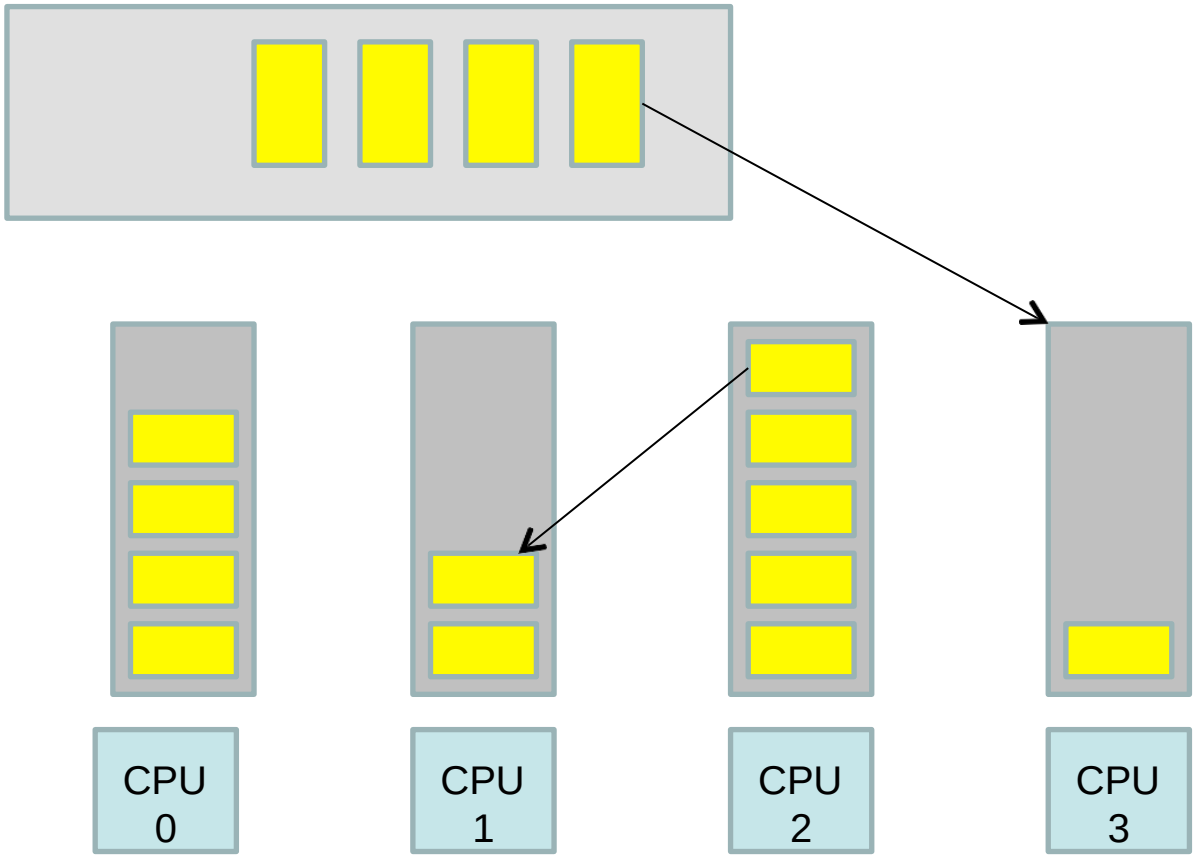
- Static partition of processes across CPUs



Advantages
Easy to implement
Scalable (no contention)
Locality

Disadvantages
Load imbalance

Hybrid Approach



- Use local and global queues
- Load balancing across queues feasible
- Locality achieved by processor affinity wrt the local queues
- Similar approach followed in Linux 2.6

Load Balancing

- On SMP systems, one processor may be overworked, while another underworked
- Load balancing attempts to keep the workload evenly distributed across all processors
- Two techniques
  - Push Migration : A special task periodically monitors load of all processors, and redistributes work when it finds an imbalance
  - Pull Migration : Idle processors pull a waiting task from a busy processor

Scheduling in Linux

Chester Rebeiro
IIT Madras

Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel,* 3rd Edition

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low priority task
- Normal Processes
  - Interactive
    - Constantly interact with their users, therefore spend a lot of time waiting for key presses and mouse operations.
    - When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)
  - Batch
    - Do not require any user interaction, often run in the background.

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low priority task
- Normal Processes
  - Interactive
    - Constantly interact with their users, therefore spend a lot of time waiting for key presses and mouse operations.
    - When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)
  - Batch
    - Do not require any user interaction, often run in the background.

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low priority task
- Normal Processes
  - Interactive
    - Constantly interact with their users, therefore spend a lot of time waiting for key presses and mouse operations.
    - When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)
  - Batch
    - Do not require any user interaction, often run in the background.

Once a process is specified real time, it is always considered a real time process

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low
- Normal Processes
  - Interactive
    - Constantly interact with their user
      for key presses and mouse oper
    - When input is received, the proce
      be between 50 to 150 ms)
  - Batch
    - Do not require any user interaction, often run in the background.

A process may act as an interactive process for some time and then become a batch process.

