Chapitre 10

Processus et Linux

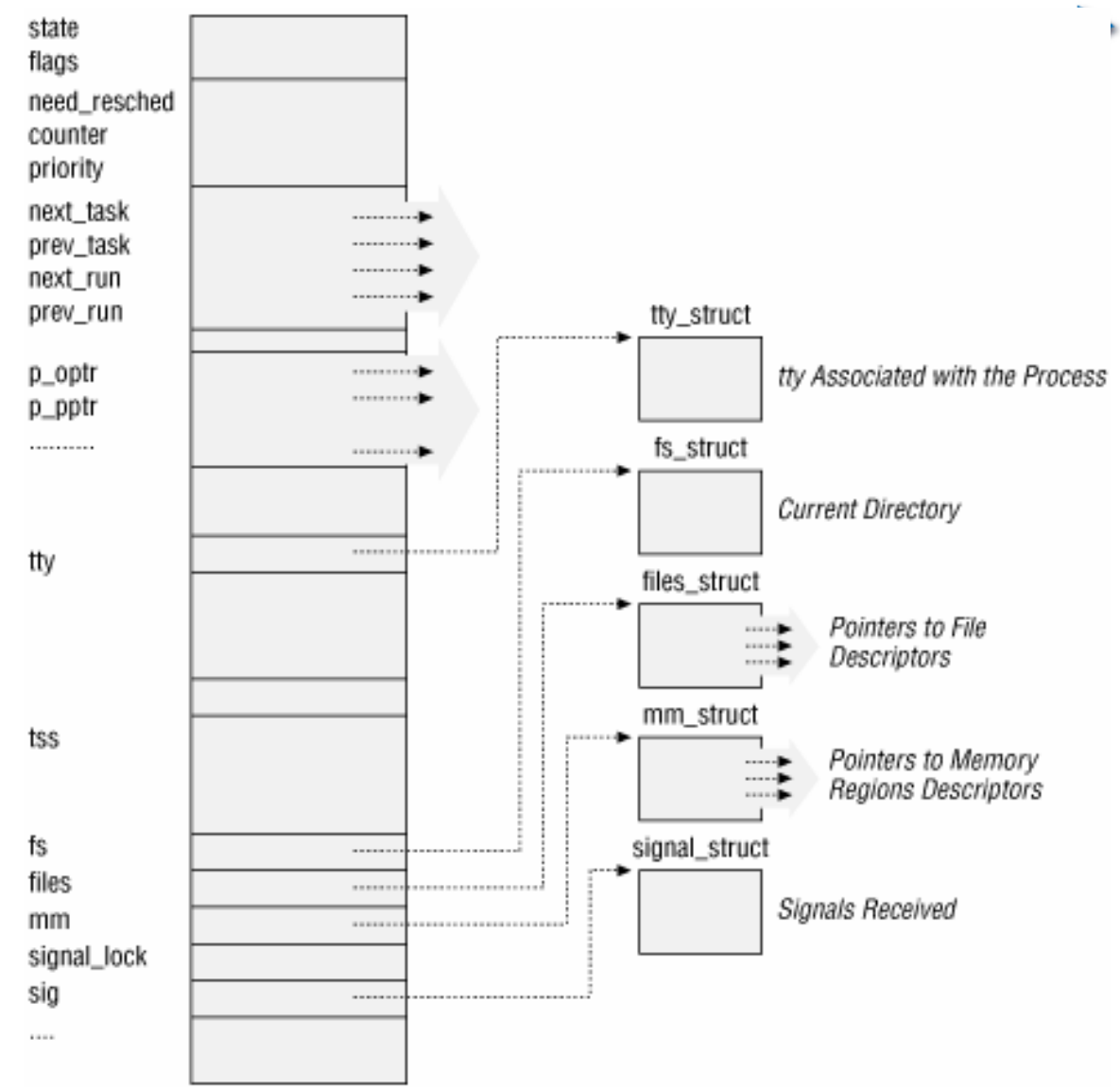
Définitions

- Un processus est une instance de programme en cours d'exécution
- Du point de vue kernel, le processus est une entité à qui sont allouées des ressources
  - CPU, mémoire…
- Quand un processus est crée, il reçoit une copie exacte de l'espace d'adressage de son parent
  - Il exécute le même code, commençant à la première instruction suivant l'appel système de création
- Notion de groupe de processus
  - Ex: « ls | grep … »
  - Permet de représenter la notion de *job*
- Pour des raisons de performance, introduction des *threads*
  - Fils d'exécutions partageant une grosse partie des données de l'application

