

A decorative border composed of a grid of colored dots in various colors including purple, blue, green, yellow, red, and brown, framing the text on the slide.

Système d'Exploitation

Sous-système de Gestion de Processus

Université François Rabelais de Tours
Faculté des Sciences et Techniques
Antenne Universitaire de Blois

Licence Sciences et Technologies

Mention : Informatique

2^{ème} Année

Mohamed TAGHELIT
taghelit@univ-tours.fr

A decorative graphic consisting of a grid of small, colored dots. The dots are arranged in a crosshair pattern, with a horizontal row and a vertical column intersecting at the center. The dots are in various colors including blue, green, yellow, orange, red, purple, and pink.

Plan

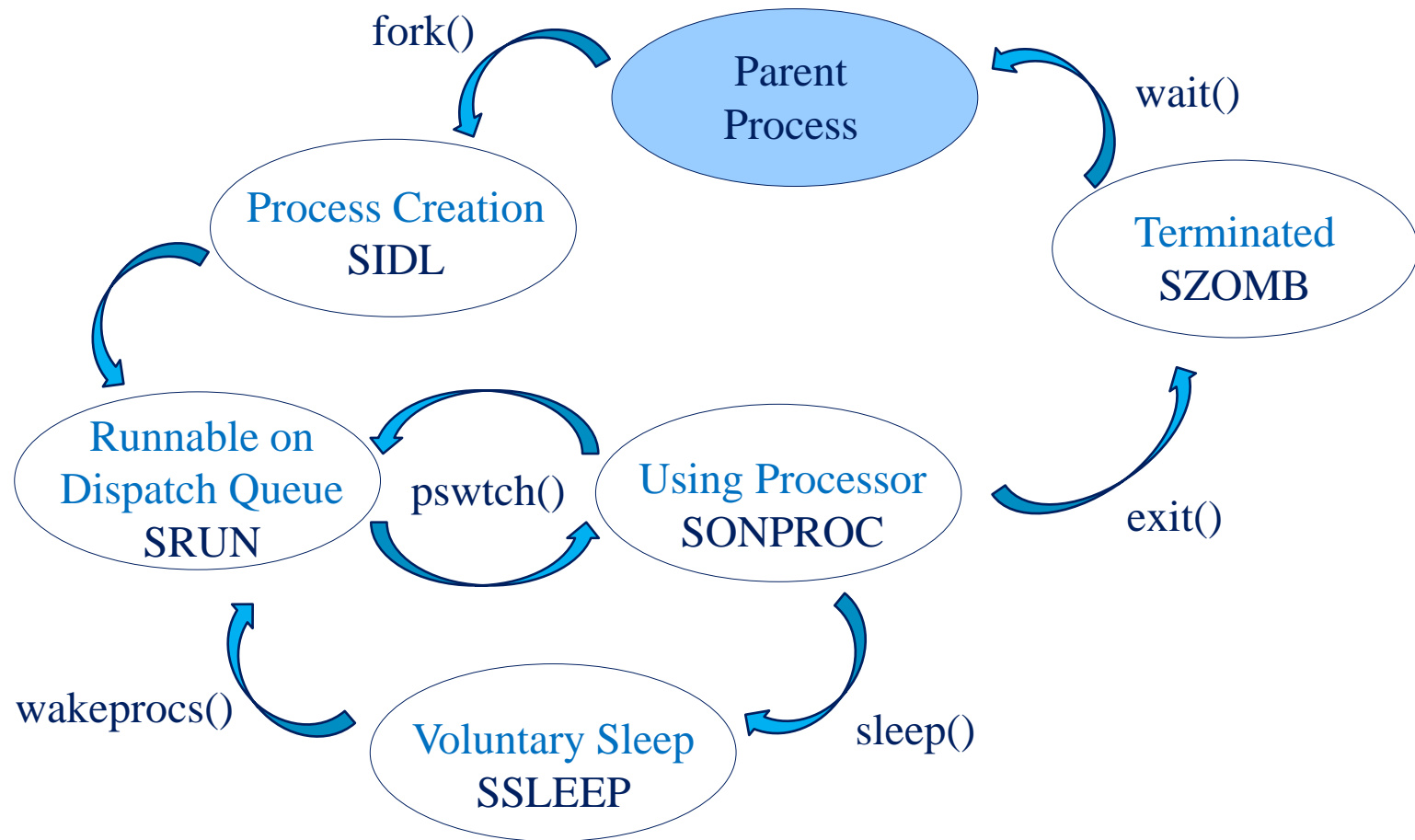
1. Sous-système Gestion de Processus
2. États d'un Processus
3. Ordonnancement d'Exécution
4. Ordonnancement Unix

Sous-système Gestion de Processus

Constitue le cœur d'un SE → coordonne tous les besoins nécessaires à la gestion des processus.

- Cycle de vie d'un processus → création, exécution et terminaison
- Ordonnancement (*Scheduling*) → les processus sont placés dans des files et leur exécution est ordonnée en fonction de leurs classe et niveau de priorité
- Commutation (*Switching*) → détermine combien de temps allouer l'UC à un processus et quand le lui retirer pour l'allouer à un autre
- Temps (*Timing*) → suivre le temps d'exécution d'un processus en surveillant le temps consommé (et donc l'horloge)
- Mémoire utilisée → coordination avec le sous-système de gestion mémoire pour les besoins d'allocation/libération mémoire (création processus, besoins propres des processus, ...)
- Coordination avec le sous-système de fichier → association processus/fichiers, localisation et transfert en mémoire pour exécution
- Gestion des exceptions → émission et réception des signaux (erreurs durant l'exécution)

États d'un processus



Ordonnancement d'Exécution

- Base des SE multiprogrammés → optimiser l'utilisation de l'Unité Centrale
- Circonstances exigeant un ordonnancement

- État d'exécution → État d'attente

Ordonnancement

préemptif

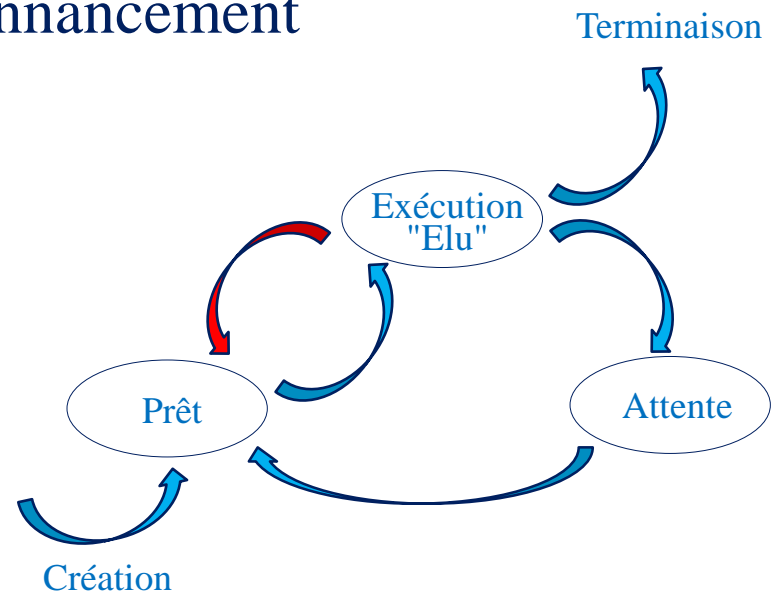


- État d'exécution → État prêt

- État d'attente → État prêt

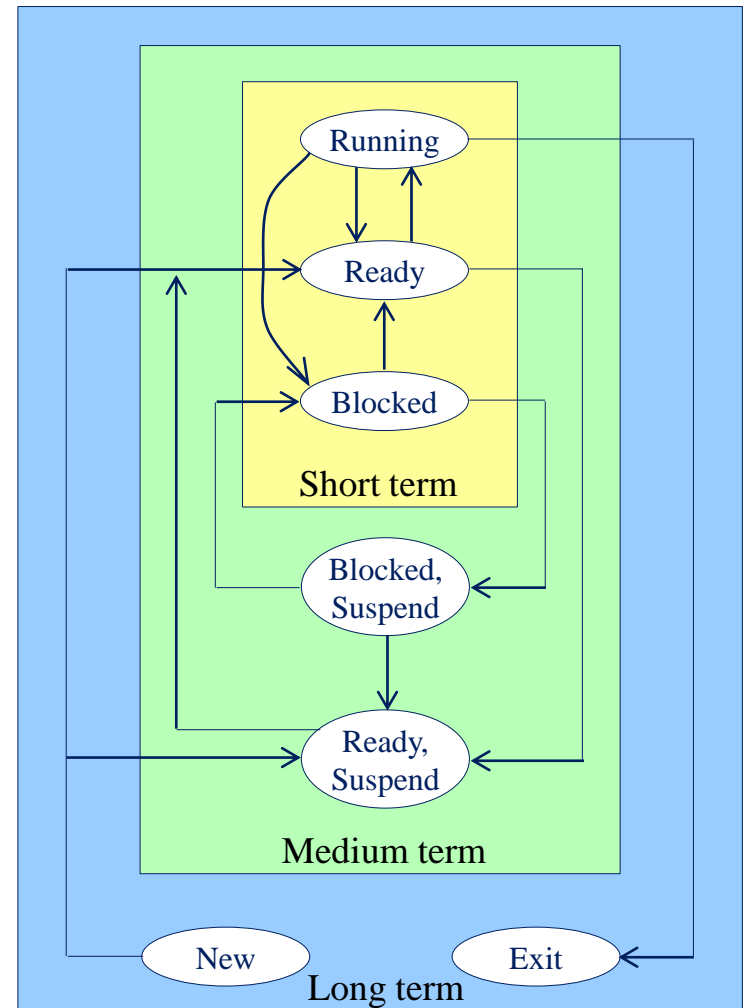
- État d'exécution → Terminaison

- Création → État prêt



Types d'Ordonnancement

- Ordonnancement à long terme
 - Décision d'ajout de processus pour exécution
 - Contrôle le degré de multiprogrammation
- Ordonnancement à moyen terme
 - Décision d'ajout de processus en mémoire
- Ordonnancement à court terme
 - Décision du choix du processus devant s'exécuter



Ordonnancement Préemptif

- Traduit le fait que le processeur peut être réquisitionné
- Ordonnancement préemptif
 - Interruption de l'exécution du processus en cours
 - Possibilité de réquisition du processeur
- Ordonnancement non préemptif
 - Changement de contexte à la fin de l'exécution du processus en cours
 - Changement de contexte volontaire du processus en cours

Critères d'Ordonnancement

- Utilisation de l'UC → occuper l'UC au mieux [40% ↔ 90%]
- Débit ou Rendement (*throughput*) → nombre de processus terminés par unité de temps
- Temps de rotation ou de Service (*turnaround time*) → intervalle de temps entre la soumission d'un processus et son achèvement (¬ système interactif)
- Temps d'attente (*waiting time*) → somme des périodes passées en attente dans la file des processus prêts
- Temps de réponse (*response time*) → intervalle de temps entre la soumission d'une requête et la production de la première réponse