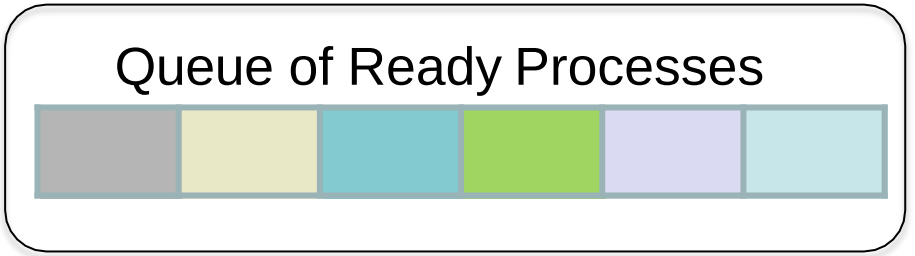
Linux uses sophisticated heuristics based on past behavior of the process to decide whether a given process should be considered interactive or batch

History (Schedulers for Normal Processors)

- O(n) scheduler
  - Linux 2.4 to 2.6
- O(1) scheduler
  - Linux 2.6 to 2.6.22
- CFS scheduler
  - Linux 2.6.23 onwards

O(n) Scheduler

- At every context switch
  - Scan the list of runnable processes
  - Compute priorities
  - Select the best process to run



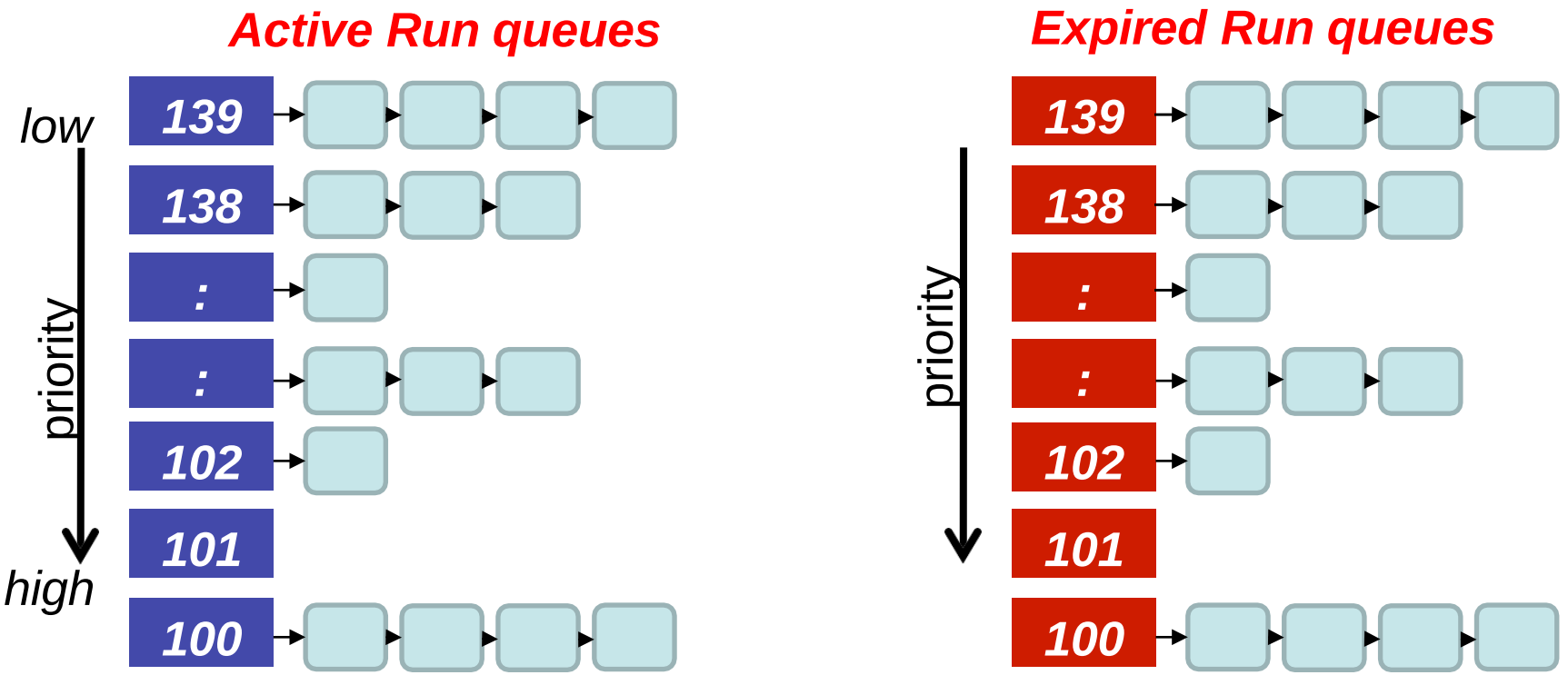
- O(n), when n is the number of runnable processes … not scalable!!
  - Scalability issues observed when Java was introduced (JVM spawns many tasks)
- Used a global run-queue in SMP systems
  - Again, not scalable!!

O(1) scheduler

- Constant time required to pick the next process to execute
  - easily scales to large number of processes
- Processes divided into 2 types
  - Real time
    - Priorities from 0 to 99
  - Normal processes
    - Interactive
    - Batch
    - Priorities from 100 to 139 (100 highest, 139 lowest priority)

