Processus et Synchronisation par Files d'Attente

Définition

- Processus

  Instance d'un programme en cours d'exécution

  - Pour le système, un processus c'est
    - Un état d'exécution
    - Un contexte
      - Registres, pile système, informations d'état, tables de pages, ...
      - Implémenté sous la forme d'une structure
    - Une consommation (détention) de ressources
      - Mémoire allouée, Fichiers ouverts, ...

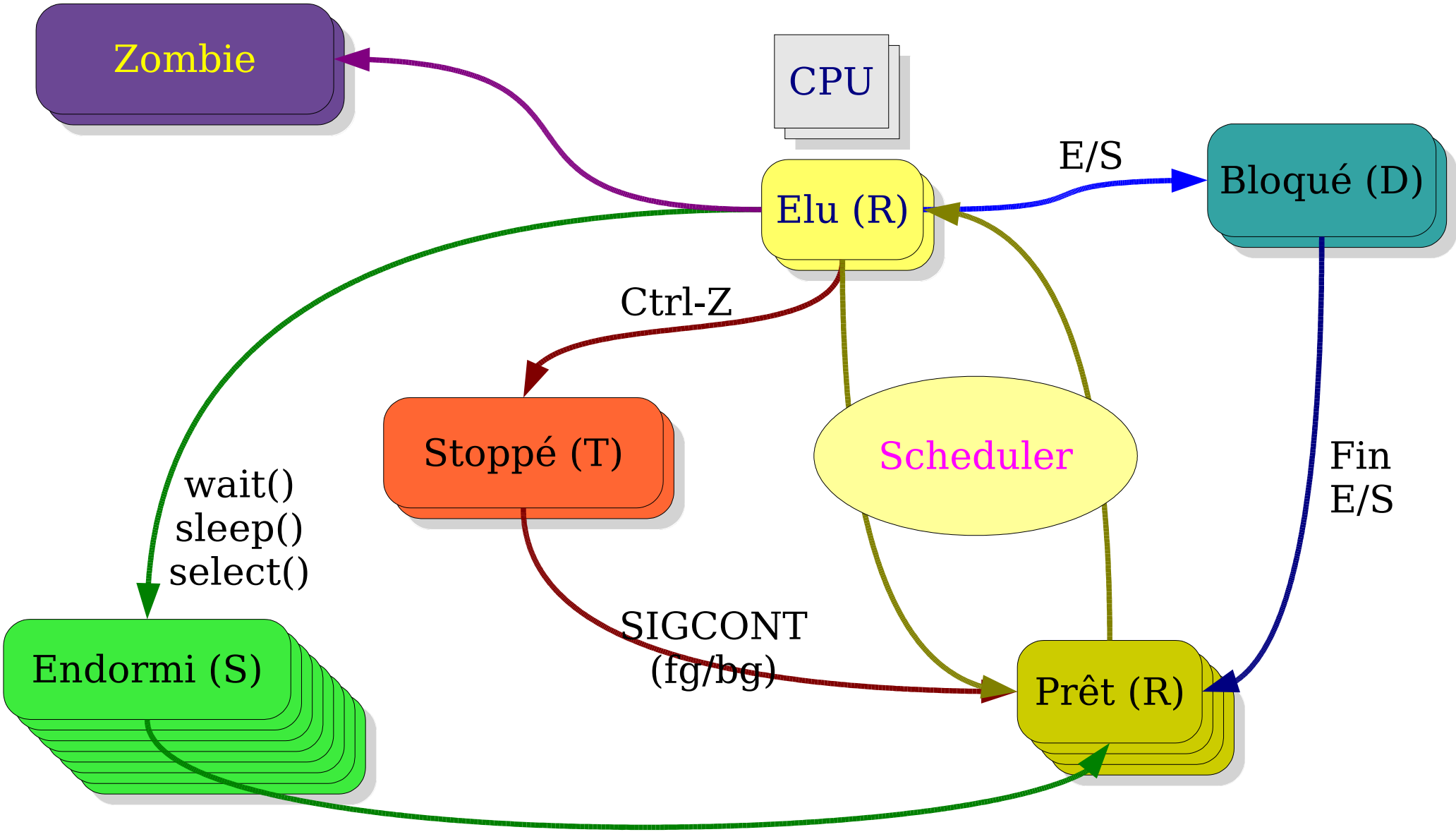
Zoologie des Processus

- Variantes autour de la notion de processus
  - « **Lourd** » : notion classique de processus Unix
    - POSIX : Création par fork() = duplication du contexte
    - Séparation (et protection) de l'espace d'adressage propre
  - « **Leger** » : notion plus récente de « thread »
    - POSIX : Création par pthread_create()
      - partage du contexte
    - Hébergés dans le contexte d'un processus lourd
      - thread main
    - Création plus rapide
    - Partage des données
    - MAIS : problèmes de concurrence…