Ordonnancement FCFS

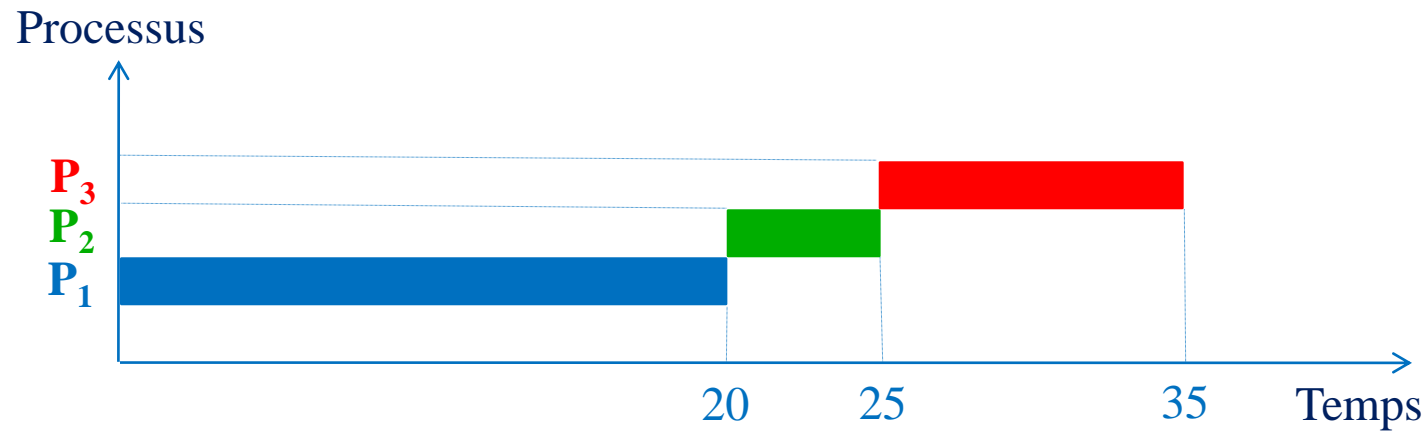
First-Come, First-Served

- Ordonnancement le plus simple
- Principe : premier arrivé, premier servi. Le premier processus qui demande l'unité centrale la reçoit en premier
- Pas de réquisition
- Implémentation → une seule file d'attente (processus prêts) FIFO
- Temps d'attente assez long
- Les processus courts peuvent être pénalisés

Exemple d'Ordonnancement FCFS

| Processus | Durée | Instant Soumission |
|-----------|-------|--------------------|
| P_1 | 20 | 0 |
| P_2 | 5 | 0 |
| P_3 | 10 | 0 |

①
↓
②
↓
③
Ordre d'exécution



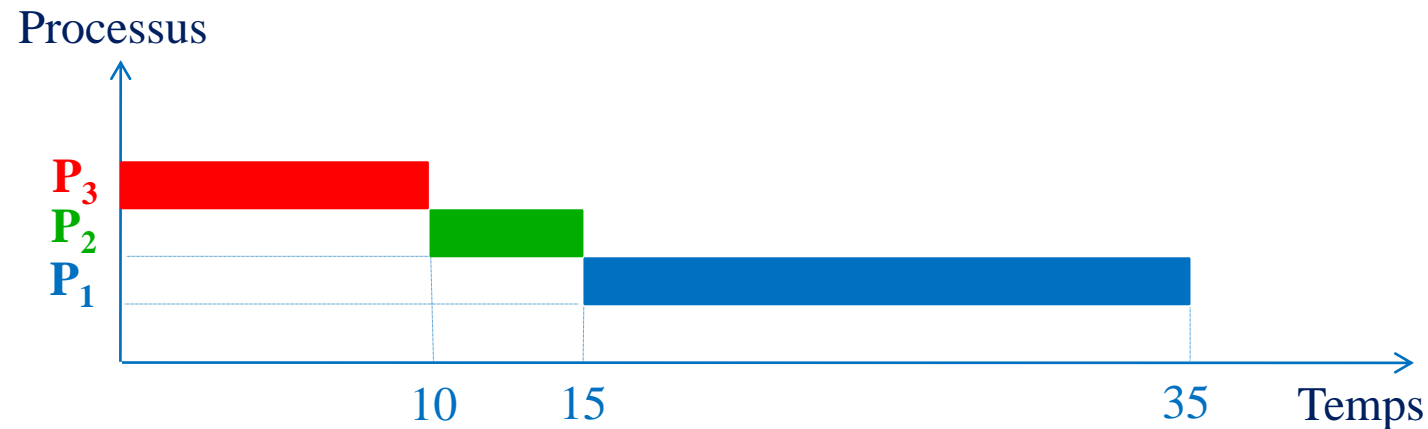
- Temps d'attente moyen = 15 ms

Exemple d'Ordonnancement FCFS

| Processus | Durée | Instant Soumission |
|----------------|-------|--------------------|
| P ₁ | 20 | 0 |
| P ₂ | 5 | 0 |
| P ₃ | 10 | 0 |

③
↑
②
↑
①

Ordre
d'exécution



- Temps d'attente moyen = 8,33 ms

Ordonnancement SJF

Shortest-Job-First

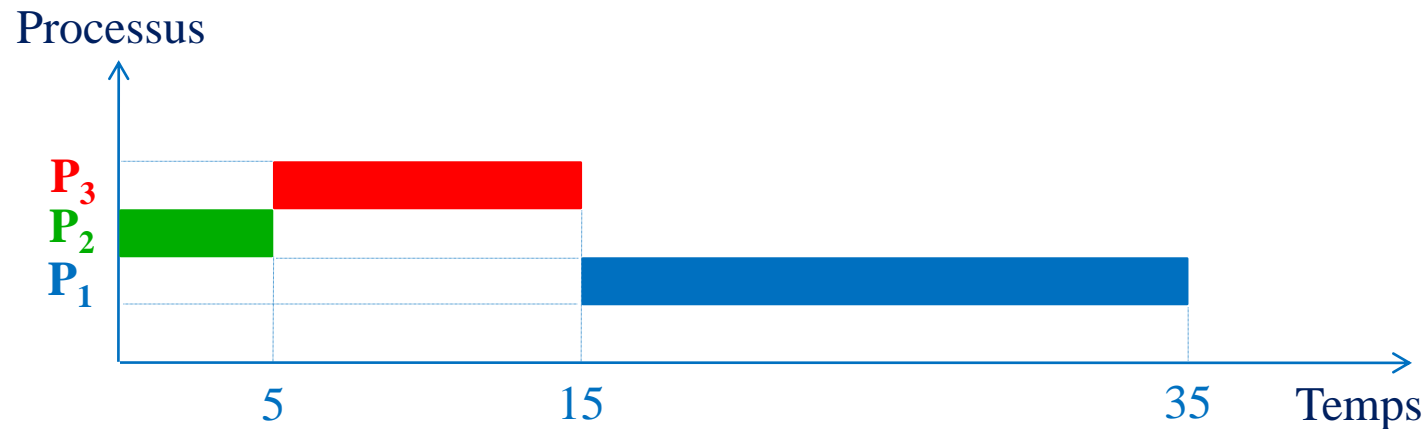
- Principe : le processus suivant le plus court d'abord
- Si deux processus ont une même durée, alors appel à l'ordonnancement FCFS
- Implémentation → une seule file d'attente (processus prêts) non FIFO
- Pas de réquisition
- Temps d'attente moyen optimal → en exécutant un processus court avant un processus long, on diminue le temps d'attente du processus court plus que n'augmente le temps d'attente du processus long \Rightarrow le temps moyen décroît
- Temps d'attente moyen $<$ Temps d'attente moyen FCFS
- Difficulté à connaître le temps d'exécution des processus

Exemple d'Ordonnancement SJF

| Processus | Durée | Instant Soumission |
|-----------|-------|--------------------|
| P_1 | 20 | 0 |
| P_2 | 5 | 0 |
| P_3 | 10 | 0 |

③
①
②

Ordre d'exécution



- Temps d'attente moyen = 6,66 ms

Commutation de Processus

- Une commutation est possible à chaque fois que le SE reprend le contrôle par rapport au processus en cours d'exécution.
- Événements pouvant donner le contrôle au SE :

| Mechanism | Cause | Use |
|------------------------|--|--|
| <i>Interrupt</i> | External to the execution of the current instruction | Reaction to an asynchronous external event |
| <i>Trap</i> | Associated with the execution of the current instruction | Handling of an error or an exception condition |
| <i>Supervisor call</i> | Explicit request | Call to an operating system function |

- Interruptions
 - Interruption Horloge
 - Interruptions E/S
 - Défaut de page

Interruption Horloge

- Horloge matérielle interrompant le système à des intervalles de temps fixes
- La période de temps entre deux interruptions est appelée *CPU tick*, *clock tick* ou *tick* (Linux → *jiffy*)
- Sous Unix, généralement, *tick* = 10 ms
- Fréquence de l'horloge (nombre de *ticks/s*) → HZ (*param.h*)
- Les fonctions du noyau mesurent toujours le temps en nombre de *ticks*
- Routine de traitement dépendant du matériel
 - Doit être courte !
 - Très prioritaire !
- Définition d'un quantum → 6 ou 10 *ticks*

Routine de Traitement de l'IT Horloge

1. Réarmer l'interruption horloge
2. Mise à jour des statistiques d'utilisation CPU du processus courant
3. Accomplir des fonctions relatives à l'ordonnancement
 - Recalculer la priorité des processus, expiration des quanta, ...
4. Envoyer SIGXCPU au processus courant si quota CPU dépassé
5. Mise à jour de l'heure du jour et de certains compteurs
6. Traiter les *callouts*
 - retransmission des paquets, fonctions du *scheduler* et du gestionnaire mémoire, scrutation des unités ne supportant pas les interruptions, ...
7. Réveiller processus système si nécessaire
 - *swapper, pagedaemon, ...*
8. Traiter les alarmes

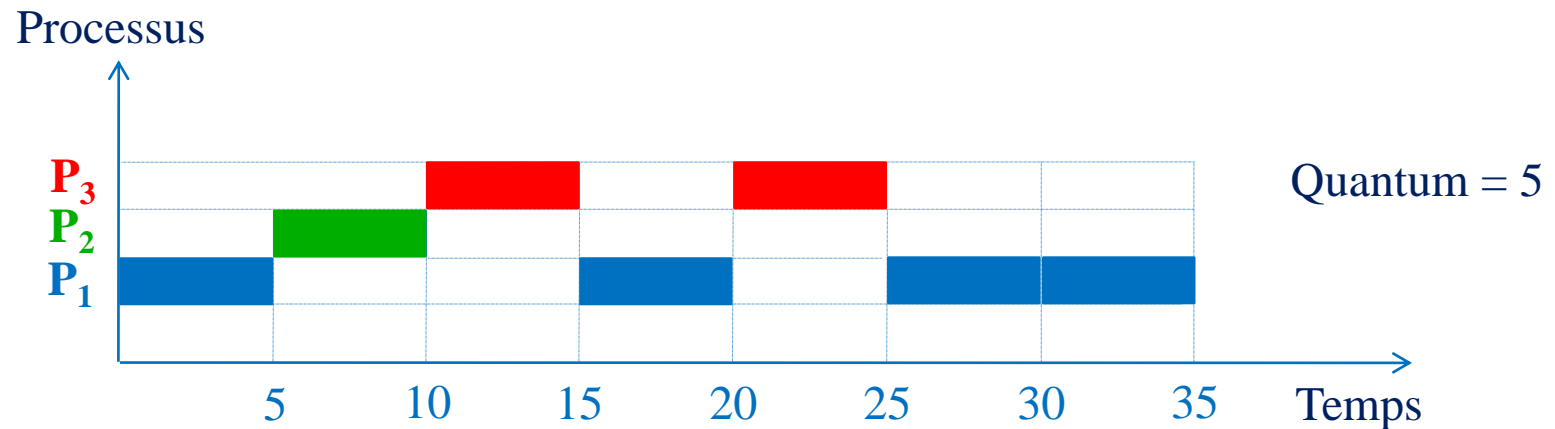
Ordonnancement par Tourniquet (RR)