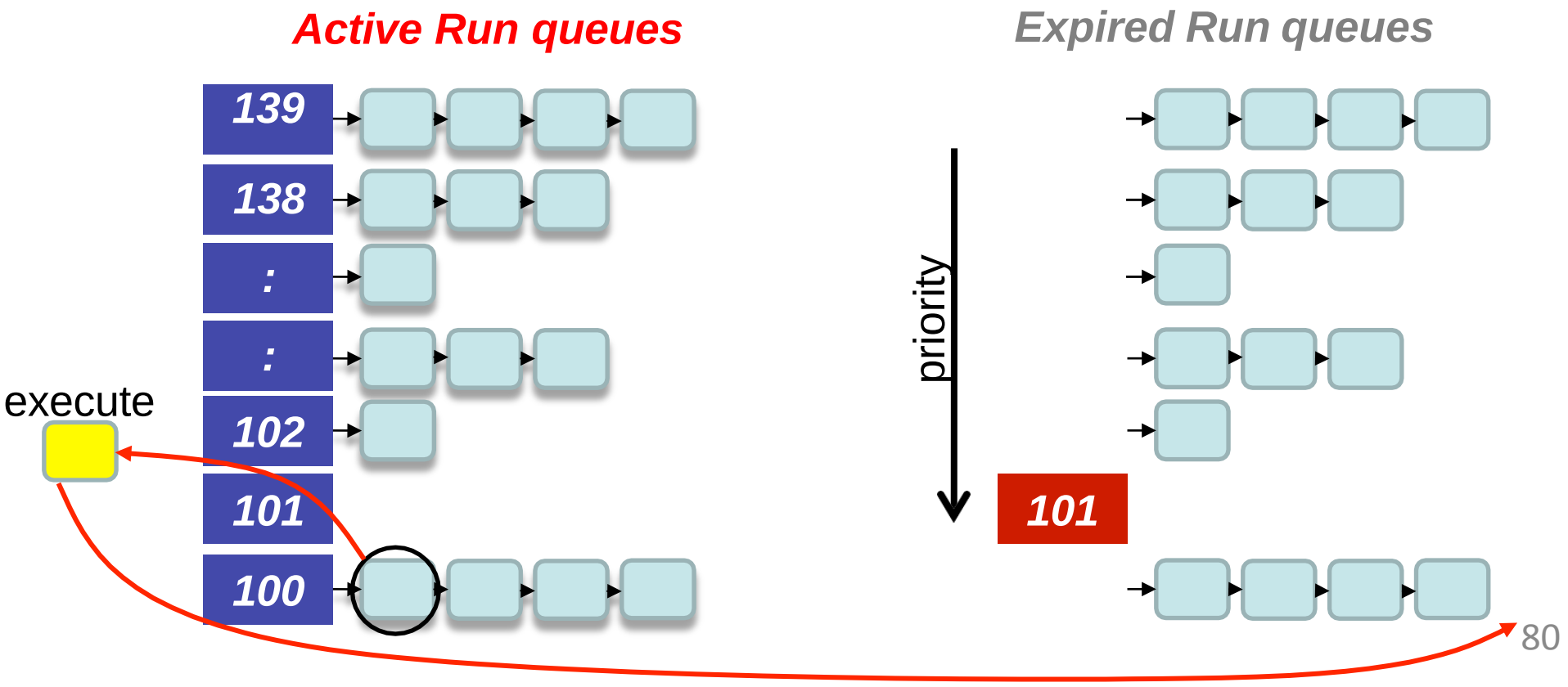
Scheduling Normal Processes

- Two ready queues in each CPU
  - Each queue has 40 priority classes (100 – 139)
  - 100 has highest priority, 139 has lowest priority