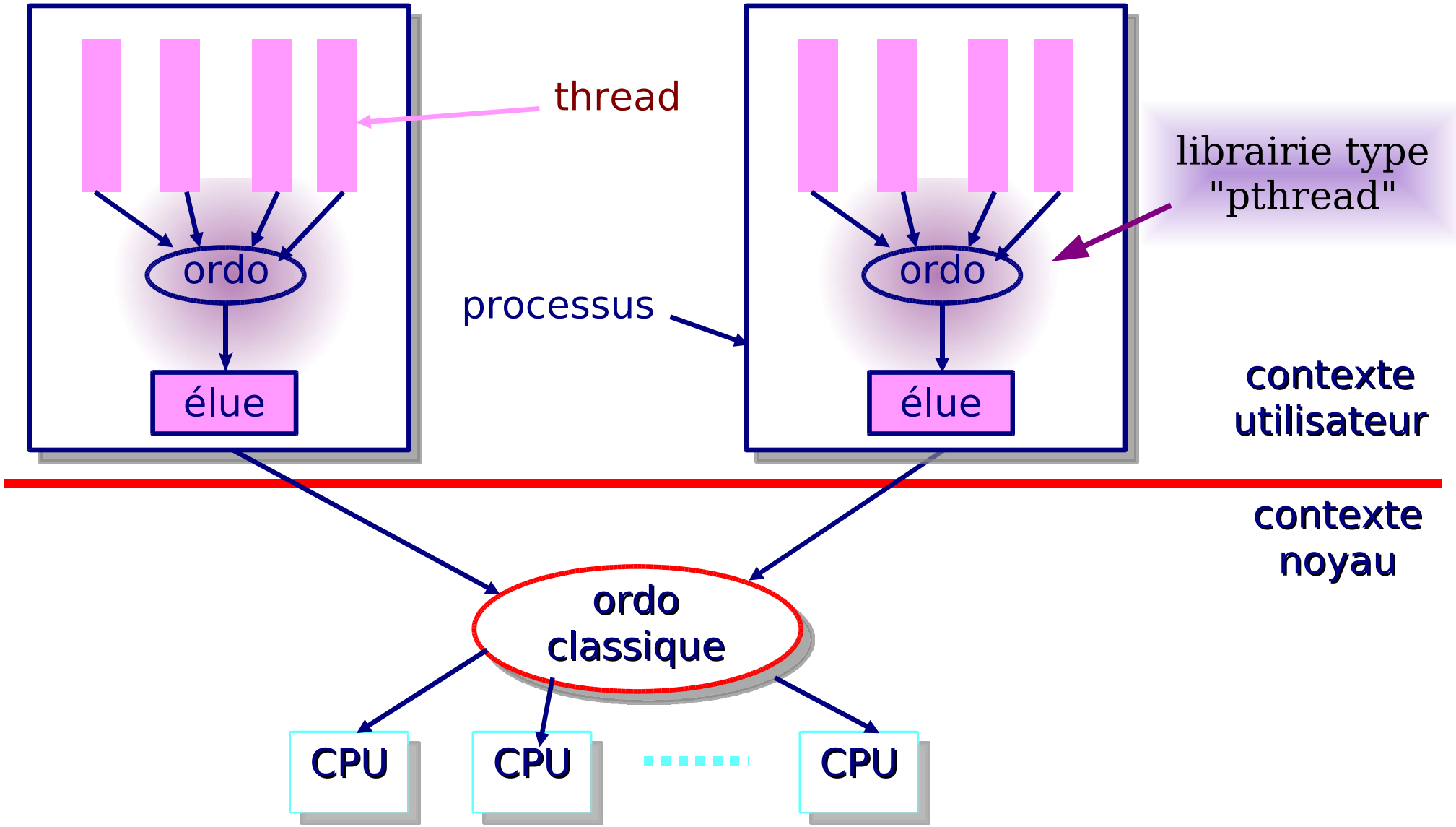
Etats d'un Processus : Vue Utilisateur

- Champ **STAT** de la commande `ps -x` (**interne noyau**)
  - **R** : **TASK_RUNNING**
    - Prêt à s'exécuter ou en cours d'exécution
  - **S** : **TASK_INTERRUPTIBLE**
    - Endormi en attente interruptible (non exclusive) d'un événement
  - **D** : **TASK_UNINTERRUPTIBLE**
    - Endormi en attente non interruptible (exclusive) d'un evènement (typiquement une E/S)
  - **Z** : **TASK_ZOMBIE**
  - **T** : **TASK_STOPPED**
    - Arrêté ou Tracé
  - **W** : Pas de page résidente
  - **N** : Gentillesse positive
  - + Un état supplémentaire implicite : processus élu