Round-Robin

- Conçu spécialement pour les systèmes à temps partagé
- Principe : allouer le processeur à tour de rôle aux processus pour une durée prédéfinie = quantum
- Similaire FCFS + préemption
- Implémentation → une seule file d'attente (processus prêts) FIFO
 - Création d'un processus → insertion en queue de file
 - Élection → choisir le processus en tête de file
 - Fin quantum → réinsertion en queue de file
- Choix du quantum primordial
 - Trop petit → nombre important de commutations
 - Trop grand → temps de réponse important

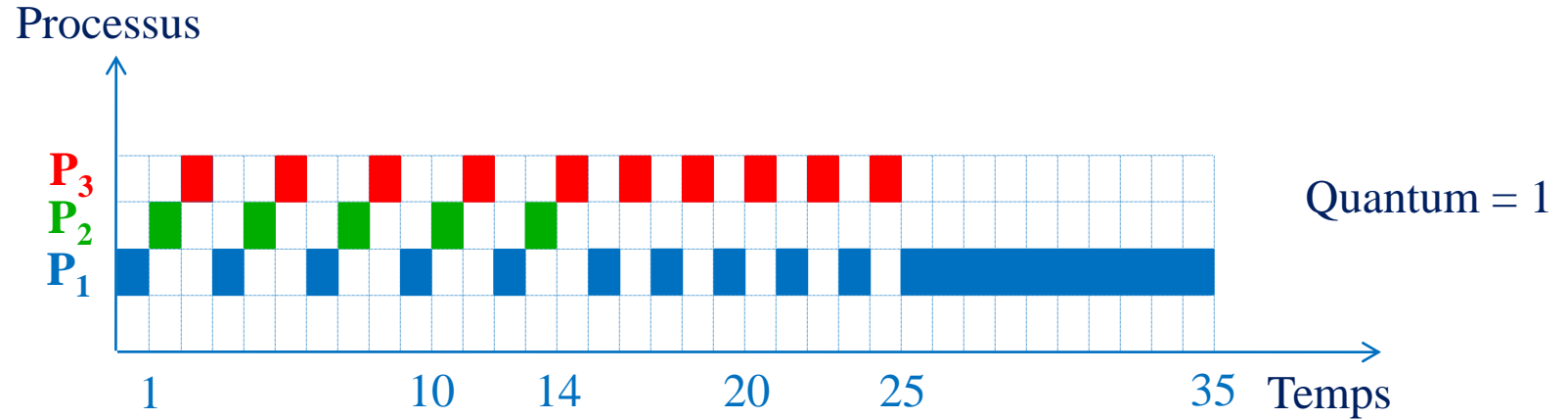
Exemple d'Ordonnancement RR

| Processus | Durée | Instant Soumission |
|-----------|-------|--------------------|
| P_1 | 20 | 0 |
| P_2 | 5 | 0 |
| P_3 | 10 | 0 |