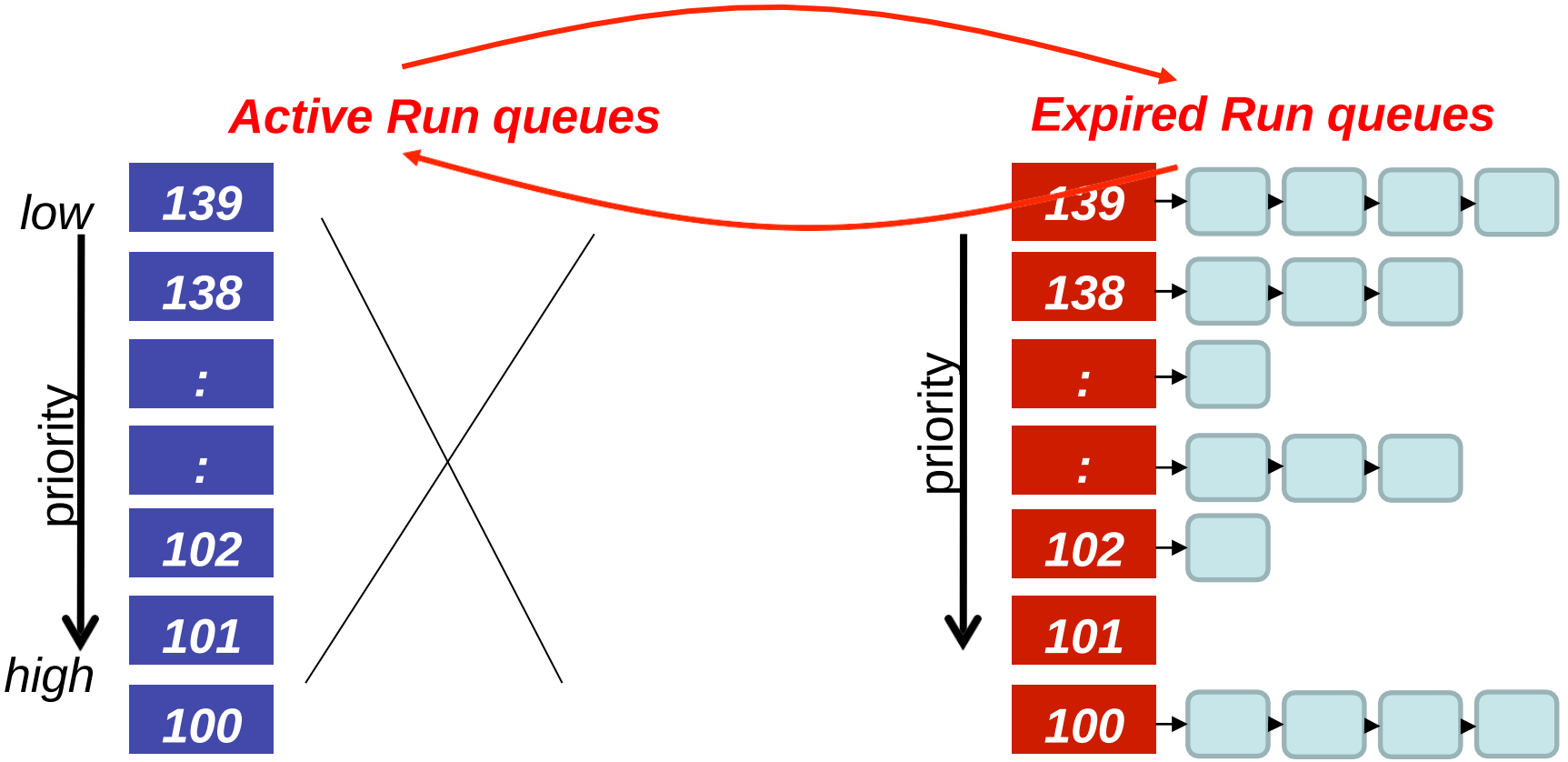
The Scheduling Policy

- Pick the first task from the lowest numbered run queue
- When done put task in the appropriate queue in the expired run queue



The Scheduling Policy

- Once active run queues are complete
  - Make expired run queues active and vice versa



contant time?

- There are 2 steps in the scheduling
  1. Find the lowest numbered queue with at least 1 task
  2. Choose the first task from that queue

- step 2 is obviously constant time
- Is step 1 contant time?
  - Store bitmap of run queues with non-zero entries
  - Use special instruction '*find-first-bit-set*'
    - *bsfl* on intel

More on Priorities

- 0 to 99 meant for real time processes
- 100 is the highest priority for a normal process
- 139 is the lowest priority
- Static Priorities
  - 120 is the base priority (default)
  - **nice :** command line to change default priority of a process

    $nice –n N ./a.out
  - N is a value from +19 to -20;
    - most selfish ‘-20’; (I want to go first)
    - most generous ‘+19’; (I will go last)

Based on a heuristic

Dynamic Priority

- To distinguish between batch and interactive processes
- Uses a ‘bonus’, which changes based on a heuristic

dynamic priority = MAX(100, MIN(static priority – bonus + 5), 139))

Has a value between 0 and 10

If bonus < 5, implies less interaction with the user
thus more of a CPU bound process.
The dynamic priority is therefore decreased (toward 139)

If bonus > 5, implies more interaction with the user
thus more of an interactive process.
The dynamic priority is increased (toward 100).

Dynamic Priority (setting the bonus)

- To distinguish between batch and interactive processes
- Based on average sleep time
  - An I/O bound process will sleep more therefore should get a higher priority
  - A CPU bound process will sleep less, therefore should get lower priority

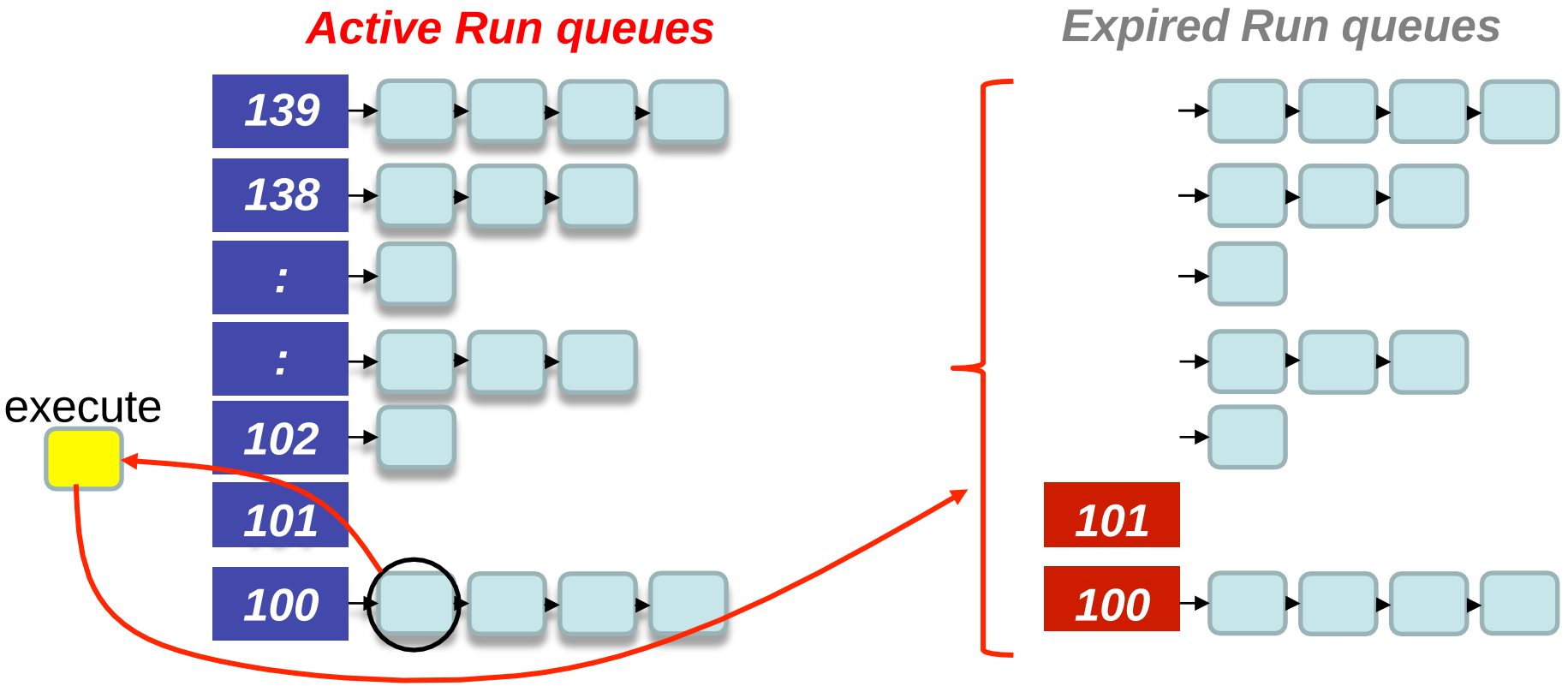
dynamic priority = MAX(100, MIN(static priority – bonus + 5), 139))