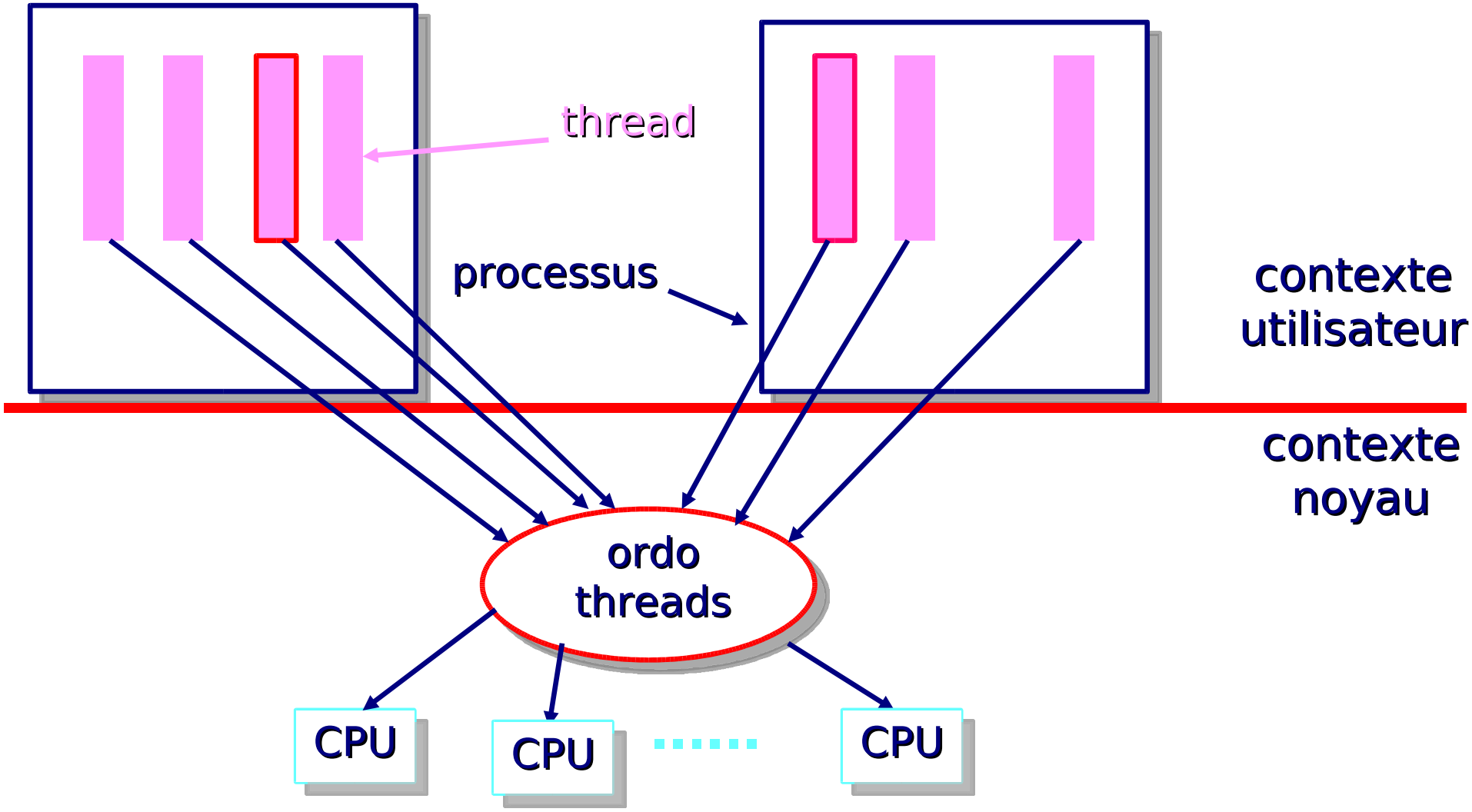
Graphe des Transitions entre Etats



Thread de Niveau Utilisateur



Threads de Niveau Noyau



Quels Type de Processus Supportés par Linux ?

- Threads de niveau noyau
  - Création par l'A.S. `clone(fn,arg,flags)`
    - `fork() et vfork() = dérivés de clone()`
    - flags = options de clonage
      - Choix de ce qui doit être partagé (ou cloné)
      - flags = 0 : processus lourd

Options de Clonage

- Valeurs combinables dans le champ flags de clone()
  - CLONE_VM : espace mémoire
  - CLONE_FS : root, cwd, umask
  - CLONE_FILES : fichiers ouverts
  - CLONE_PARENT : même parent
  - CLONE_PID : même pid (uniquement si parent = 0, lors du boot)
  - CLONE_PTRACE : « ptracé » comme parent
  - CLONE_SIGHAND : même traitants d'interruptions
  - CLONE_THREAD : même groupe que parent (= thread POSIX)
  - CLONE_SIGNAL : CLONE_THREAD + CLOSE_SIGHAND
  - CLONE_VFORK : bloque parent jusqu'à mort ou exec du fils

Threads Internes du Noyau

- Threads particulières :
  - Ne basculent jamais en mode utilisateur
  - Plus légères
    - Pas besoin de les encombrer de ce qui concerne le mode utilisateur
  - Utiles pour traîter certains travaux en tâche de fond
- Création :
  - `int kernel_thread(fn, arg,flags)`

