Design-ul Sistemului de Operare *Pintos*

Butiri Dan, Lazar Laurentiu, Pârvu Leonard

November 8, 2015

Chapter 1

Prezentare generală

1.1 Echipa

1. Butiri Dan
   (a) Threads: Priority Scheduler
2. Lazar Laurentiu
   (a) Threads: Priority Scheduler
3. Pârvu Leonard
   (a) Threads: Alarm Clock
   (b) Threads: Multilevel Feedback Queue Scheduler

Chapter 2

Design-ul Modulului *Threads*

2.1 Alarm Clock

Aceasta prima parte a proiectului consta in reimplementarea functiei *timer_sleep()* din *"devices/timer.c"*. Implementarea initiala presupune ocuparea procesorului atunci cand se apeleaza aceasta functie pentru un thread, el fiind pastrat in *ready_list*.

2.1.1 Structuri de date

Pentru a evita acest busy waiting am ales sa adaugam structurii thread din *"threads/thread.h"* un field *wakeup_time* pentru a memora timpul cand trebuie sa se trezeasca threadul respectiv.

De asemenea, in *"devices/timer.c"*, se va adauga o noua lista, *sleeping_list*, ce contine toate threaduri pentru care a fost apelata functia *timer_sleep()*.

Astfel atunci cand pentru un thread se apeleaza functia *timer_sleep()*, acesta va fi mutat in *sleeping_list* in loc sa fie pastrat in *ready_list*, situatie in care tine procesorul ocupat.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
```

```
    char name[16];                      /* Name (for debugging purposes).
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads l

                int64_t wakeup_time;

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

2.1.2 Algoritmi

Atunci cand se va apela functia *timer_sleep()*, se va lua threadul curent, se va calcula si memora timpul la care trebuie sa se trezeasca, timp_curent + sleep_time, se va insera in lista *sleeping_list* si se va scoate din lista *ready_list*.

La fiecare apel al functie *timer_interrupt()* se va parcurge lista *sleeping_list* pentru a vedea daca exista un thread ce trebuie trezit si introdus in lista de threaduri ready_list.

Pentru a minimiza timpul petrecut pentru a parcurge lista sleeping_list, in cadrul metodei *timer_interrupt()* se vor pastra threadurile sortate crescator dupa timpul la care trebuie sa se trezeasca. Astfel, nu trebuie parcursa intreaga lista. Se va parcurge lista doar pana cand intalnim un thread care trebuie trezit, dupa acela sigur nefiind alt thread ce trebuie trezit. In cazul in care nu trebuie trezit niciun thread, lista se va parcurge in totalitate.

2.1.3 Sincronizare

Pentru a evita conditiile de cursa atunci cand mai multe threaduri apeleaza *timer_sleep()*, se vor dezactiva intreruperile temporar, pana la terminarea functiei.

Astfel, nu pot aparea intreruperi atunci cand se efectueaza un apel la functia *timer_sleep()*. Nu exista conditii de cursa.

2.2 Priority Scheduler

2.2.1 Structuri de date

Deoarece prioritatea unui thread se poate schimba, am ales sa adaugam un nou camp intreg *initial_priority*, in structura thread, pentru a retine prioritatea unui thread inainte de a primii donatii.

Tot in structura thread, am adaugat un pointer catre o structura de tip lock, in care se va retine lacatul dupa care asteapta threadul curent. Acest lacat, avand un *lock holder*, mai ajuta thread-ul curent si prin faptul ca retine thread-ul care a primit donatie de prioritate de la thread-ul curent (un thread poate, la un moment dat, sa astepte dupa un singur lacat, si deci, sa ofere donatie directa unui singur thread).

Deasemenea, un thread a mai primit si o lista de lacate pe care le-a obtinut, aceasta lista fiind necesara la recalcularea prioritatilor dupa eliberarea unui lacat.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes).
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads l
```

```
    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    /* Added for Priority Scheduler */
    int initial_priority;               /* Thread's initial priority,
                                           not altered by donations. */
    struct list acquired_locks;
    struct lock* waiting_on_lock;       /* The lock for which the thread
                                           was blocked. */

  #ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
  #endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

Strucutra de lacat s-a schimbat prin faptul ca lacatele sunt implementate fara ajutorul semafoarelor, deci campurile dintr-un semafor apar acum si in structura de lacat. Deasemenea, in structura de lacat a fost nevoie de un element pentru liste, cu ajutorul caruia un lacat este adaugat in lista de *acquired_locks* a unui thread.

```
struct lock
  {
    struct thread *holder; /* Thread holding lock (for debugging). */

    /* Added for Priority Scheduler */
    bool is_open;           /* If is_open == true, the
                               lock can be acquired. */
    struct list waiters;

    struct list_elem elem; /* List element for a thread
                               acquired_locks list */
```

```
};
```

2.2.2 Algoritmi

Pentru a planifica intotdeauna thead-ul cu prioritatea cea mai mare, thread-urile sunt pastrate in *ready_list* in ordinea prioritatilor. La inserarea unui thread cu o prioritatea care mai exista in *ready_list*, acesta va fi pus in spatele tuturor thread-urilor cu care este egal ca prioritate. Intotdeauna va fi planificat primul thread din *ready_list*.