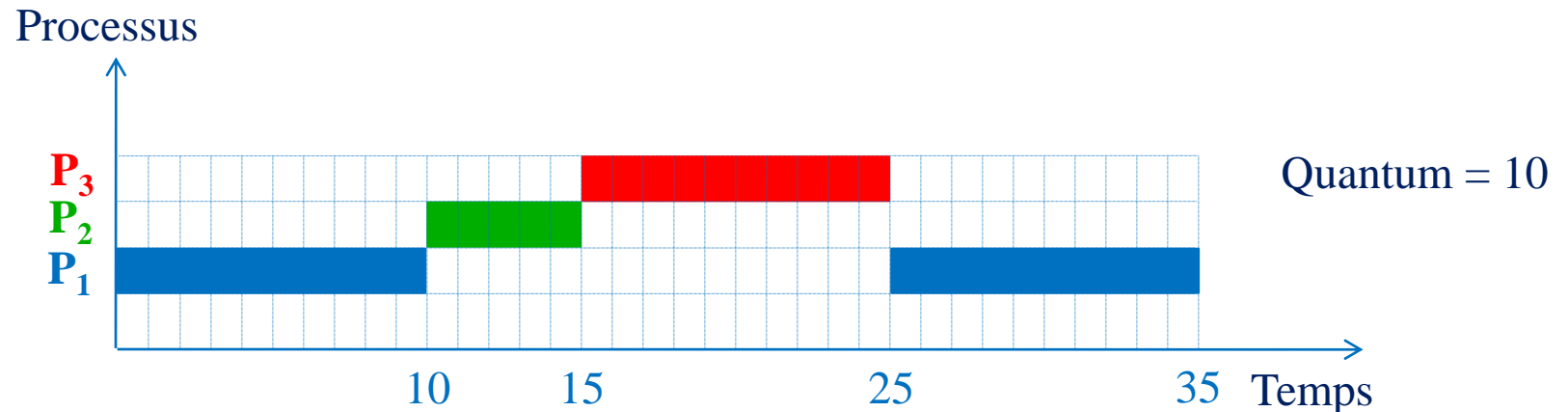


- Temps de réponse moyen = 5 ms

Exemple d'Ordonnancement RR



- Temps de réponse moyen = 1 ms

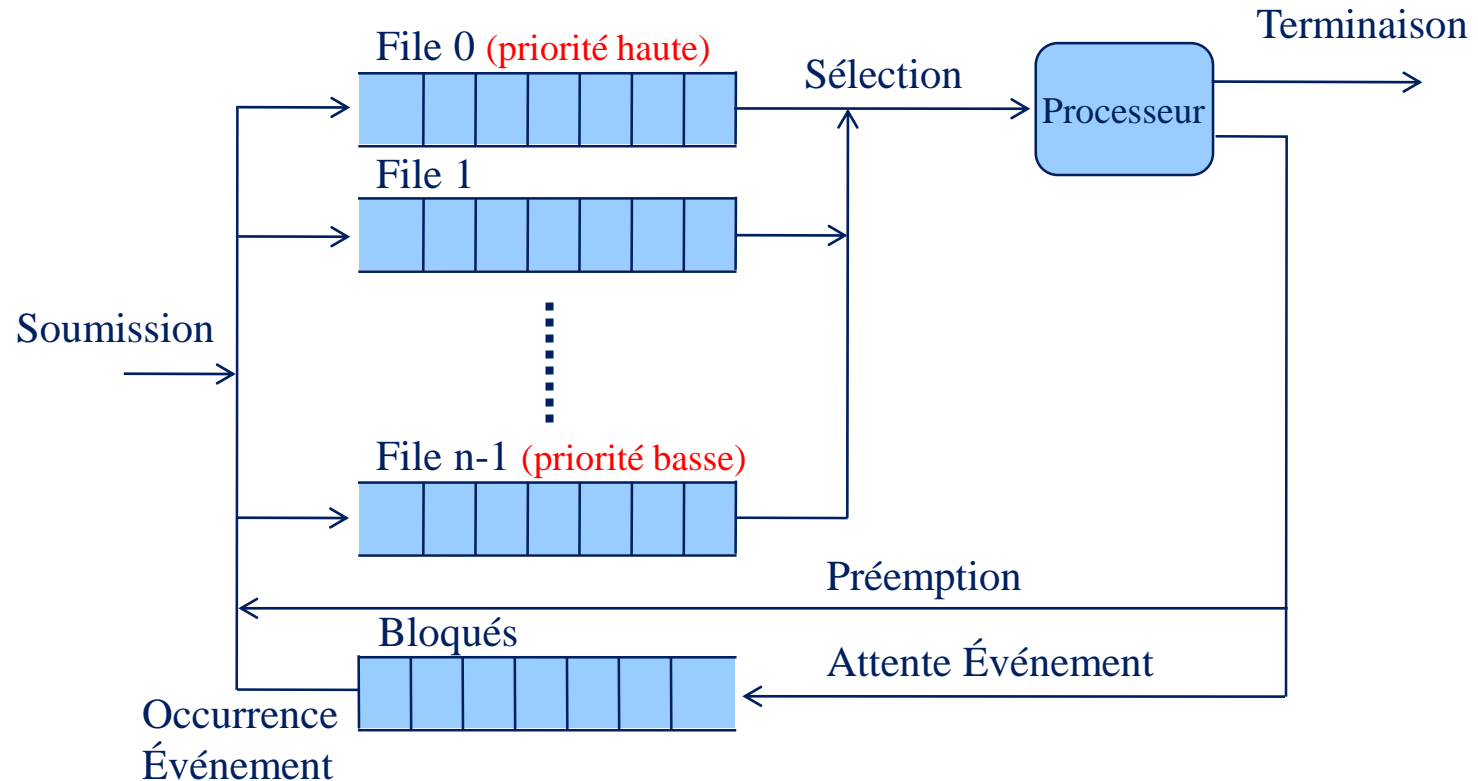


- Temps de réponse moyen = 8,33 ms

Ordonnancement à Files Multiniveaux

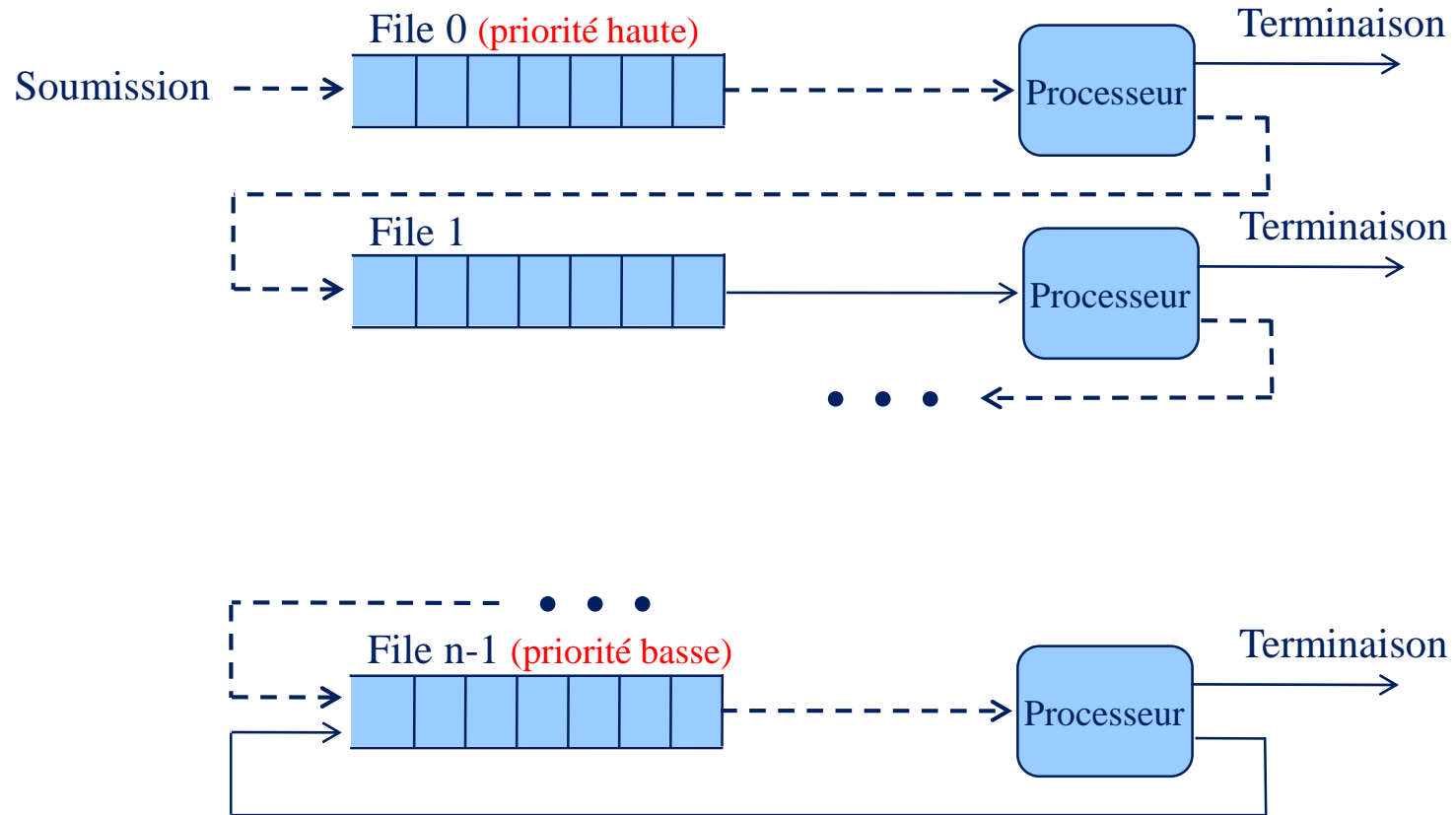
- Objectif : donner la possibilité de classer en différents groupes les processus et satisfaire chacun des groupes en fonction de ses besoins
 - Par exemple, les processus en avant-plan et les processus en arrière-plan qui ont des besoins de temps de réponse différents, ...
- Principe :
 - Partitionner la file des processus prêts en plusieurs files séparées
 - Les processus sont affectés à une file (en fonction du type ou de la priorité des processus, par exemple)
 - Le processus est toujours affecté à la même file
 - Le processus change de file au cours de son exécution
 - Chaque file possède son propre algorithme d'ordonnancement
 - Établir un ordonnancement entre les files (généralement, ordonnancement préemptif à priorité fixée)
- Implémentation → une file par niveau

Ordonnancement avec Priorités Statiques



Risque de famine si les files + prioritaires sont toujours approvisionnées.

Ordonnancement avec Priorités Dynamiques



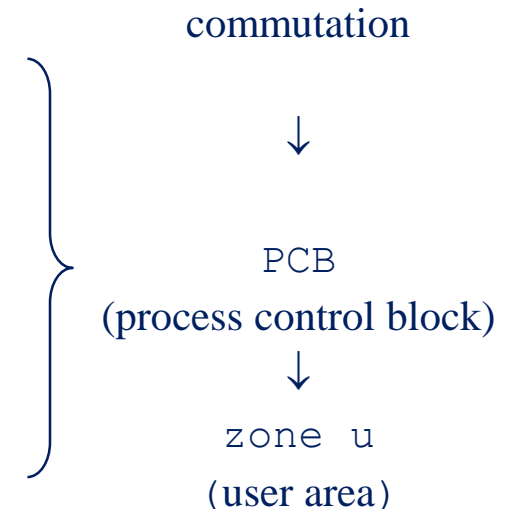
Remonter régulièrement (x quanta) d'un niveau toutes les tâches pour éviter la famine.

Ordonnancement traditionnel Unix

- Ordonnancement basé sur les priorités
- Chaque processus a une priorité qui change dans le temps
- Le *scheduler* sélectionne toujours le processus prêt de plus haute priorité
- Ordonnancement des processus de même priorité par préemption et modification dynamique de leur priorité selon leur utilisation de la CPU
- Si un processus de plus haute priorité devient prêt, préemption du processus courant même s'il n'a pas entièrement consommé son quantum
- Le noyau Unix traditionnel est non-préemptif
 - un processus en mode noyau ne peut être forcé à céder la CPU pour un processus plus prioritaire
 - le processus courant peut volontairement libérer la CPU lorsqu'il doit bloquer sur une ressource, ou bien il est préempté lorsqu'il retourne en mode utilisateur

Contexte d'un processus

- Espace d'adressage utilisateur
 - code, données, pile user, segments de mémoire partagée, ...
- Informations de contrôle
 - zone `u` et structure `proc`
- Appartenance (credentials)
 - `UID`, `GID`
- Variables d'environnement
 - “variable = valeur”
- Contexte matériel
 - Compteur ordinal (program counter-PC),
 - Pointeur de pile (stack pointer-SP),
 - Mot d'état du processeur (processor status word-PSW):
état du système, mode d'exécution,
niveau de priorité d'interruption, ...
 - Registres de gestion mémoire
 - registres FPU (floating point unit)



Zone u (user)

- `struct user` → `<sys/user.h>`
- Fait partie de l'espace du processus → swappable
- mappable et visible uniquement lorsque le processus s'exécute (référence : `u`)
 - process control block (`pcb`),
 - pointeur vers la structure `proc` du processus,
 - `UID` et `GID` effectifs et réels,
 - arguments, valeurs de retour ou code d'erreur de l'appel système courant,
 - informations sur les signaux (gestionnaires, ...),
 - entête du programme (tailles du code, des données et de la pile ainsi que des information sur la gestion mémoire),
 - table des descripteurs de fichiers ouverts (extensible dynamiquement : `SVR4`),
 - pointeurs vers `vnodes` du répertoire courant et du terminal de contrôle,
 - statistiques d'utilisation CPU, quotas disques, limites des ressources,
 - pile système (dans certaines implémentations),
 - ...

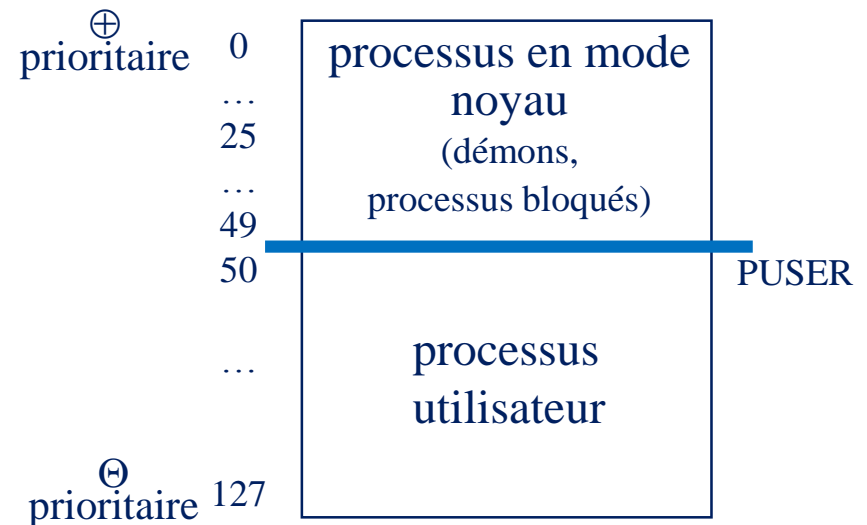
Structure `proc`

- `struct proc → proc.h`
- Appartient à l'espace noyau → visible par le noyau à tout instant, même quand le processus n'est pas en cours d'exécution,
 - identification : `PID, GID, [SID]`,
 - pointeur zone `u`,
 - état courant du processus,
 - pointeurs (avant, arrière) vers listes de processus prêts, bloqués, ...
 - événement bloquant,
 - priorité + information d'ordonnancement,
 - masque des signaux,
 - information mémoire,
 - ...

Priorités des processus

- La priorité d'un processus peut être une valeur entière comprise entre 0 et 127
- La structure `proc` contient les champs (relatifs à la priorité) suivants :

| | | |
|---|------------------------|--|
| - | <code>p_pri</code> | Priorité (d'ordonnancement) courante |
| - | <code>p_usrpri</code> | Priorité en mode utilisateur (égale à <code>p_pri</code> en mode user) |
| - | <code>p_cpu</code> | Mesure de l'utilisation récente de la CPU |
| - | <code>p_nice</code> | Incrément de priorité contrôlable par l'utilisateur |
| - | <code>p_slptime</code> | Temps passé par le processus dans l'état endormi |



Calcul de Priorité

- Répartir équitablement le processeur →
 - baisser la priorité des processus lors de l'accès au processeur
- A chaque *tick*, la routine horloge incrémente `p_cpu` pour le processus courant (`p_cpu++`)
- Régulièrement, le noyau appelle `schedcpu()` pour réduire la valeur de `p_cpu`
 - appelée 1 fois par seconde
 - Pour tous les processus prêts :
`p_cpu = p_cpu * decay`

`decay = 1/2`

SVR3

`decay = (2*load_average) / (2*load_average+1)`

4.3BSD

`p_usrpri = PUSER + (p_cpu/4) + (2*p_nice)`

- Évite la famine des processus de faible priorité
- Favorise les processus qui font des E/S par rapport aux processus qui font du calcul
- Un processus qui a eu un accès récent → `p_cpu` élevé → `p_usrpri` élevé.

Les primitives internes

C'est toujours le processus de plus haute priorité qui s'exécute, à moins que le processus courant soit en mode noyau

Si tous les processus prêts sont dans des files de moindre priorité, le processus courant continue son exécution même si son quantum de temps a expiré

- Après 4 *ticks*, appel de *setpriority()* pour mettre à jour la priorité du processus courant
- 1 fois par seconde, appel de *schedcpu()* pour la mise à jour des priorités de tous les processus
 - retire le processus de la file, recalcule la priorité, remet le processus dans la file (éventuellement différente)
- *roundrobin()* appelée en fin de quantum pour élire un nouveau processus dans la même file