Quelques threads noyau usuelles

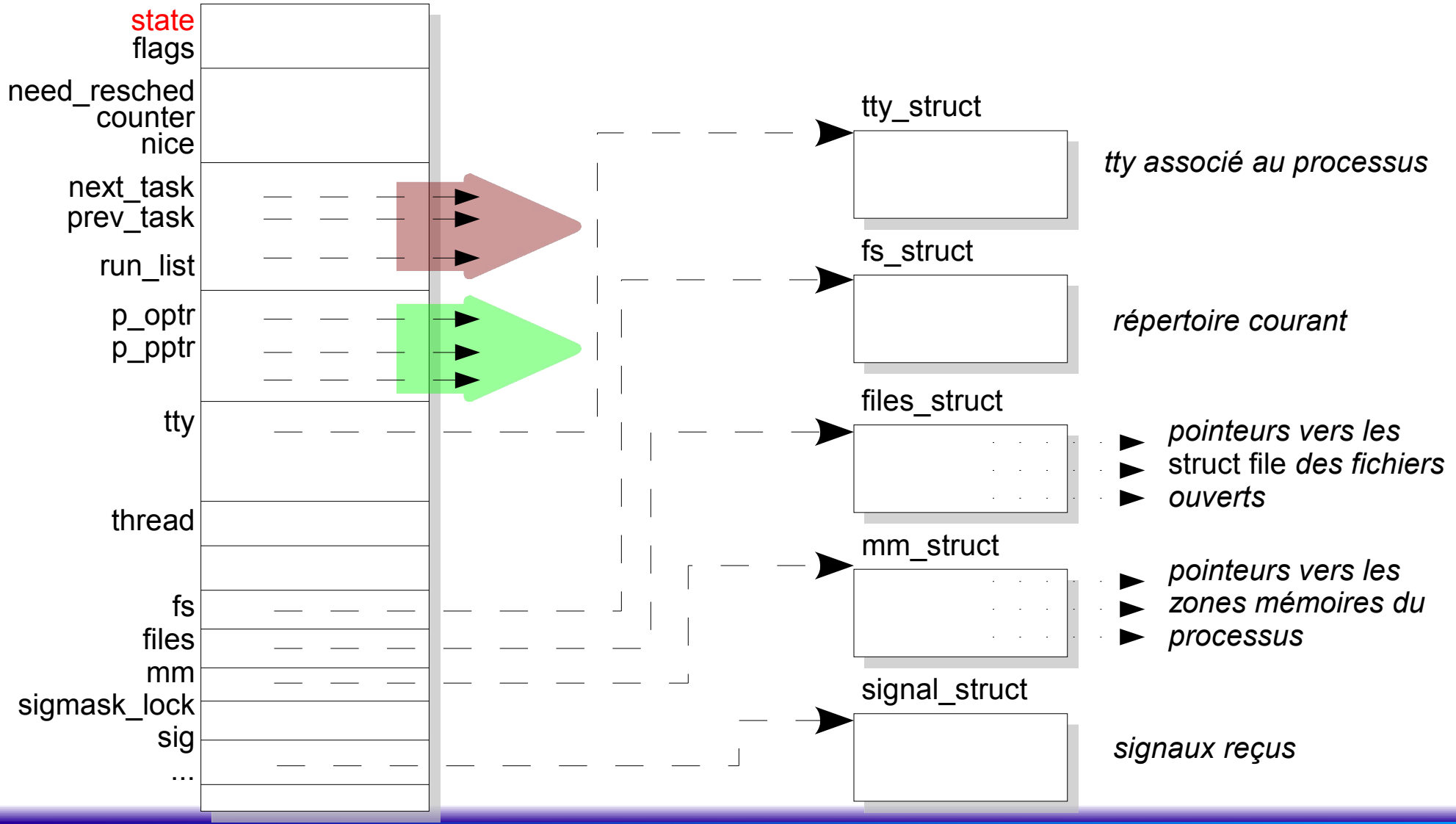
- `keventd` : execute task queue qt_context
  - On reparle au chapitre 8 …
- `kapm` : événements liés à APM
- `kswapd` : collecte mémoire
- `kflushd/bdflush` : évacue les pages « sales » du cache disque pour récupérer de la mémoire
- `kupdated` : evacue aussi page sales, mais pour éviter trop de pertes en cas de crash
- `ksoftirqd` : exécute les tasklets (un par CPU)
  - On en reparle aussi au chapitre 8…

Deux Thread Noyau Particulières : 0 et 1

- Thread de pid 0 :
  - Seule thread créée spontanément
    - Dans `start_kernel()`
  - Son rôle :
    - Lancer thread 1 (en utilisant `kernel_thread()`)
    - Exécuter `cpu_idle()` forever ...
      - Attente d'une interruption
      - Choisie par scheduler quand aucune autre thread dans l'état TASK_RUNNING
- Thread de pid 1 :
  - Exécute `init()`
    - Fin init noyau
  - Charge exécutable init par `execve()`
    - Devient processus normal

Contexte d'un Processus

- Descripteur : `struct task_struct` (<linux/sched.h>)



Processus Courant

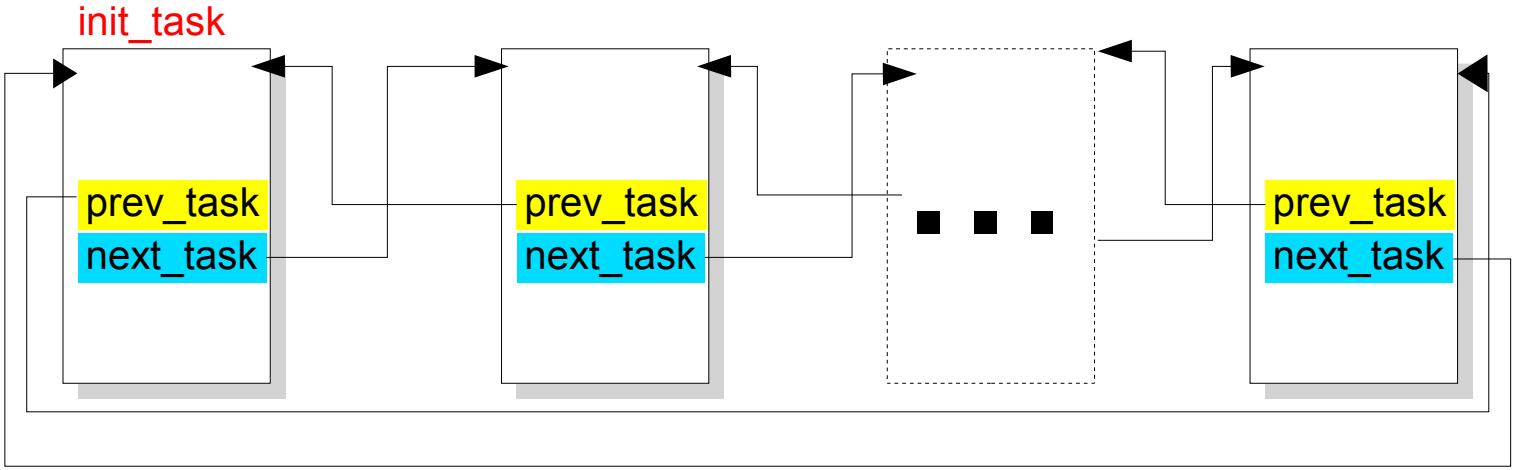
- (Pseudo-)variable **`current`** `(task_struct)`
  - Macro : masque 13 bits du pointeur de pile (reg `%esp`)
  - La structure se trouve toujours au début du segment de pile
    - Lors d'un changement de contexte
      - Echange du contenu des registres
      - Change automatiquement la valeur de **`current`**
- Tous les processus (lourds ET légers) ont un descripteur

