1 Gestion des processus

★ **Exercice 1: Descripteur de processus (DP)**
Linux 2.6 implémente les processus en utilisant une structure de données appelée *descripteur de processus* ou *task_struct* déclarée dans les sources au niveau du fichier *include/linux/sched.h*, lignes 1193-1531 (Listing 1).

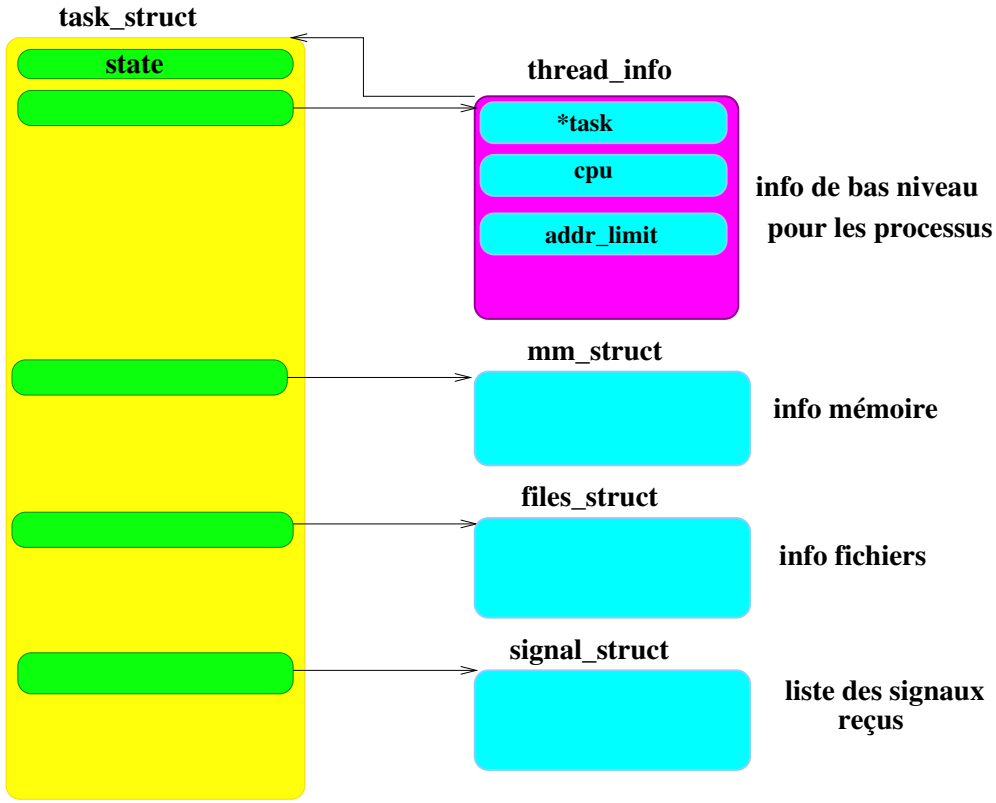

FIGURE 1 – Descripteur d'un processus

États d'un processus

▷ **Question 1:** Comment est représenté l'état d'un processus ? En déduire des exemples d'états que peut prendre un processus linux.

Identifiant de processus

▷ **Question 2:** À un processus est associé un identifiant (**pid**) défini au niveau de son descripteur. Sur combien de bits, est-il codé ? Quelle est la valeur maximale que peut prendre un pid en se basant sur le type de données utilisé ?

Remarques.

— Le système attribue les pid d'une façon séquentielle. À un moment donné, on affecte le dernier pid attribué plus 1 jusqu'à une valeur maximale qui correspond à `(PID_MAX_DEFAULT - 1)` Ensuite, le système réutilise le plus petit numéro de pid non utilisé. L'administrateur système peut changer (réduire) cette limite en utilisant `/proc/sys/kernel/pid_max`

— Afin d'être compatible avec les standards POSIX qui imposent que l'ensemble de threads appartenant à une application multithread doivent avoir le même pid, Linux utilise un identifiant *tgid* partagé par l'ensemble des threads et correspond au pid du leader du groupe. L'appel système *getpid()* renvoie la valeur de *tgid* du processus en cours.