Définitions

- Les premiers kernels Unix ne considéraient pas les *threads*
  - Une application multithread était vue comme un processus
  - Gestion des threads en User Space (*pthread*)
  - Nécessite d'écrire du code explicitement pour la synchronisation
- Linux utilise des processus légers
  - Processus partageant des ressources (espace d'adressage, fichiers ouverts…)
- Implémentation
  - Un processus léger est associé à un *thread*
  - Partage des ressources
  - Scheduling par le kernel
  - Native Posix Thread Library (NPTL)
    - 1 thread niveau utilisateur correspond à un processus léger kernel
    - Partie intégrante de la glibc
  - Next Generation Posix Threading Package (NGPT)
    - Abandonné en 2003

Process Descriptor



Le *process descriptor* permet au kernel de maintenir des informations sur un processus

États d'un processus

- Champs *state* du descripteur
- Tableau de flags
  - Mais mutuellement exclusifs
- TASK_RUNNING
  - Processus en cours d'exécution ou en attente du cpu
- TASK_INTERRUPTIBLE
  - Processus suspendu en attente d'une condition
  - Interruption hardware, libération d'une ressource système, signal… peuvent le réveiller
- TASK_UNINTERRUPTIBLE
  - Comme interruptible, sauf qu'un signal ne le réveillera pas
  - Utilisé par exemple pour accès hardware, pour ne pas être interrompu
- TASK_STOPPED
  - Le processus est arrêté
  - Réception de SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
- TASK_TRACED
  - Le processus a été arrêté par un debugger
  - Appel de *ptrace()*

États d'un processus

- 2 états supplémentaires sont aussi indiqués dans le champs *exit_state*
- EXIT_ZOMBIE
  - Le processus est terminé
  - Son parent n'a pas encore effectué de *wait4()* ou de *waitpid()*
  - Le kernel ne peut supprimer le processus car son parent pourrait avoir besoin d'informations
- EXIT_DEAD
  - Le processus est supprimé du système
  - Son parent a fait un *wait...*
  - Utilisé pour des raisons de synchronisation

Identification de processus

- Chaque processus (léger) est décrit par un *process descriptor*
- L'adresse de ce descripteur suffit pour les différencier
  - Le kernel utilise souvent des *process descriptor pointers*
- Mais on préfère manipuler des PID
  - Stocké dans le champs *pid* du *process descriptor*
- Les PIDs sont générés séquentiellement
  - Le PID d'un nouveau processus est celui du dernier crée +1
  - Valeur max : PID_MAX_DEFAULT -1 (32767)
    - Peut être changé dans */proc/sys/kernel/pid_max*
    - 4 194 303 en 64 bits
- Peut être nécessaire de recycler les PIDs
  - Utilisation d'une bitmap *pidmap_array*
  - 32k entrées en 32bits, tient donc sur un cadre
  - Sur 64 bits, plusieurs cadres nécessaires

Identification de processus

- Linux associe un PID à tous les processus (lourds et légers)
- Mais le programmeur s'attend à ce que les processus légers de la même application aient le même PID
- Linux utilise des *thread groups*
  - Ensemble des processus légers d'une même application
  - Le premier processus léger est le *thread group leader*
  - Les threads partagent le PID du leader
  - Stocké dans le champs *tgid*
  - Un appel à *getpid()* retourne le *tgid* et non le *pid*
- Obtenir le processus courant
  - Macro *current*

Listes de processus

- Les processus sont gérés par le kernel grâce à des listes doublement chaînées
- Liste de tous les processus
  - Accessible depuis *init_task task_struct*
    - *Process descriptor* du processus 0 (*swapper*)
  - *prev* pointe vers le dernier processus inséré
- Liste des *TASK_RUNNING*
  - Maintient d'une liste séparée pour les processus en attente du CPU
  - Évite de devoir scanner tous les processus pour trouver celui à exécuter
    - Mais il faut quand même scanner tous les candidats pour trouver le meilleur
    - Amélioration en 2.6
- Liste des TASK_*INTERRUPTIBLE
  - Utilisation des *wait queues*

Wait queues

- Les *wait queues* implémentent des attentes conditionnelles
- Un processus voulant attendre un événement se place dans la *wait queue* correspondante
- C'est donc un ensemble de processus qui seront réveillés par le kernel

```
struct __wait_queue_head {
   spinlock_t lock;
   struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;

struct __wait_queue {
  unsigned int flags;
  struct task_struct * task;
  wait_queue_func_t fun;
   struct list_head task_list;
}
typedef struct __wait_queue wait_queue_t;
```