heuristic

Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9
1 second	10

Dynamic Priority and Run Queues

- Dynamic priority used to determine which run queue to put the task
- No matter how ‘nice’ you are, you still need to wait on run queues --- prevents starvation



Setting the Timeslice

- Interactive processes have high priorities.
  - But likely to not complete their timeslice
  - Give it the largest timeslice to ensure that it completes its burst without being preempted. More heuristics

```
If priority < 120
        time slice = (140 – priority) * 20    milliseconds
else
        time slice = (140 – priority) * 5    milliseconds
```

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	19	5 ms

Summarizing the O(1) Scheduler

- Multi level feed back queues with 40 priority classes
- Base priority set to 120 by default; modifiable by users using nice.
- Dynamic priority set by heuristics based on process' sleep time
- Time slice interval for each process is set based on the dynamic priority

Limitations of O(1) Scheduler

- Too complex heuristics to distinguish between interactive and non-interactive processes
- Dependence between timeslice and priority
- Priority and timeslice values not uniform

Completely Fair Scheduling (CFS)

- The Linux scheduler since 2.6.23
- By Ingo Molnar
  - based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas.
  - Incorporated in the Linux kernel since 2007
- No heuristics.
- Elegant handling of I/O and CPU bound processes.

Completely Fair Scheduling (CFS)