★ Exercice 2: La thread_info en mémoire (x86)

Afin de limiter la quantité d'informations (du descripteur du processus) qui doit être chargée en mémoire en permanence, on a introduit la *thread_info*. Cette dernière est la seule qui doit être chargée en permanence en mémoire. Elle contient les informations de base sur le processus et contient un pointeur vers la *task_struct*. Linux stocke la *thread_info* (qui fait 52 octets) dans la même zone mémoire avec la pile **noyau** du processus pour des raisons d'efficacité.

▷ **Question 1:** Donner la taille en octets d'une page mémoire (cf listing 4).

▷ **Question 2:** Repérer la zone mémoire contenant la structure ***thread_info*** dans le listing 1. En déduire (cf listing 3) la taille de cette zone mémoire ?

▷ **Question 3:** Comment on peut retrouver l'adresse de la structure *thread_info* et l'adresse de son descripteur à partir du contenu du registre de pile ?

2 Gestion de la mémoire

Etant donné qu'une partie du gestionnaire de la mémoire est dépendante du matériel, on se placera dans le contexte de l'architecture x86.

★ **Exercice 3: Espace d'adressage**

L'espace d'adressage d'un processus est découpé en 2 parties :

l'espace utilisateur [0-PAGE_OFFSET[réservé au code et données du programme et des bibliothèques patagées, ainsi qu'au tas et à la pile. Cette partie diffère d'un processus à l'autre.

l'espace noyau ($\geq$ PAGE_OFFSET) réservé au noyau (code et données) et est identique pour tous les processus.

▷ **Question 1:** Combien vaut PAGE_OFFSET (listing 4) ? En déduire la taille de l'espace d'adressage utilisateur.

▷ **Question 2:** Repérer dans le listing 1, le champ qui définit l'espace d'adressage d'un processus au niveau de son descripteur. Interpréter les champs de la structure associée (cf listing 5).

L'espace d'addressage d'un processus est divisé en un certain nombre de régions. Les régions ne se chevauchent pas et englobent des adresses ayant les mêmes protection et usage. Par exemple "une librairie partagée en lecture seule" ou "le tas du processus". Une liste complète des régions mappées d'un processus peut être obtenue via ***/proc/[pid]/maps***.

▷ **Question 3:** Interpréter le contenu de la figure 2

```
$ cat /proc/self/maps
08048000-0804c000 r-xp 00000000 03:07 293186     /bin/cat
0804c000-0804d000 rw-p 00003000 03:07 293186     /bin/cat
0804d000-0806e000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 03:07 390929     /lib/ld-2.3.2.so
40016000-40017000 rw-p 00015000 03:07 390929     /lib/ld-2.3.2.so
40017000-40018000 rw-p 00000000 00:00 0
40027000-4014f000 r-xp 00000000 03:07 390935     /lib/libc-2.3.2.so
4014f000-40157000 rw-p 00127000 03:07 390935     /lib/libc-2.3.2.so
40157000-4015a000 rw-p 00000000 00:00 0
4015a000-401a9000 r--p 00000000 03:08 734469     /usr/lib/locale/locale-archive
bfffe000-c0000000 rwxp fffff000 00:00 0
```

FIGURE 2 – Les régions mémoire

★ **Exercice 4: Gestion de la table des pages**

Linux utilise une pagination à trois niveaux généralisée à toutes les architectures. Chaque processus (espace d'adressage) dispose d'un répertoire de pages global (PGD : Page Global Directory), dont l'adresse est donnée par le registre cr3 mis à jour à chaque changement de contexte. Chaque entrée de la PGD correspond à un répertoire de pages (PMD : Page Middle Directory) dans lequel chaque entrée pointe sur une table de pages (PTE : Page Table Entries). Chaque entrée de cette table donne l'adresse de la page physique correpondante.