Wait queues

- Chaque élément de la *wait queue* représente un processus en attente
  - L'adresse de son descripteur est dans le champs *task*
  - *task_list* relie les processus en attente du même évènement
- Réveiller tous les processus en attente n'est pas forcément une bonne idée
  - Plusieurs en attente d'une ressource exclusive…
  - *thundering herd problem*
- Introduction de 2 types de processus
  - Processus exclusifs : réveillés chacun leur tour par le kernel
  - Processus non exclusifs : tous réveillés pas le kernel
  - Champs *flags* (exclusif == 1)
  - Contrôle plus fin avec *wait_queue_func_t*

Utiliser les wait queues

- Création d'une wait queue
  - *DECLARE_WAIT_QUEUE_HEAD(name)*
    - Allocation statique
  - *init_waitqueue_head(wait_queue_head_t *q)*
    - Allocation dynamique
- Déclaration d'un élément
  - *init_waitqueue_entry(q,p)*
    - *q->flags=0;*
    - *q->task=p;*
    - *q->func = default_wake_function;*
  - *DEFINE_WAIT*
    - Initialise une *wait_queue_t* pour le processus actuellement en exécution sur le CPU
- Ajout dans une queue
  - *add_wait_queue* ou *add_wait_queue_exclusive*
- Suppression
  - *remove_wait_queue*

Utiliser les wait queues

- Un processus souhaitant attendre une condition peut utiliser une fonction
- *sleep_on(wait_queue_head_t *wq)*
  - Fonctionne sur le processus en cours
  - Met le processus en TASK_UNINTERRUPTIBLE
  - Le place dans la wait queue
  - Appelle le scheduler
- *interruptible_sleep_on(...)*
- *sleep_on_timeout(...)* et *interruptible_sleep_on_timeout(...)*
  - Le processus sera réveillé après une certaine durée
- *wait_event(wait queue, condition)*
  - Le processus est réveillé si la condition est remplie
  - *wait_event(&ma_wq, (truc == 1));*

Utiliser les wait queues

- Pour réveiller un processus, le kernel utilise les macros *wake_up, wake_up_nr, wake_up_all, wake_up_interruptible, wake_up_interruptible_nr, wake_up_interruptible_all, wake_up_interruptible_sync, wake_up_locked*
- Prennent en compte les processus en état TASK_INTERRUPTIBLE ou TASK_UNINTERRUPTIBLE
- Réveillent tous les processus non exclusifs qui ont l'état requis
- Les macros avec *nr* réveillent un nombre spécifié de processus exclusifs
- Les macros avec *all* réveillent tous les processus exclusifs
- Les autres réveillent 1 processus exclusif
- Les macros qui n'ont pas *sync*
  - Vérifient que la priorité des processus réveillé n'est pas supérieure à celle du processus courant
  - Appelle *schedule()* si nécessaire

Limites des ressources d'un processus

- Chaque processus a un ensemble de limitations sur les ressources
- Stockés dans *current->signal->rlim*
  - Tableau de *struct rlimit*
    - *unsigned long rlim_cur;*
    - *unsigned long rlim_max;*
- *RLIMIT_AS* : Taille maximale de l'espace d'adressage. Vérifiée lors d'un *malloc()*
- *RLIMIT_CORE* : Taille maximale du core dump
- *RLIMIT_CPU* : Temps CPU max en secondes. Reçoit un *SIGXCPU* si dépassement, puis *SIGKILL* si non terminaison
- *RLIMIT_DATA* : Taille maximale du tas
- *RLIMIT_FSIZE* : Taille maximale de fichier, SIGXFSZ si dépassement

Limites des ressources d'un processus

- *RLIMIT_LOCKS* : nombre maximal de lock sur fichiers
- *RLIMIT_MEMLOCK* : Taille maximale de mémoire non swappable
- *RLIMIT_MSGQUEUE* :
- *RLIMIT_NOFILE* : Max de file descriptors ouverts
- *RLIMIT_NPROC* : Max de processus que l'utilisateur peut posséder
- *RLIMIT_RSS* : Max de cadres que le processus peut posséder (non utilisé)
- *RLIMIT_SIGPENDING* : Nombre max de signaux en attente
- *RLIMIT_STACK* : Taille maximale de la pile

Limites des ressources d'un processus

- Obtenir la limite actuelle pour une ressource
  - *current->signal->rlim[RLIMIT_CPU].rlim_cur*
- On peut augmenter la limite sur une ressource jusqu'au maximum autorisé
  - *getrlimit()* et *setrlimit()*
- Seul l'administrateur peut augmenter la limite max
- Possibilité de mettre sans limite
  - *RLIM_INFINITY*
- Quand un utilisateur se log sur la machine
  - Le kernel démarre un processus superutilisateur
  - Ce processus fixe les limites
  - Et exécute ensuite un shell de login qui deviendra propriété de l'utilisateur
    - Il conserve les limites du parent