Exemple

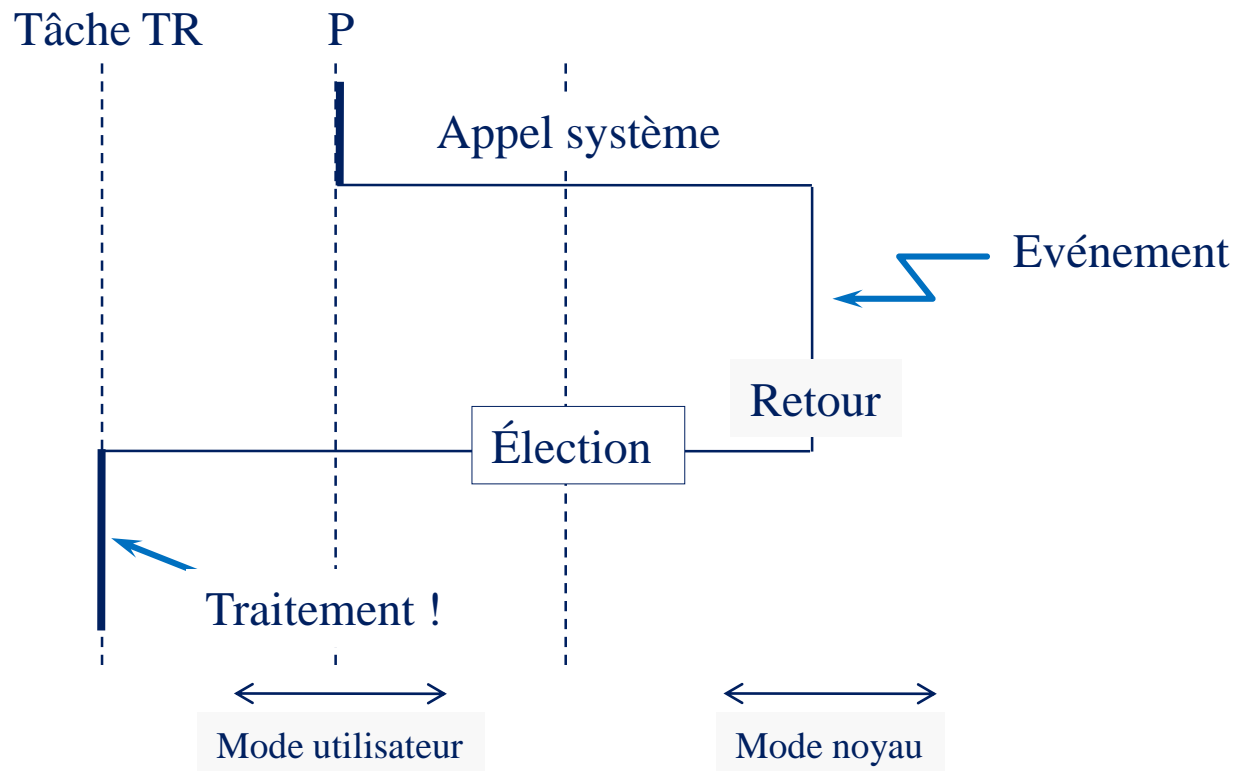
Supposons :

- Trois processus A, B et C
- Créés au même instant avec priorité de base = 60 (PUSER)
- Interruption horloge, 60 fois par seconde
- Aucun processus ne bloque par lui-même
- Aucun autre processus n'est prêt à s'exécuter
- Calcule de priorité :
 - $p_cpu = p_cpu * \frac{1}{2}$
 - $p_usrpri = PUSER + (p_cpu/2)$

| Time | Processus A | | Processus B | | Processus C | |
|------|-------------|------------------|-------------|------------------|-------------|------------------|
| | Priority | CPU Count | Priority | CPU Count | Priority | CPU Count |
| 0 | 60 | 0 1 2 . | 60 | 0 | 60 | 0 |
| 1 | 75 | 30 | 60 | 0 1 2 . | 60 | 0 |
| 2 | 67 | 15 | 75 | 30 | 60 | 0 1 2 . |
| 3 | 63 | 7 8 9 . | 67 | 15 | 75 | 30 |
| 4 | 76 | 33 | 63 | 7 8 9 . | 67 | 15 |
| 5 | 68 | 16 | 76 | 33 | 63 | 7 8 9 . |
| 6 | | | | | 67 | |

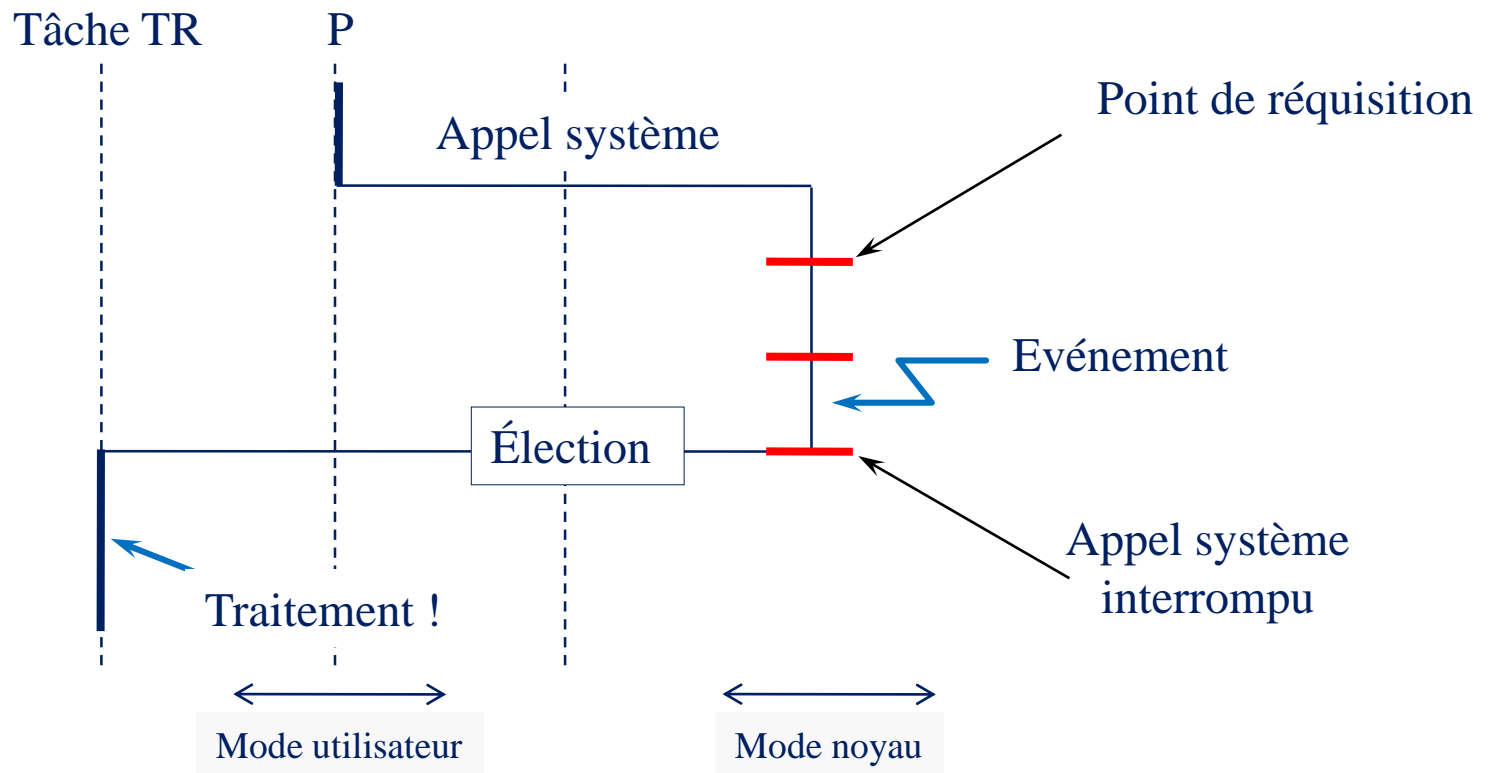
Contraintes Temps Réel

- Objectif : satisfaire des contraintes de temps
 - Processus temps réel très prioritaire en attente d'événement
- Impossible dans la plupart des Unix car noyau non-préemptif !



Points de réquisition

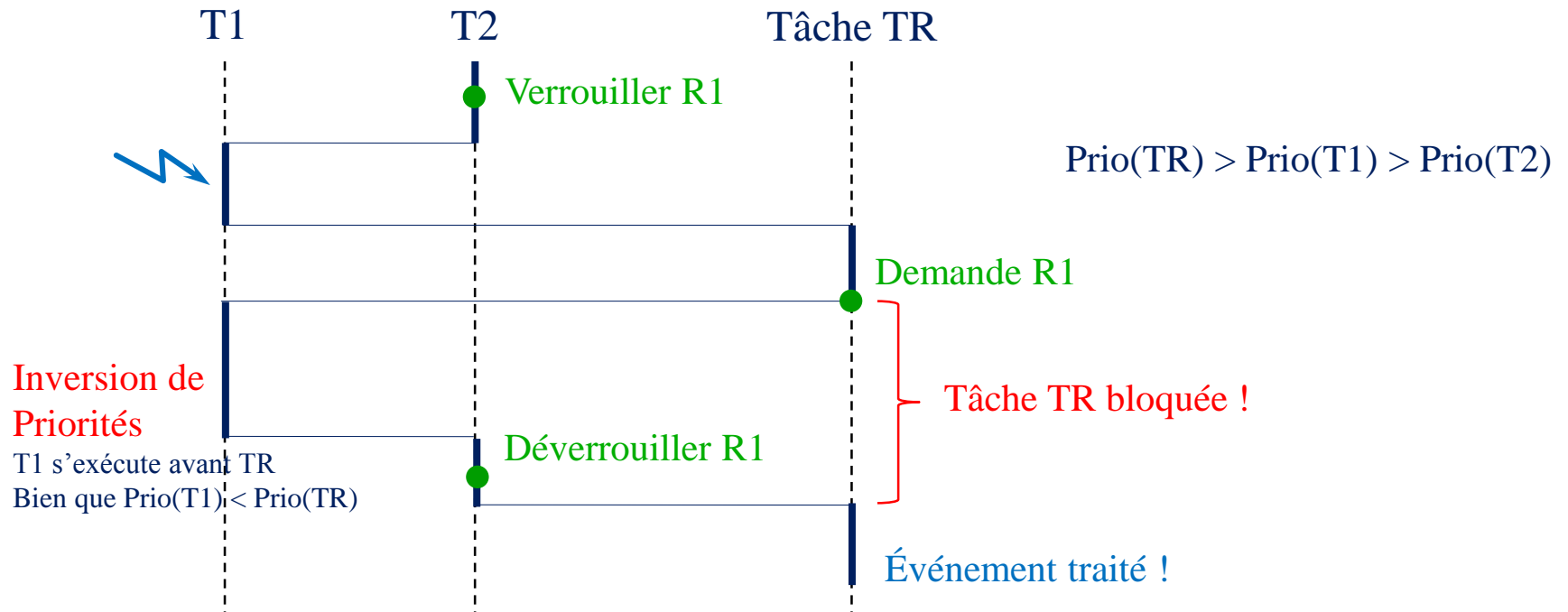
- Solution 1 : vérifier régulièrement si un processus plus prioritaire doit être exécuté (SVR4)



- Il est difficile de placer de nombreux points de réquisition
⇒ latence de traitement importante

Noyaux Préemptifs (Solaris 2.x)