Preemptivitatea este asigurata prin faptul ca, de fiecare data cand apare un nou thread in sistem sau de fiecare data cand prioritatea unui thread este modificata, se verifica daca exista in *ready_list* un thread cu prioritate mai mare decat cea a thread-ului curent. Daca da, atunci thread-ul curent este suspendat si se apeleaza planificatorul.

Pentru problemele de sincronizare cu lacate, algoritmul este impartit in doua locuri, in functiile *lock_acquire* si *lock_release*.

La apelul functiei *lock_acquire*, in cazul in care lacatul este ocupat, thread-ul curent va fi blocat - acesta este scos din *ready_list* si adaugat in lista de asteptare a lacatului respectiv, lista fiind sortata dupa prioritatea thread-urilor care asteapta. Deasemenea, in cazul in care prioritatea thread-ului blocat este mai mare ca cea a detinatorului lacatului, este necesara o donatie de prioritate. Aceasta donatie poate fi clasificata in doua cateogrii:

- Detinatorul lacatului este READY - se face o donatie de la thread-ul curent la detinatorul lacatului si detinatorul lacatului este repozitionat in *ready_list*.

- Detinatorul lacatului este BLOCKED si a facut o donatie de prioritate - este necesara o propagare a donatiilor. Se face donatie de la thread-ul curent la detinatorul lacatului si detinatorul lacatului este repozitionat in lista de asteptare a lacatului care l-a blocat. Apoi se efectueaza propagarea donatiei de prioritate.

La apelul functiei *lock_release*, daca thread-ul curent a primit donatie de prioritate, este necesara recalcularea prioritatii sale. In cazul in care niciun

thread nu mai asteapta dupa thread-ul curent, acesta revine la prioritatea sa initiala. Altfel, noua sa prioritate va fi maximul dintre prioritatile thread-urilor care asteapta dupa thread-ul curent.

2.3 Multilevel Feedback Queue Scheduler

Acest planificator decide ce threaduri vor rula modificand dinamic prioritatile acestora pe baza a doua caracteristici ale lor: cat de dragute sunt, *niceness*, si cat de consumatoare de procesor sunt, *recent_ cpu*.

2.3.1 Structuri de date

Pentru a realiza acest lucru am adaugat in structura de thread din *"threads/thread.h"* doua campuri care sa inglobeze cele doua caracteristici pe baza carora se recalculeaza prioritatile.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes).
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads l

                int64_t wakeup_time;
                int niceness;
                int recent_cpu;

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
```

```
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

De asemenea se va adauga o variabila globala, *loag_ avg*.

2.3.2 Algoritmi

Presupunem ca threadurile A, B si C au niceness 0, 1 si 2. Fiecare au o valoare de *recent_ cpu* de 0. Se completeze tabelul de mai jos aratand decizile de planificare, prioritatile si valorile de *recent_ cpu* pentru fiecare thread pentru numarul de tickuri.

timer ticks	recent_cpu A B C	priority A B C	thread to run
	A B C	A B C	
0	0 0 0	63 61 59	A
4	4 0 0	62 61 59	A
8	8 0 0	61 61 59	A
12	12 0 0	60 61 59	B
16	12 4 0	60 60 59	A
20	16 4 0	59 60 59	B
24	16 8 0	59 59 59	A
28	20 8 0	58 59 59	B
32	20 12 0	58 58 59	C
36	20 12 4	58 58 58	A

Performanta acestui planificator o sa fie afectata deoarece de fiecare data cand trebuie sa se faca schedule trebuie sa se recalculeze prioritatile. Acest lucru conduce la introducerea unor intarzieri.

Formulele pentru calcularea valorilor pentru *priority*, *niceness* si *ready_ threads* presupun numere intregi, insa pentru calcularea valorilor *recent_ cpu* si *load_ avg*

trebuie folosite numere reale. Pintosul nu pune la dispozitie suport pentru aritmetica in virgula mobila. Astfel calculele efectuate cu numere reale trebuie simulate utilizand numere intregi.

Astfel se va crea un header numit *fixed_point_arithmetic.h* care va contine macrourile necesare executarii operatiilor cu numere reale.

Fie x si y numere in fix point arithmetic, n un numar intreg, p + q = 31 si f fiind 1 « q. Astfel se vor implementa macrouri pentru urmatoarele operatii:

- conversia lui n la fix point: n * f
- conversia lui x la intreg (rotunjire spre zero): x / f
- conversia lui x la intreg (rotunjire spre cel mai apropiat) : (x + f / 2) / f daca x>=0 sau (x - f / 2) / f daca x<=0
- suma dintre x si y: x + y
- diferenta dintre x si y: x - y
- suma dintre x si n: x + n * f
- diferenta dintre x si n: x - n * f
- inmultirea dintre x si y: ((int64_t)x) * y / f
- inmultirea dintre x si n: x * n
- impartirea lui x la y: ((int64_t)x) * f / y
- imparitrea lui x la n: x / n

De asemenea trebuie ignorate apelurile la functia *set_priority* si implementate functiile *thread_set_nice*, *thread_get_nice*, *get_load_avg*, *get_recent_cpu*, *increase_recent_cpu*, *calc_recent_cpu*.