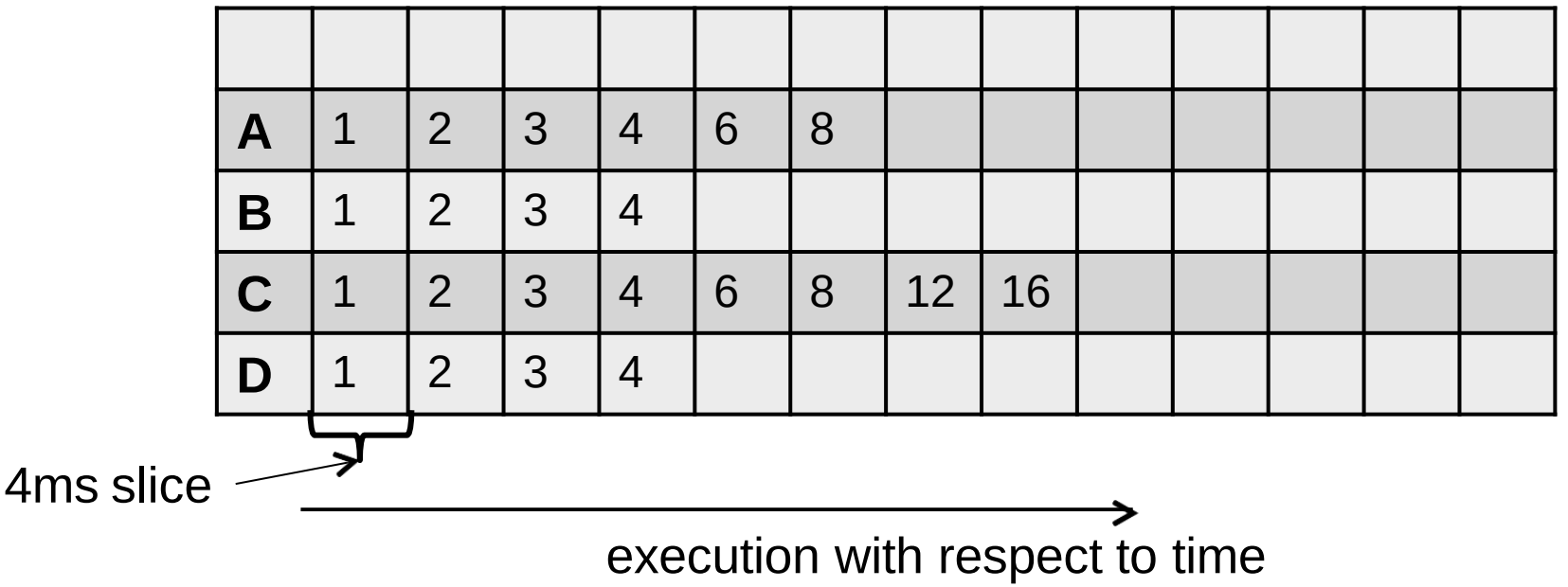
Ideal Fair Scheduling

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness



Ideal Fair Scheduling

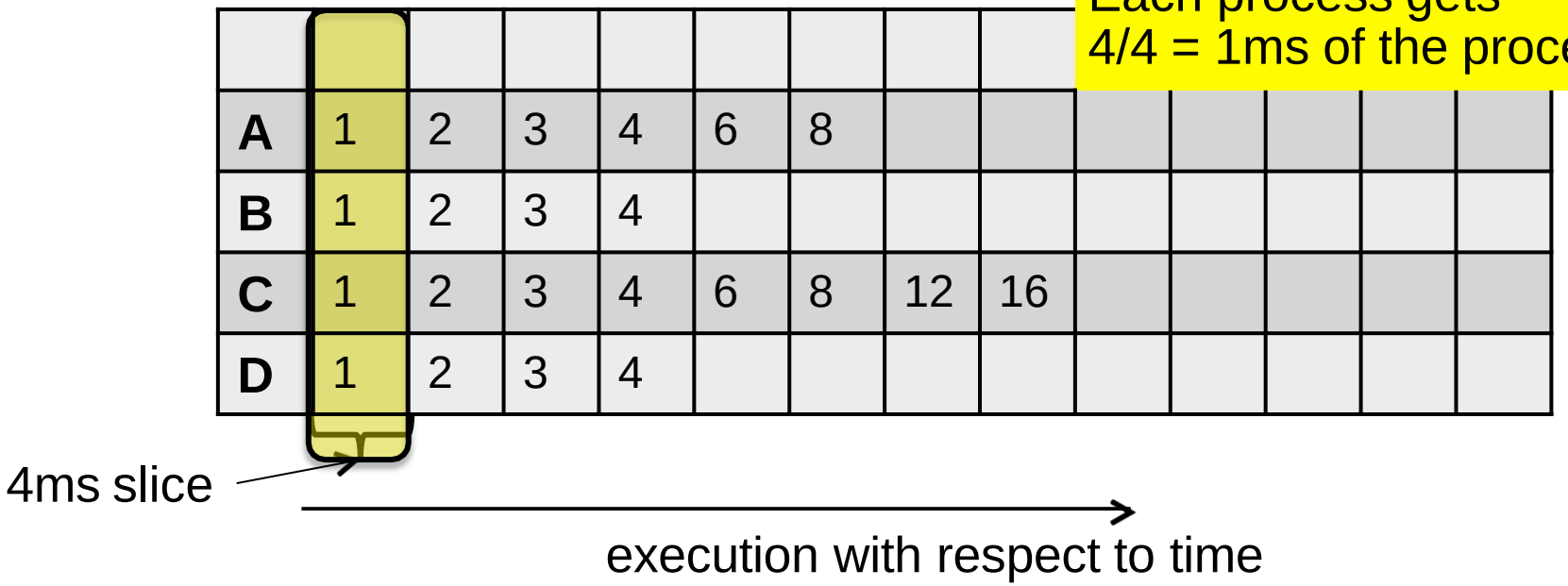
Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

Each process gets
4/4 = 1ms of the processor time



Ideal Fair Scheduling

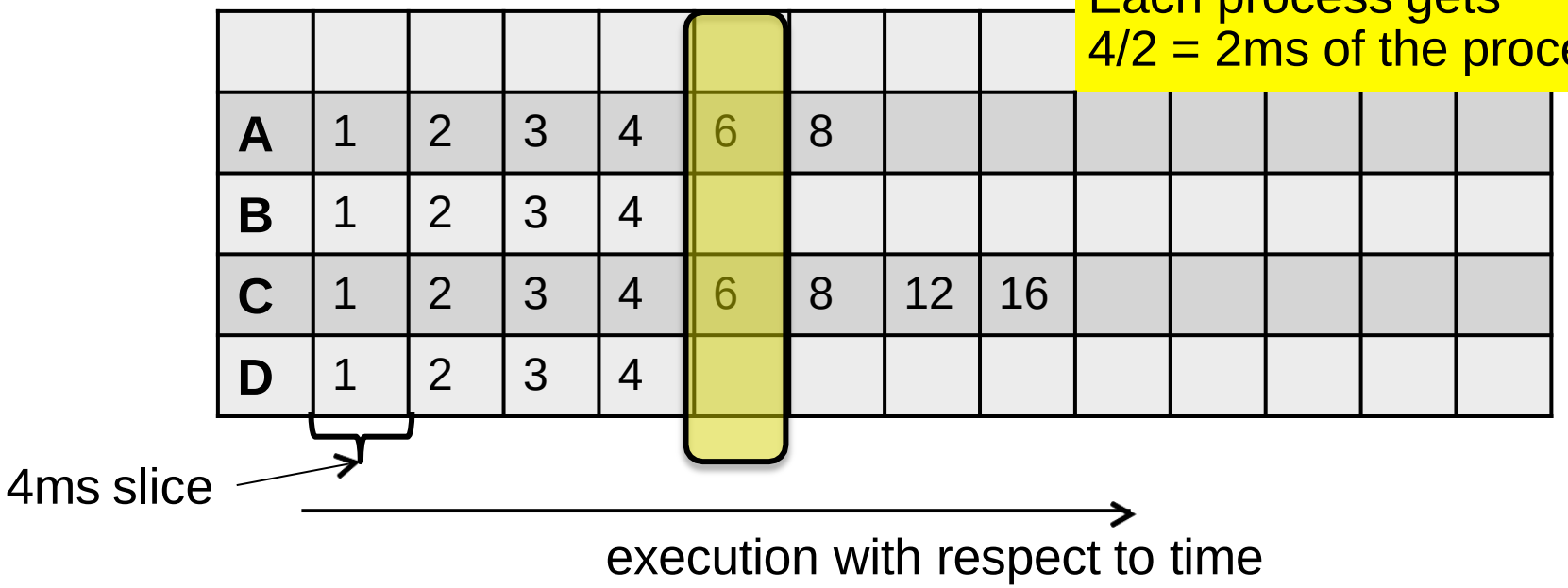
Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