▷ **Question 1:** Soit l'adresse virtuelle `0x84456cd9`. Faire un schéma qui montre comment cette adresse est traduite.

▷ **Question 2:** Le x86 utilise juste deux niveaux de pagination. D'après vous comment linux se débrouille avec ça ? Comment est structurée une adresse virtuelle ? Quel est le nombre d'entrées dans les différentes tables de pagination ? Cf listing 6.

▷ **Question 3:** Donner un schéma de traduction de l'adresse 0x84456cd9 dans le contexte spécifique du x86 ?

▷ **Question 4:** Interpréter les 2 lignes du fichier ***arch/x86/include/asm/pgtable_32_types.h*** (listing 7).

Descripteur de page Le descripteur de page correspond à une entrée de la table PTE et défini par la structure `pte_t`. Les entrées des autres tables sont également définies similairement par `pmd_t` et `pgd_t`.

▷ **Question 5:** pte_t est déclarée comme un simple entier de 32 bits. Comment d'après vous est divisée cette structure ?

Listing 1 – include/linux/sched.h

```
182  #define TASK_RUNNING            0
183  #define TASK_INTERRUPTIBLE      1

188  #define EXIT_ZOMBIE             16

1193 struct task_struct {
1194         volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
1195         void *stack;

1284         struct mm_struct *mm;

1308         pid_t pid; // typedef __kernel_pid_t          pid_t;
                        // typedef int             __kernel_pid_t;
1309         pid_t tgid;

1531 };

2058 union thread_union {
2059         struct thread_info thread_info;
2060         unsigned long stack[THREAD_SIZE/sizeof(long)];
2061 };
```

Listing 2 – arch/x86/include/asm/thread_info.h

```
26 struct thread_info {
27         struct task_struct      *task;          /* main task structure */

44 };

196 /* how to get the thread information struct from ASM */
197 #define GET_THREAD_INFO(reg)     \
198         movl $-THREAD_SIZE, reg; \
199         andl %esp, reg
```

Listing 3 – arch/x86/include/asm/page_32_types.h

```
18 #define THREAD_ORDER    1
19 #define THREAD_SIZE     (PAGE_SIZE << THREAD_ORDER)
```

Listing 4 – arch/x86/include/asm/page_types.h

```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (1UL << PAGE_SHIFT)
#define PAGE_MASK       (~(PAGE_SIZE-1))

#define __PAGE_OFFSET     (0xC0000000)
#define PAGE_OFFSET       ((unsigned long)__PAGE_OFFSET)
```

Listing 5 – include/linux/mm_types.h

```
130 struct vm_area_struct {
131         struct mm_struct * vm_mm;     /* The address space we belong to. */
132         unsigned long vm_start;       /* Our start address within vm_mm. */
133         unsigned long vm_end;         /* The first byte after our end address
134                                          within vm_mm. */

139         pgprot_t vm_page_prot;          /* Access permissions of this VMA. */

186 };


222 struct mm_struct {
223         struct vm_area_struct * mmap;           /* list of VMAs */


236         pgd_t * pgd;

253         unsigned long stack_vm;
254         unsigned long start_code, end_code, start_data, end_data;
255         unsigned long start_brk, brk, start_stack;

318 };
```

Listing 6 – arch/x86/include/asm/pgtable-2level_types.h

```
26 #define PGDIR_SHIFT     22
27 #define PTRS_PER_PGD    1024

30 /*
31  * the i386 is two-level, so we don't really have any
32  * PMD directory physically.
33  */

35 #define PTRS_PER_PTE    1024
```

Listing 7 – arch/x86/include/asm/pgtable_32_types.h

```
17 #define PGDIR_SIZE      (1UL << PGDIR_SHIFT)
18 #define PGDIR_MASK      (~(PGDIR_SIZE - 1))
```