Identification des Processus

- Identification non amibiguë
  - Addresse linéaire (32 bits) du descripteur
    - adresse **`struct task_struct`**
- Identification classique Unix : champ **`pid`**
  - cas général : **`getpid()`** <=> **`current->pid`**
  - cas particulier : threads type POSIX
    - Toutes les threads qui partagent le même contexte doivent répondre au même pid
    - Introduction d'un identifiant de groupe : **`tgid`**
      - **`pid`** du premier processus (thread) du groupe
      - Identique pour toutes les threads du groupe
      - **`getpid()`** retourne **`current->tgid`**

Liste Chaînée des Processus

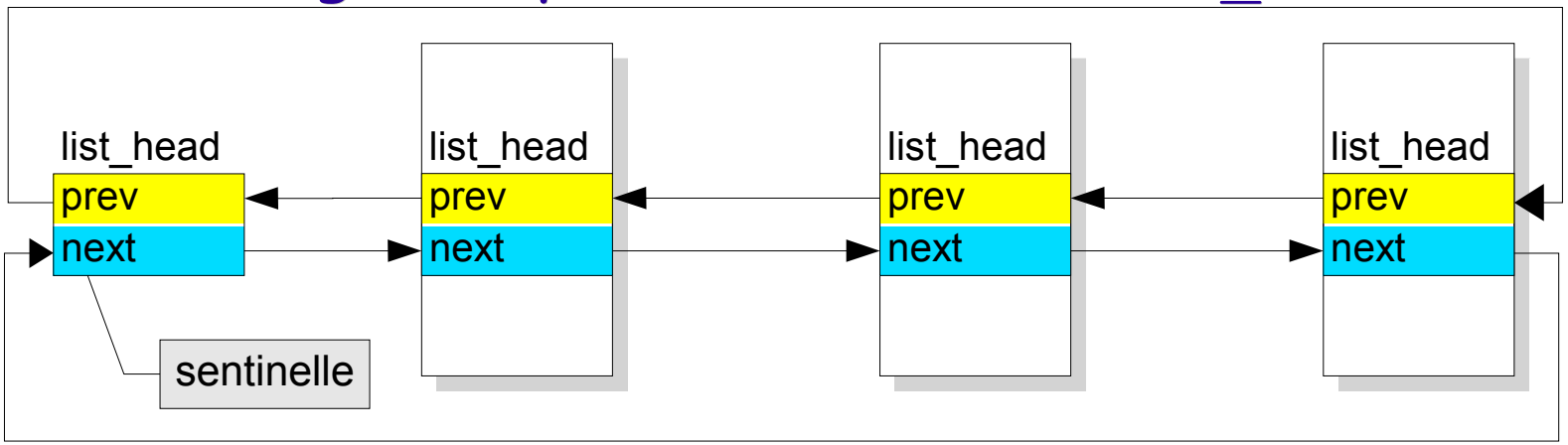
- Liste circulaire doublement chaînée
  - Pointeurs `prev_task` et `next_task` du descripteur
  - Tête de liste : `init_task` (processus init, pid=1)



  - Macros utiles
    - `SET_LINKS/REMOVE_LINKS`
    - `for_each_task(p)`
      - `for (p=&init_task ;(p=p->next_task)!=&init_task ;)`

A Propos de Listes Doublement Chaînées...

- Structure récurrente dans le noyau
  - Solution générique : la `struct list_head`



  - Attention : ne pointe pas directement au début de la structure chaînée
    - Macros classiques : list_add(new,prev), list_add_tail (new,head), list_del(entry), list_empty(head)
    - Récupération adr struct : list_entry(ptr,type,field)
    - Boucle : list_for_each(ptr,head)

Liste des Processus Prêts (TASK_RUNNING)

- Champ `run_list` du descripteur de processus
  - type `list_head`
- Macros spécifiques
  - `add_to_runqueue(&task_struct)`
    - Insertion en début de liste
  - `del_from_runqueue(&task_struct)`
  - `move_first_runqueue(&task_struct)`
  - `move_last_runqueue(&task_struct)`
  - `bool task_on_runqueue(&task_struct)`
  - `wake_up_process(&task struct)`
    - Place le processus dans l'état `TASK_RUNNING`
    - invoque `add_to_runqueue()`

Relations entre Processus

- Le descripteur du processus pointe vers plusieurs processus
  - `p_opptr` : parent original
    - Celui qui a créé le processus
  - `p_pptr` : parent courant
    - Différent si le processus est tracé
  - `p_cptr` : Dernier fils créé
  - `p_ysptr` : frère cadet suivant
    - Créé juste après par le même père
  - `p_osptr` : frère aîné précédent
    - Créé juste avant par le même père