Each process gets
4/2 = 2ms of the processor time



Ideal Fair Scheduling

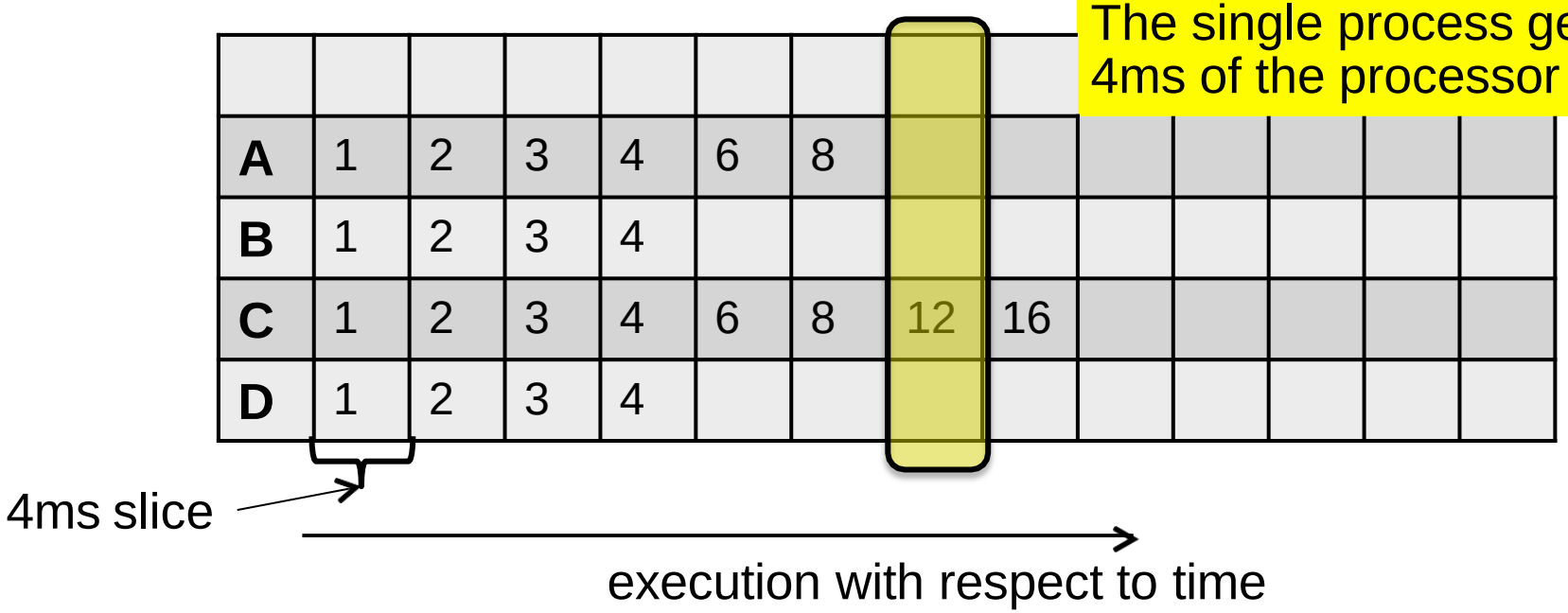
Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

The single process gets the entire 4ms of the processor time



Virtual Runtimes

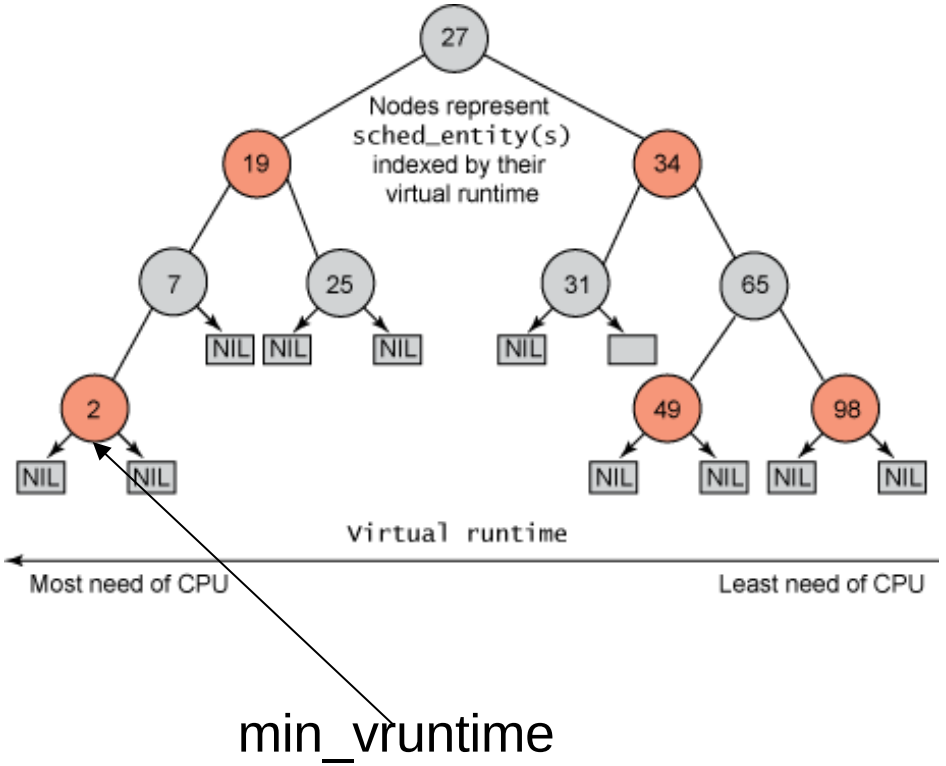
- With each runnable process is included a virtual runtime (vruntime)
  - At every scheduling point, if process has run for t ms, then (vruntime += t)
  - vruntime for a process therefore monotonically increases