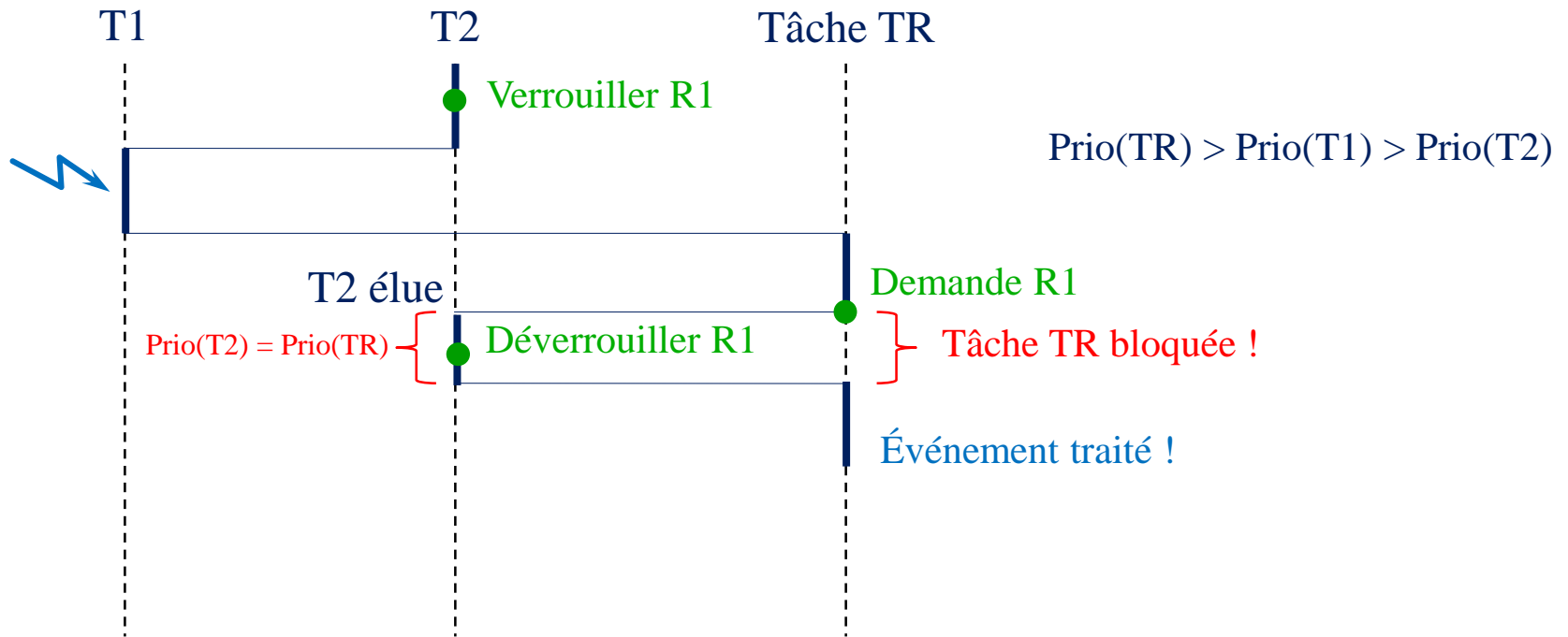
- Solution 2 : rendre le noyau préemptif
⇒ en mode noyau l'exécution peut être interrompue par des processus plus prioritaires
- Protéger toutes les structures de donnée du noyau par des verrous (~ sémaphores)



Tâche TR dépendant de l'exécution de la tâche T2 !

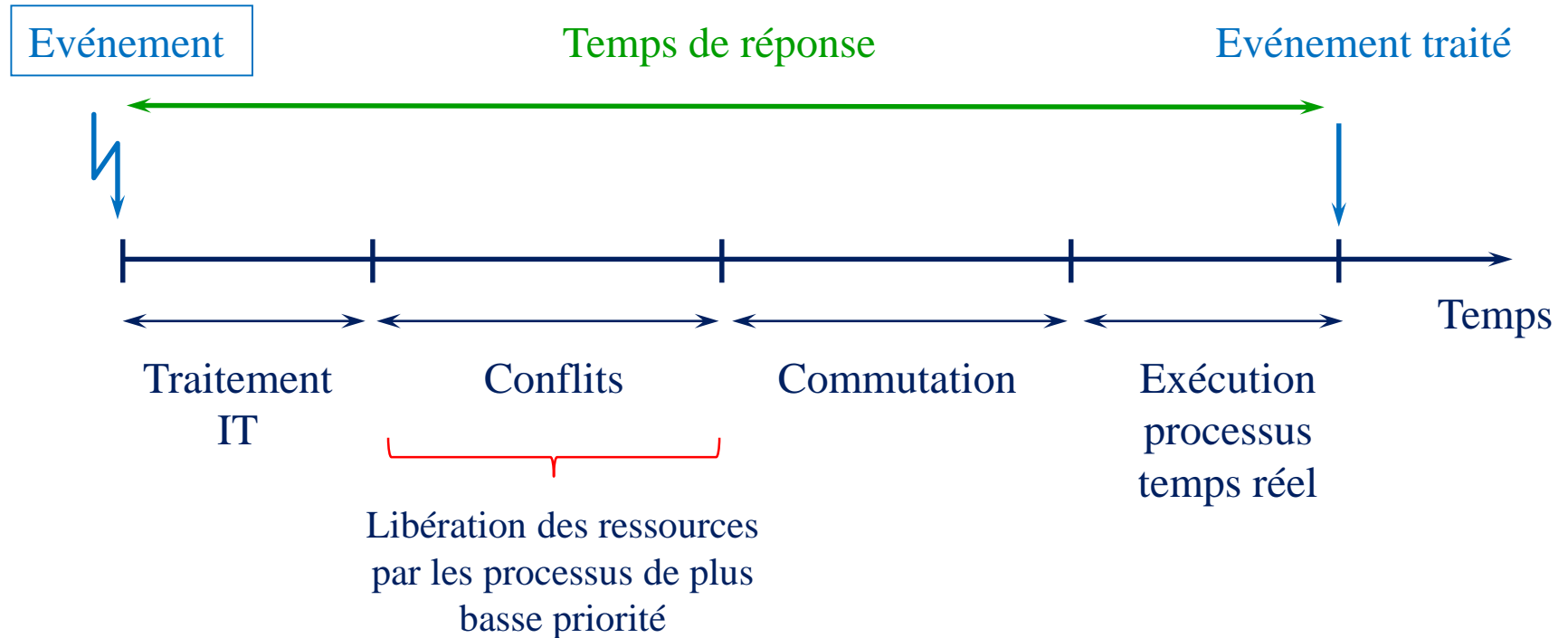
Héritage de priorité

- Solution : Donner à la tâche qui possède la ressource la priorité de la tâche temps réel



T2 hérite temporairement de la priorité de TR

Temps de réponse

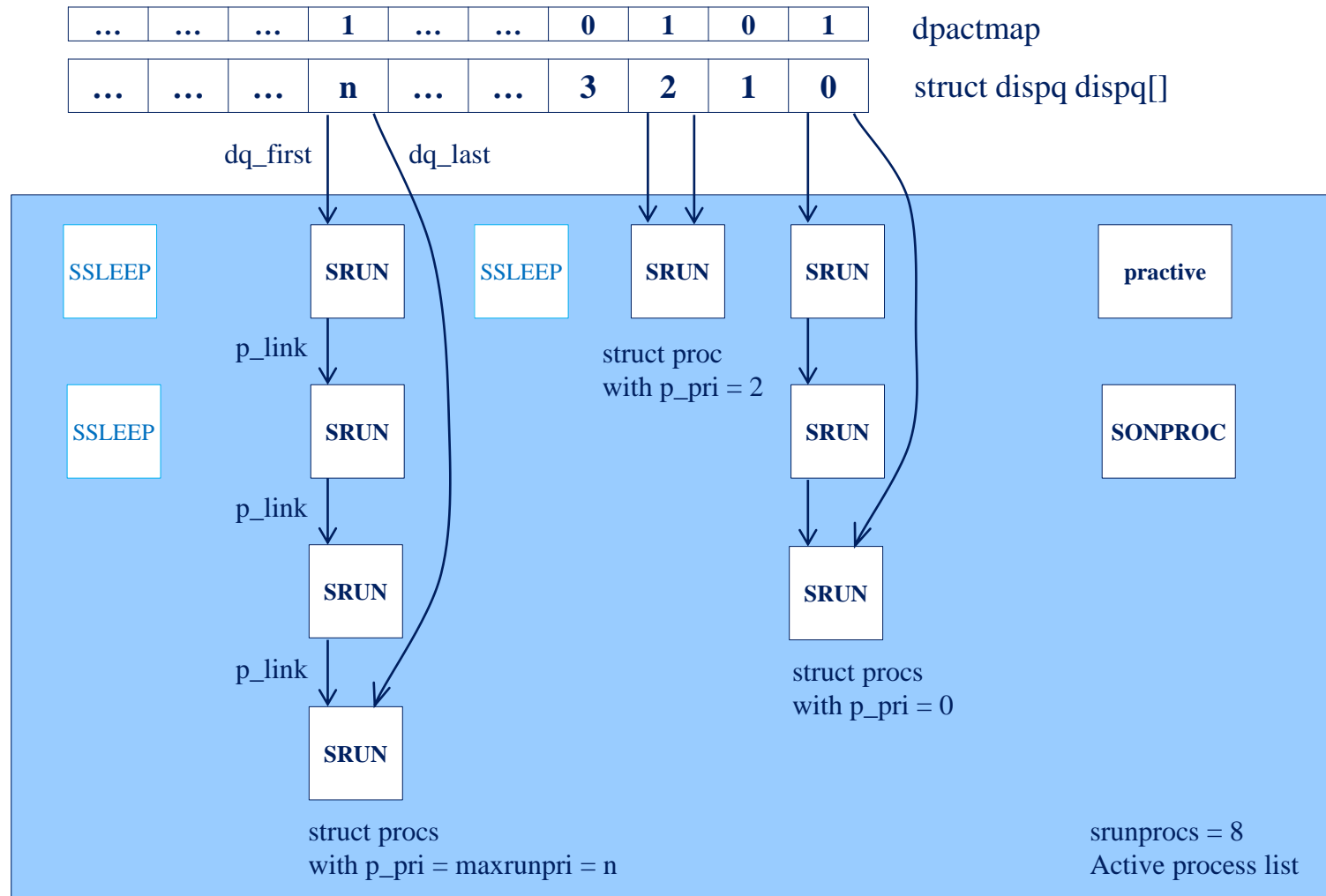


A decorative border composed of a grid of colored dots in various colors including purple, blue, green, yellow, red, and brown, framing the central text.

Ordonnancement Unix : Implémentation

(Hors Programme)

Implémentation des Files de Processus Prêts

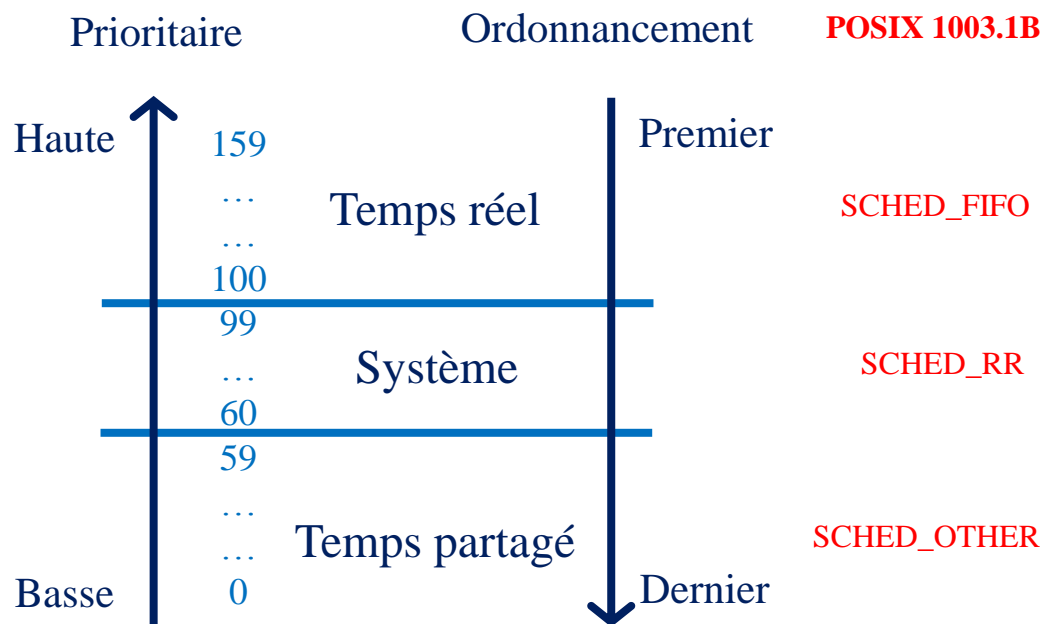


Algorithme de *switch* ()