Schéma des chaînages entre processus

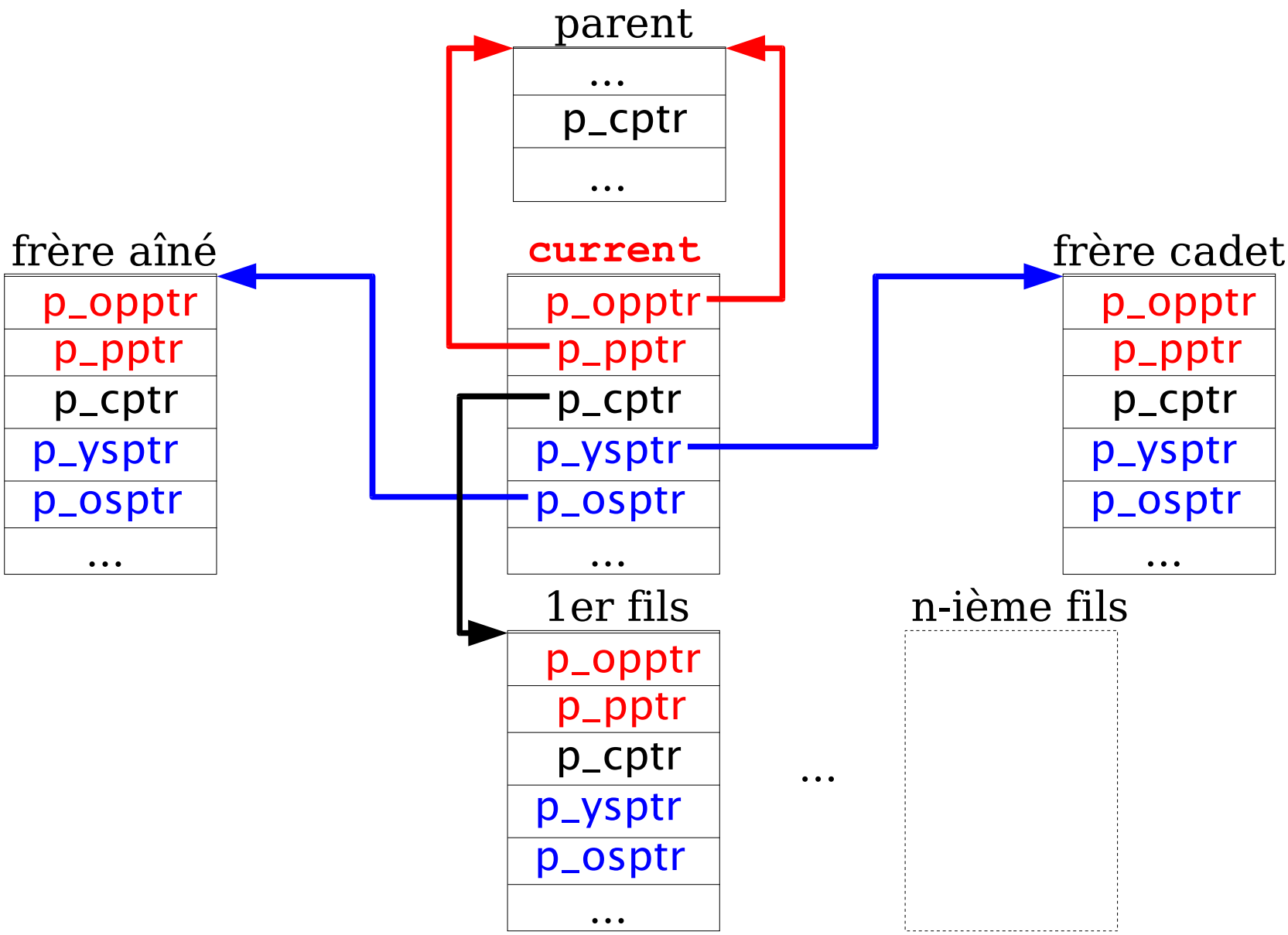


Schéma des chaînages