Process Switch

- Un changement de processus n'intervient que lors d'un appel à *schedule()*
- 2 étapes
  - Changer le *Page Table Directory* pour mettre en place le nouvel espace d'adressage
  - Changer la pile kernel et le contexte hardware qui fournit les informations pour redémarrer le processus
- La deuxième étape est effectuée par *switch_to*
  - Une des méthodes les plus dépendantes du hardware
  - 3 paramètres : *prev, next, last*
    - *prev* et *next* indiquent l'adresse des descripteurs de processus à *switcher*
    - *last* sert sauvegarder une référence vers le dernier processus schédulé

Process Switch

- Exemple d'utilisation de *last*
  - 3 processus : *A*, *B*, *C*
  - Switch de *A* vers *B*
    - *prev == A* et *next == B*
  - Plus tard, *C* est en exécution
  - Switch de *C* vers *A*
    - *prev == C* et *next == A*
    - Rétablissement du contexte de *A*
    - Dans ce contexte, on avait *prev == A* et *next == B*…
    - Le scheduler perd l'information sur *C*, sauf si elle est explicitement sauvegardée

Création de processus

- Un processus fils a un espace d'adressage différent du père
- Inefficace de le recopier
- Utilisation de Copy on Write
- Utilisation de processus légers qui partagent
  - Les tables de page, donc l'espace mémoire
  - Les fichiers ouverts…
- Appel système *vfork()*
  - Crée un processus qui partage l'espace mémoire de son parent
  - Pour éviter des problème d'accès concurrent, le père est bloqué jusqu'à terminaison du fils ou exécution d'un nouveau programme

Création de processus légers

- Les processus légers sont crées avec *clone()*
- Quelques paramètres
  - *fn* : Fonction à exécuter par le nouveau processus léger. Quand la fonction termine, le fils termine
  - *arg* : Pointeur vers arguments pour *fn*
  - *flags* : Informations diverses
- *Flags*
  - CLONE_VM : Partage toutes les tables de page et les descripteurs mémoire
  - CLONE_FS : Partage de la table qui identifie les répertoire racine et courant, et le bitmap des permissions initiales
  - CLONE_FILES : Partage de la table des fichiers ouverts
  - CLONE_SIGHAN : Partage de la table des handlers de signaux
  - CLONE_VFORK : c'est un appel a *vfork*
  - CLONE_STOPPED : force le fils à commencer dans l'état TASK_STOPPED
- Un appel à *fork()* est en fait un appel à *clone()* avec les flags qui vont bien
- Pareil pour *vfork()*

Threads kernel

- Des taches vitales sont déléguées à des processus
  - Vider les cache disque
  - Swapper certaines pages
- Pas forcément prioritaires, peuvent tourner en tache de fond
- Certaines de ces taches ne peuvent tourner qu'en mode kernel
  - Aucun intérêt à avoir des structures pour gérer User Mode
  - On utilise donc des *threads* spéciaux
- Création par *kernel_thread()*
  - Fonction et arguments à exécuter
  - Flags pour clone

Processus 0

- Ancêtre de tous les processus
- Appelé *idle process* ou *swapper*
- Crée durant la phase d'initialisation du kernel
- Exécuter *start_kernel()*
  - Fait plein de choses…
  - Et démarre un nouveau processus
    - *kernel_thread(init, NULL, CLONE_FS|CLONE_SIGHAND);*
- Nouveau processus (léger) de PID 1
  - Partage toutes les données avec son père
- Après avoir crée 1, le processus 0 appelle *cpu_idle()*
  - Appelle en boucle *hlt*
- Le scheduler ne choisit le processus 0 que si aucun processus n'est en *TASK_RUNNING*

Processus 1 et les autres

- Le processus 1 exécute *init()* qui finit l'initialisation du kernel
- Ensuite, utilise *execve()* pour exécuter le programme *init*
  - Devient un processus normal du système
- Autres *kernel threads*
  - *keventd* :
  - *kapmd* : gère les évenement de gestion d'énergie (APM)
  - *kswapd* : récupération périodique de mémoire
  - *pdflush* : écrit les buffers sur disque
  - *kblockd*
  - *ksoftirqd*

Destruction de processus

- Le kernel doit être notifié de la fin d'un processus pour libérer les ressources
- Appel explicite du programmeur à *exit()*
  - Libère les ressources allouées par la bibliothèque C
  - Appelle les méthodes spécifiées par le programmeur
  - Notifie le système
- Le compilateur ajoute toujours un appel à *exit()*
- Le kernel doit pouvoir détruire un ou plusieurs processus
  - Signal reçu ou exception CPU
- 2 méthodes en linux 2.6
  - *exit_group()* : termine un groupe complet de *threads*
  - *_exit()* : termine un unique thread