- Il existe trois situations où la commutation est indiquée
 - Le processus courant bloque sur une ressource ou se termine. Il s'agit d'une commutation de contexte volontaire.
 - la procédure de recalcul de priorité résulte en l'existence d'un autre processus de priorité plus élevée que celle du processus courant.
 - Le processus courant, ou un gestionnaire d'interruption, réveille un processus de priorité plus élevée.
- Pour la commutation non volontaire, le noyau utilise un flag (*runrun*)
Sur retour en mode utilisateur, si *runrun* positionné → appel de *switch* ()
- Algorithme de *switch* ()
 - Trouver le premier bit positionné dans *whichqs*
 - Retirer le processus le plus prioritaire de la tête de file
 - Effectuer la commutation :
 - Sauvegarder le PCB (Process Control Bloc) du processus courant (inclus dans zone *u*)
 - Changer la valeur du registre de la table des pages (champs *p_addr* de struct *proc*)
 - Charger les registres du processus élu à partir de la zone *u*

SVR4 : Classes d'ordonnancement

- 3 classes de priorités sont définies

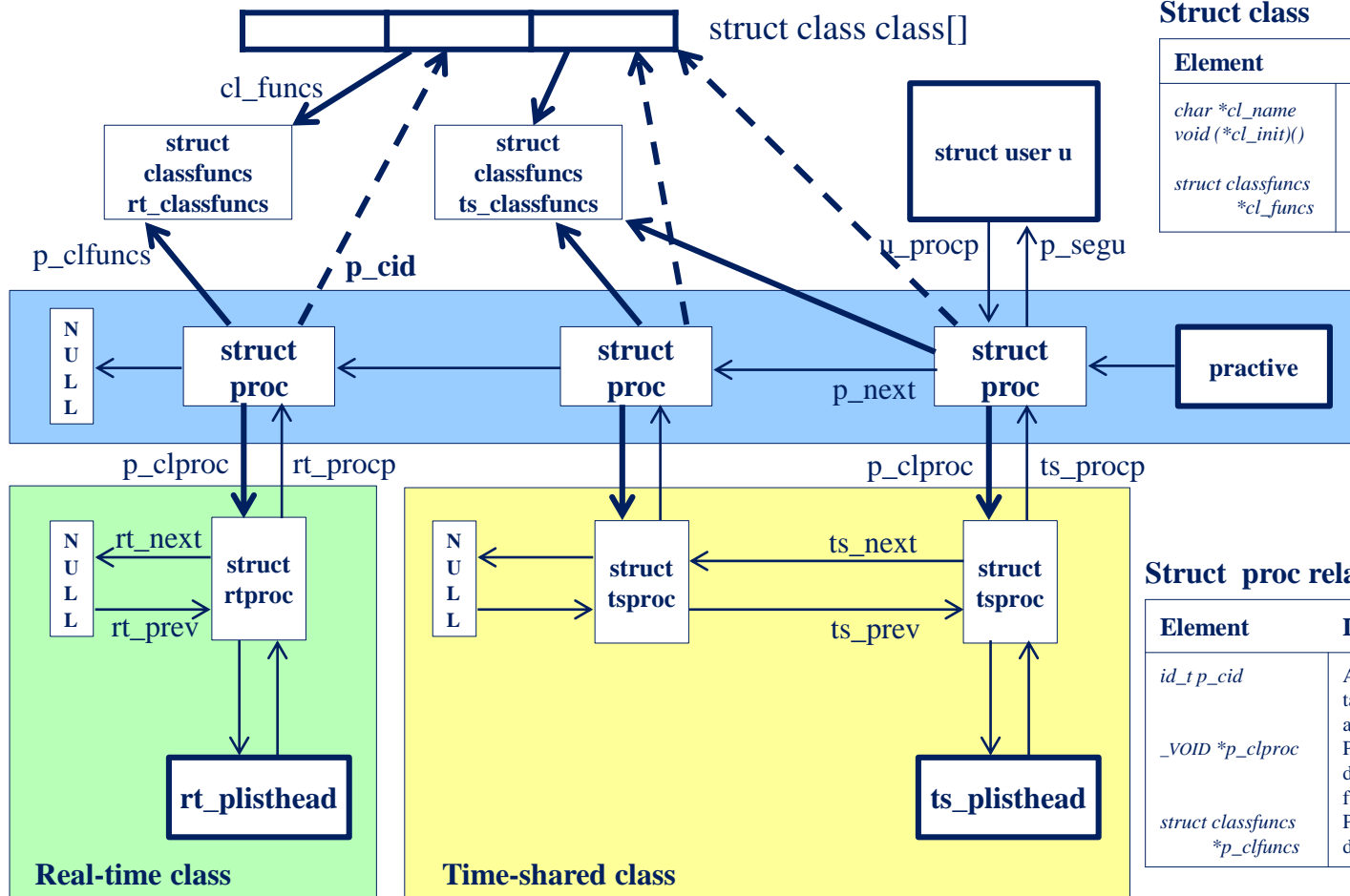


Class independent variables

| Variable | Description |
|------------------------|---|
| <i>int runrun</i> | Set to cause preemption |
| <i>int kprunrun</i> | Set to preempt at next kernel preemption point |
| <i>int curpri</i> | Priority of current process (ie., <i>u.u_procp->p_pri</i>) |
| <i>int maxrunpri</i> | Priority of highest priority active queue |
| <i>int srunprocs</i> | Total number of loaded, runnable processes waiting to be scheduled |
| <i>proc_t *curproc</i> | Pointer to the currently running process (ie., <i>u.u_procp</i>) |
| <i>dispq_t *dispq</i> | Pointer to array of dispatch queues indexed by <i>p_pri</i> |
| <i>ulong *dpactmap</i> | Points to an array of <i>dispq</i> active bitmaps, 1 per dispatch queue |

- Les processus temps réel prêts s'exécutent tant qu'ils restent prêts
- Définition des processus temps réel réservée au superviseur (appel système *prionctl* SVR5 - *setsched_prio* POSIX)

SVR4 : Structures



Struct class

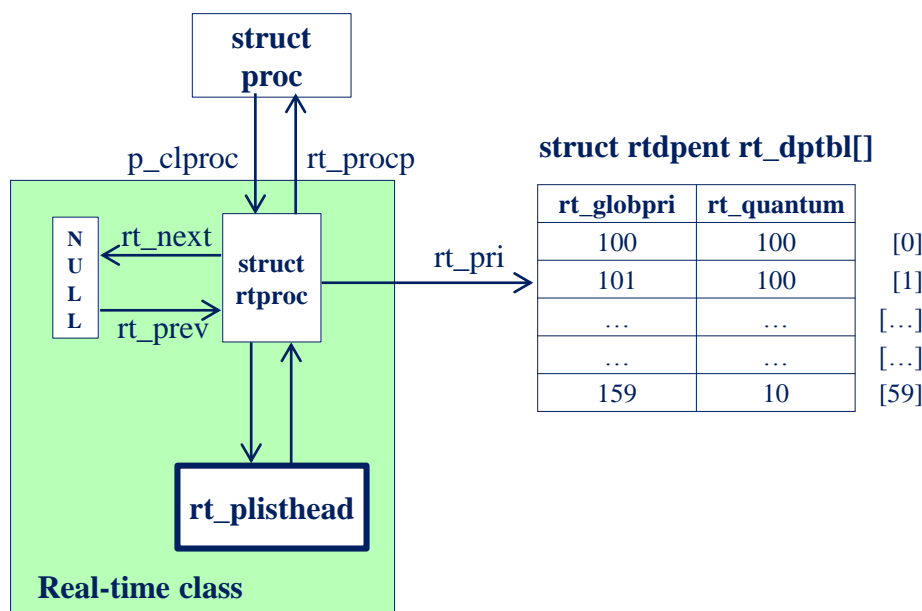
| Element | Description |
|--|--|
| <code>char *cl_name</code> | String containing the priority class name |
| <code>void (*cl_init)()</code> | Pointer to the class initialization function called at boot time |
| <code>struct classfuncs *cl_funcs</code> | Pointer to the structure containing function pointers to the class dependent functions |

Struct proc related priority class members

| Element | Description |
|---|--|
| <code>id_t p_cid</code> | An integer used to index into the <code>class[]</code> table for the class to which the process is associated with |
| <code>_VOID *p_clproc</code> | Pointer to a class dependent data structure defined by the priority class dependent functions |
| <code>struct classfuncs *p_clfuncs</code> | Pointer to the same classfuncs structure described in the priority class table |

Classe Temps Réel

- Politique d'ordonnancement à priorités fixes



Struct rtdpent

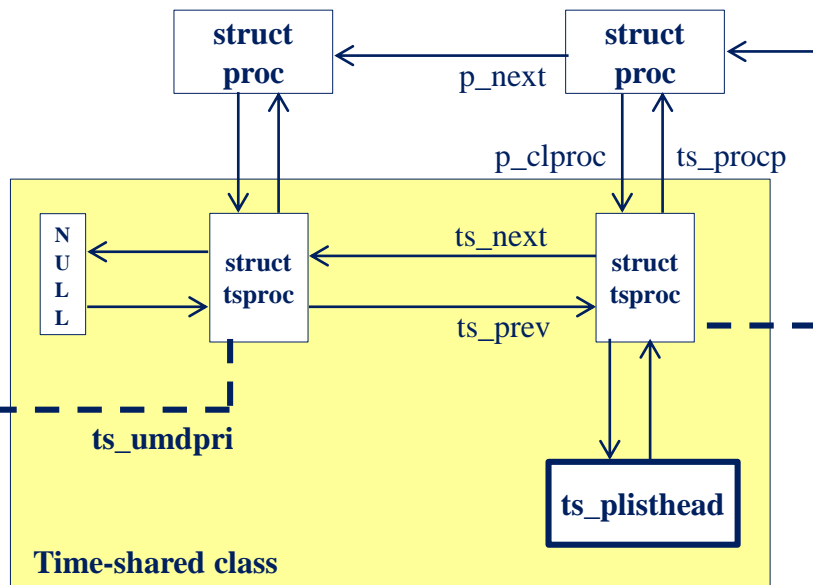
| Element | Description |
|------------------------|--|
| <i>int rt_globpri</i> | Default global priority (from 100 – 159) |
| <i>long rt_quantum</i> | Default time-quantum (slice) for this level specified in clock ticks |

Struct rtproc

| Element | Description |
|-------------------------------|--|
| <i>long rt_pquantum</i> | Time-quantum assigned to this process |
| <i>long rt_timeleft</i> | Time remaining in time-quantum |
| <i>short rt_pri</i> | Priority level (offset into <i>rt_dptbl[]</i>) |
| <i>ushort rt_flags</i> | Flags : <i>RTRAN</i> – process has run since last swap out, <i>RTBACKQ</i> – process is placed on the back of its dispatch queue when next preempted |
| <i>struct proc *rt_procp</i> | Pointer to process in the active list |
| <i>char *rt_pstatp</i> | Pointer to <i>p-stat</i> in <i>rt_procp</i> |
| <i>int *rt_pprip</i> | Pointer to <i>p_pri</i> in <i>rt_procp</i> |
| <i>uint *rt_pflagp</i> | Pointer to <i>p_flag</i> in <i>rt_procp</i> |
| <i>struct rtproc *rt_next</i> | Pointer to next <i>rtproc</i> in list |
| <i>struct rtproc *rt_prev</i> | Pointer to previous <i>rtproc</i> in list |