inverses père/fils

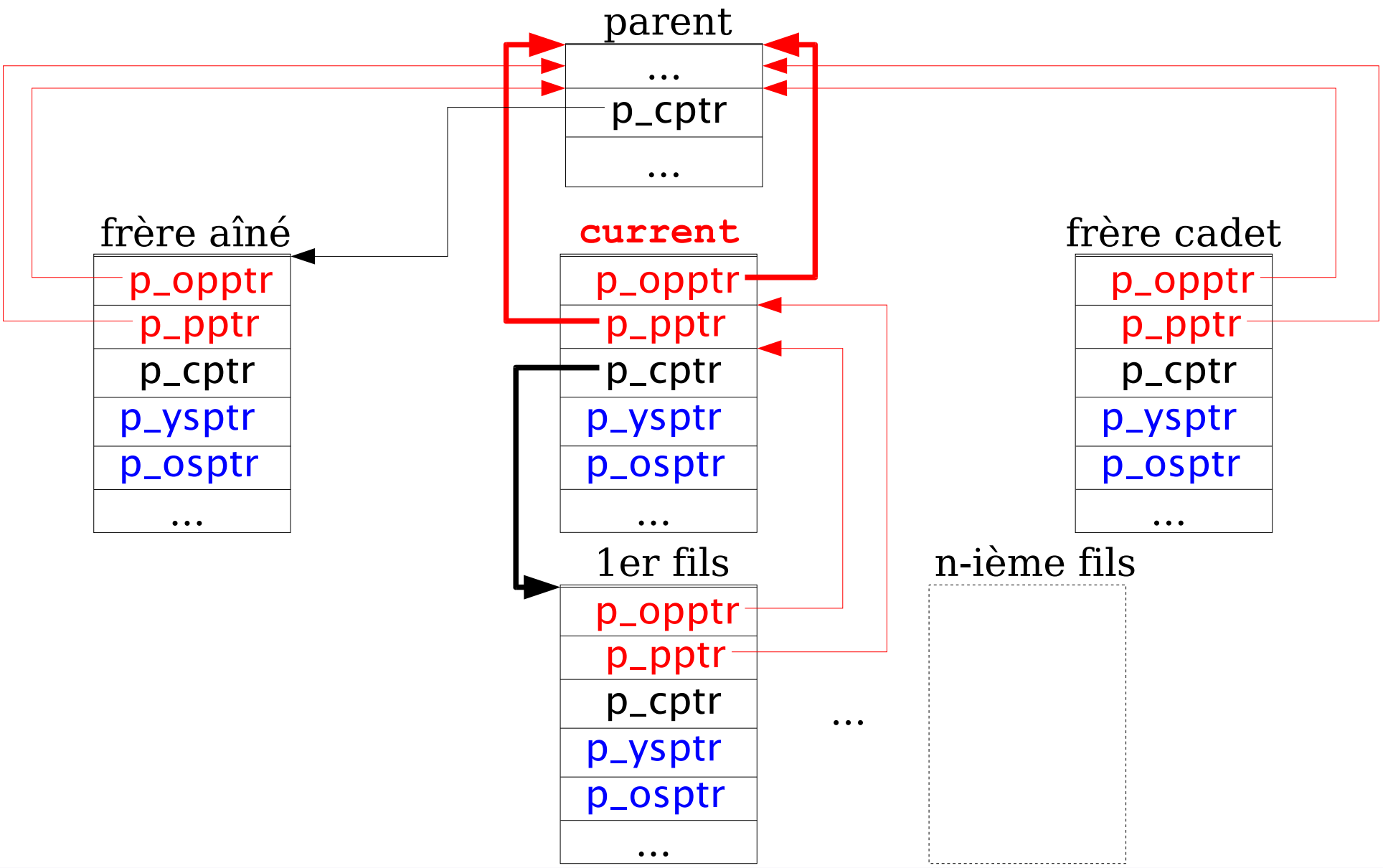
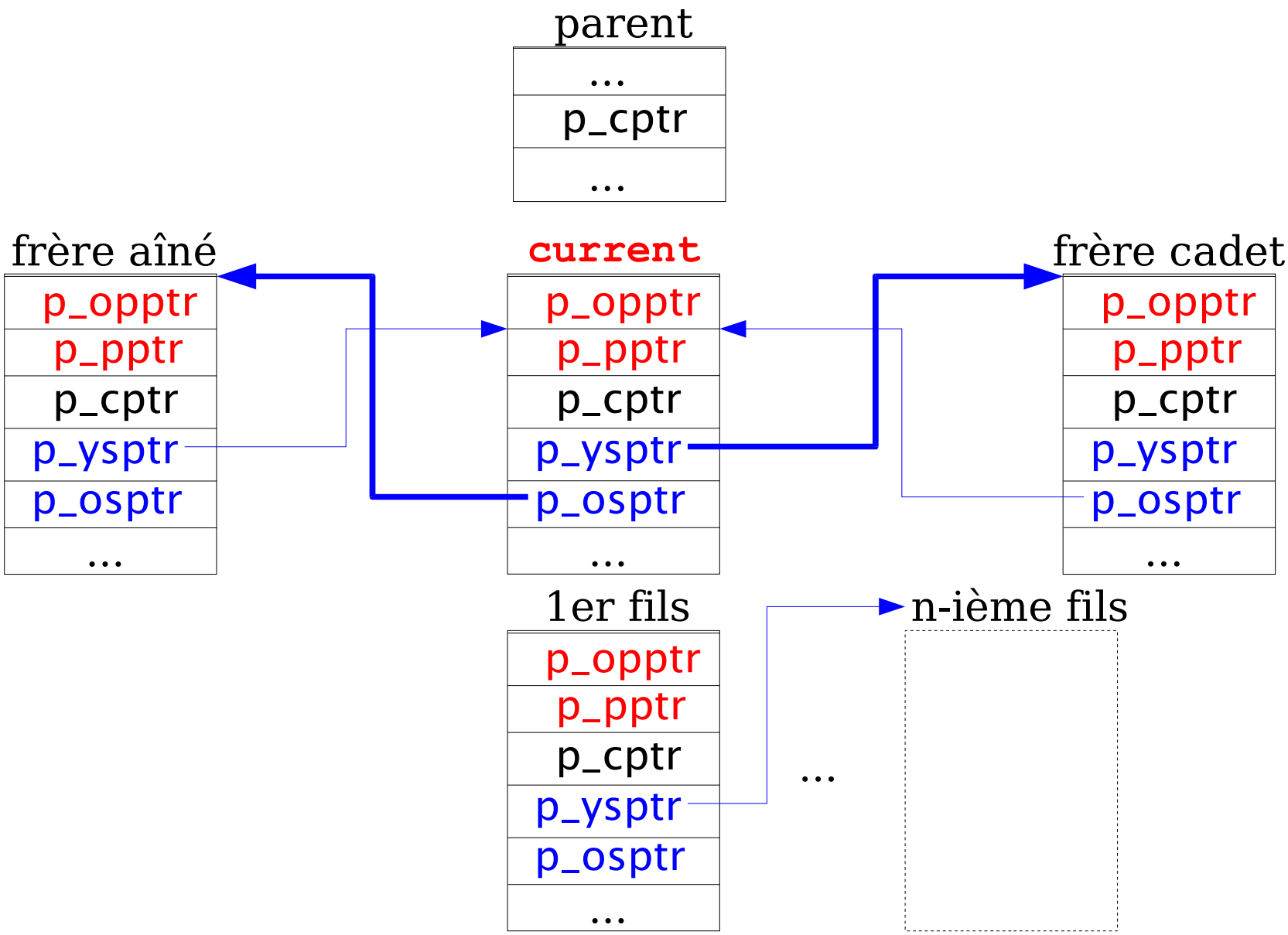