The CFS Idea

- When timer interrupt occurs
  - Choose the task with the lowest vruntime (min_vruntime)
  - Compute its dynamic timeslice
  - Program the high resolution timer with this timeslice
- The process begins to execute in the CPU
- When interrupt occurs again
  - Context switch if there is another task with a smaller runtime

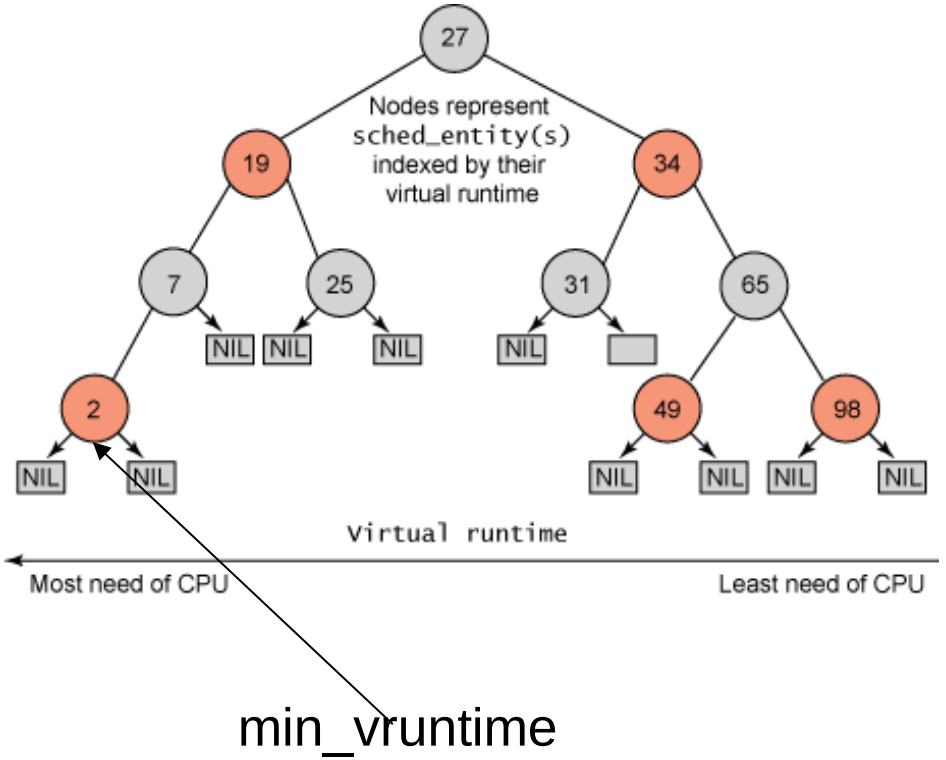
Picking the Next Task to Run

- CFS uses a red-black tree.
  - Each node in the tree represents a runnable task
  - Nodes ordered according to their vruntime
  - Nodes on the left have lower vruntime compared to nodes on the right of the tree
  - The left most node is the task with the least vruntime
    - This is cached in min_vruntime



Picking the Next Task to Run

- At a context switch,
  - Pick the left most node of the tree
    - This has the lowest runtime.
    - It is cached in min_vruntime. Therefore accessed in O(1)
  - If the previous process is runnable, it is inserted into the tree depending on its new vruntime. Done in O(log(n))
    - Tasks move from left to right of tree after its execution completes… starvation avoided



Why Red Black Tree?

- Self Balancing
  - No path in the tree will be twice as long as any other path

- All operations are O(log n)
  - Thus inserting / deleting tasks from the tree is quick and efficient

Priorities and CFS

- Priority (due to nice values) used to weigh the vruntime

- if process has run for t ms, then

  vruntime += t * (weight based on nice of process)

- A lower priority implies time moves at a faster rate compared to that of a high priority task

I/O and CPU bound processes

- What we need,
  - I/O bound should get higher priority and get a longer time to execute compared to CPU bound
  - CFS achieves this efficiently
    - I/O bound processes have small CPU bursts therefore will have a low vruntime. They would appear towards the left of the tree…. Thus are given higher priorities
    - I/O bound processes will typically have larger time slices, because they have smaller vruntime

New Process

- Gets added to the RB-tree
- Starts with an initial value of min_vruntime..
- This ensures that it gets to execute quickly