Classe Temps Partagé

- Quantum variable d'un processus à l'autre
- Inversement proportionnel à la priorité !
- Définis statiquement pour chaque niveau de priorités



struct tsdpent ts_dptbl[]

| ts_globpri | ts_quantum | ts_tqexp | ts_slpret | ts_maxwait | ts_lwait | |
|------------|------------|----------|-----------|------------|----------|-------|
| 0 | 100 | 0 | 10 | 5 | 10 | [0] |
| 1 | 100 | 0 | 11 | 5 | 11 | [1] |
| ... | ... | ... | ... | ... | ... | [...] |
| ... | ... | ... | ... | ... | ... | [...] |
| 59 | 10 | 49 | 59 | 5 | 59 | [59] |

Struct tsdpent

| Element | Description |
|-------------------------|---|
| <i>int ts_globpri</i> | Default global priority (from 0 – 59) |
| <i>long ts_quantum</i> | Default time-quantum (slice) for this level specified in clock ticks |
| <i>short ts_tqexp</i> | Assigned value to <i>ts_cpupri</i> when this process's time-quantum expires |
| <i>short ts_slpret</i> | Assigned value to <i>ts_curpri</i> upon returning to user-mode after a sleep |
| <i>short ts_maxwait</i> | Maximum number of consecutive seconds that the process can run in its allocated time-quantum before assigning <i>ts_lwait</i> to <i>ts_cpupri</i> |
| <i>short ts_lwait</i> | Assigned value to <i>ts_cpupri</i> if <i>ts_dispwait</i> exceeds <i>ts_maxwait</i> |

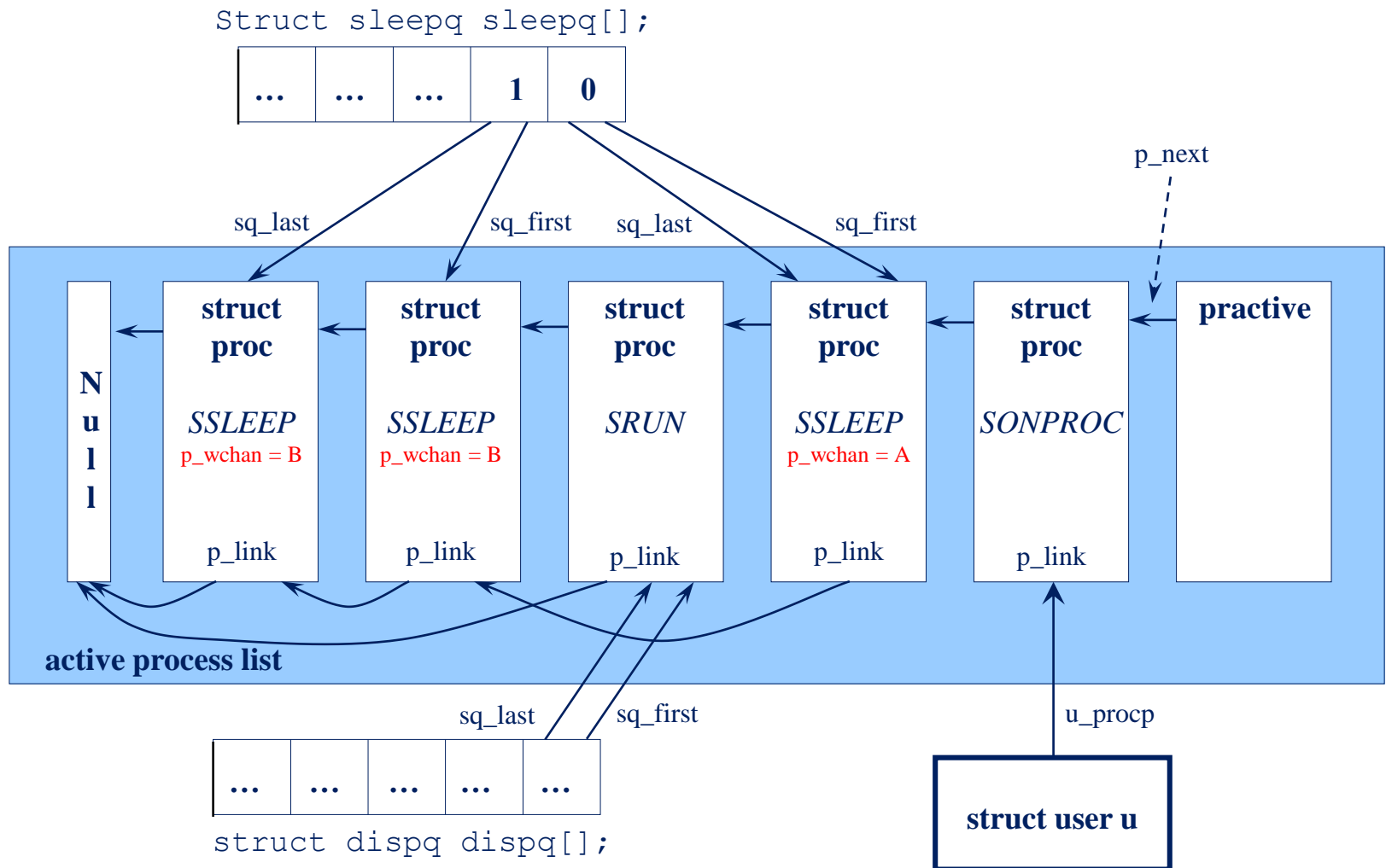
Struct tsproc

| Element | Description |
|-------------------------------|---|
| <i>long ts_timeleft</i> | Remaining time in process's time-quantum |
| <i>short ts_cpupri</i> | Kernel portion of priority value |
| <i>short ts_uprilim</i> | User modifiable – priority value limit, set with <i>priocntl</i> |
| <i>short ts_upri</i> | User modifiable – current user portion of priority value |
| <i>short ts_umdpr</i> | Offsets into <i>ts_dptbl[]</i> to find the global priority level |
| <i>char ts_nice</i> | Nice value (for backward compatibility) |
| <i>unsigned char ts_flag</i> | Operational flags |
| <i>short ts_dispwait</i> | Number of seconds since start of current time-quantum (not reset upon preemption) |
| <i>struct proc *ts_procp</i> | Pointer to process's <i>proc</i> structure |
| <i>char *ts_pstatp</i> | pointer to <i>p_stat</i> in <i>proc</i> structure |
| <i>int *ts_pprip</i> | Pointer to <i>p_pri</i> in <i>proc</i> structure |
| <i>uint *ts_pflagp</i> | Pointer to <i>p_flag</i> in <i>proc</i> structure |
| <i>struct tsproc *ts_next</i> | Link to next <i>tsproc</i> on <i>tsproc</i> list |
| <i>struct tsproc *ts_prev</i> | Link to previous <i>tsproc</i> on <i>tsproc</i> list |

Processus Endormis

- Les processus s'endorment volontairement en attente de la disponibilité d'une ressource ou de l'occurrence d'un événement
 - Inode verrouillé, lecture (écriture) dans un pipe vide (plein), achèvement d'une E/S disque, Lecture du terminal, attente de mémoire (défaut de page)
- Les processus s'endorment sur des événements
 - Événements qui permettent aux processus de se réveiller par un signal
 - Événements qui font que les signaux sont ignorés (processus endormis indéfiniment jusqu'à l'occurrence de l'événement)
- Les processus (temps partagé) sont endormis avec une haute priorité \neq priorité utilisateur ($p_pri \neq p_usrpri$) \rightarrow Au réveil le processus a une plus grande probabilité d'être élu
- Au passage en mode utilisateur, l'ancienne priorité est restaurée ($p_pri = p_usrpri$)

File des processus endormis



Priorités Système

- Les priorités système sont décrites dans la table *ts_kmdpris[]*

```
int ts_kmdpris[] = {
    60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
    70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
    80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
    90, 91, 92, 93, 94, 95, 96, 97, 98, 99
};
```

40 nombres contigus représentant les niveaux de priorités de la classe système.
Divisés en intervalles appelés niveaux de priorités système .
Chaque niveau de priorités système correspond à un type d'événements.

- Chaque processus devant s'endormir, doit préciser le type d'événement sur lequel il s'endort

Niveaux de priorités système

| Sleep parameter | Parameter value | Global priority value | Description |
|-----------------|-----------------|-----------------------|---------------------------------|
| PSWP | 0 | 99 | Swapper priority |
| PMEM | 0 | 99 | Locked memory |
| PINOD | 10 | 89 | Locked inode |
| PRIBIO | 20 | 79 | Block I/O |
| PZERO | 25 | 74 | Signal level |
| PPIPE | 26 | 73 | Empty or full pipe |
| PVFS | 27 | 72 | Wait for unmount of file system |
| TTIPRI | 28 | 71 | Terminal input |
| TTOPRI | 29 | 70 | Terminal output |
| PWAIT | 30 | 69 | Wait for SIGCHLD |
| PSLEP | 39 | 60 | Pause for signal |

Plus haute priorité système (type d'événement) pouvant être passée en paramètre à *sleep()*

```
short ts_maxkmdpri =
    (sizeof(ts_kmdpris)/4)-1;
```

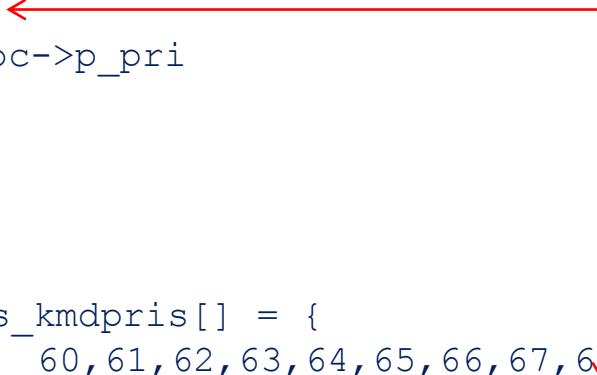
Nouvelle priorité du processus :

```
*tsproc->ts_pprip =
    ts_kmdpris[ts_maxkmdpri
    - valpar];
```

Exemple de Calcul de priorité

Processus \rightarrow `sleep(PRIBIO + 1)`

Nouvelle priorité du processus

```
*tsproc->ts_pprip =  
ts_kmdpris[ts_maxkmdpri - (PRIBIO + 1)]  
== ts_kmdpris[18]  
== 78   
== proc->p_pri
```

```
int ts_kmdpris[] = {  
    60,61,62,63,64,65,66,67,68,69,  
    70,71,72,73,74,75,76,77,78,79,  
    80,81,82,83,84,85,86,87,88,89,  
    90,91,92,93,94,95,96,97,98,99  
};
```

Bibliographie

- The magic garden explained, The internals of Unix System V Release 4
Berny Goodheart & James Cox - Prentice Hall
- Operating Systems, Internals and Design Principles,
Willam Stallings - Printice Hall
- Principes des Systèmes d'Exploitation,
A. Silberschatz, P.B. Galvin, G. Gagne - Vuibert
- Introduction aux Système Unix, Support de cours,
Eric Gressier - CNAM-CEDRIC