Schéma des chaînages inverses (frères)

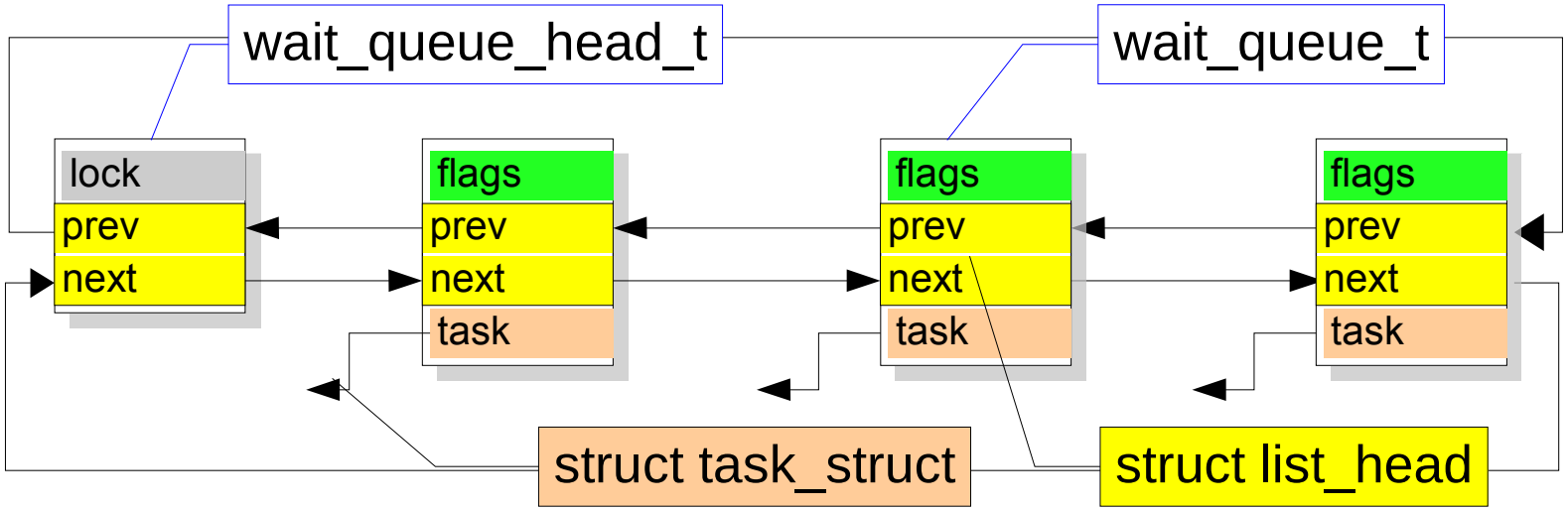


Organisation des Files de Processus

- Les processus peuvent se trouver dans différentes files en fonction de leur état
  - process list : tous les processus
    - champs `prev_task` et `next_task` de `task_struct`
  - `TASK_RUNNING` : champ run_list
  - `TASK_STOPPED` et `TASK_ZOMBIE` : pas d'autre chaînage
  - `TASK_INTERRUPTIBLE` et `TASK_UNINTERRUPTIBLE`
    - Files trop nombreuses pour apparaître directement dans la `struct task_struct`
    - Utilisation de wait queues
      - Correspondent chacune à l'attente d'un événement particulier

Wait Queues

- Structures basées sur la `struct list_head`



- flags = 1 : processus « exclusif »
  - Réveil = un seul à la fois
  - A défaut réveil des processus qui ont flags = 0
- flags = 0 : processus non « exclusif »
  - Tous les processus sont réveillés ensemble
    - Compétition possible ...

Utilisation des *Wait Queues*

- Déclaration
  - Statique : `DECLARE_WAIT_QUEUE_HEAD(wq_head)`
  - Dynamique : `init_waitqueue_head(wq_head)`
- Insertion
  - `add_wait_queue(wq_head,wq_elem)`
  - `add_wait_queue_exclusive(wq_head,wq_elem)`
- File vide ?
  - `waitqueue_active(wq_head)`
- Insertion du processus avec mise en sommeil
  - `[interruptible_]sleep_on[_timeout](wq_head[,to])`
  - `wait_event[_interruptible](wq_head,condition)`
- Extraction et réveil d'un processus
  - `wake_up[_interruptible][_sync][_nr|_all](wq_head[,nr])`

regarder priorité pour élection éventuelle