به نام خدا

دانشگاه شهید بهشتی - رشته مهندسی کامپیوتر

درس آزمایشگاه سیستم عامل
استاد آهوز

دستور کار شماره ۲ :

آشنایی با مفاهیم هم‌زمانی، زمانبندی و مدیریت نخ ها در سطح کرنل

اعضای گروه :

متین ارجمند **امیرحسین جعفرزاده**
محدثه صفاری **علی نصرالله پور**

فهرست :

مقدمه :

سیستم عامل Pintos یک سیستم عامل سبک و آموزشی است که برای تحلیل و یادگیری اصول طراحی سیستم های عامل به خصوص بخش مدیریت نخ ها، همگام سازی و زمانبندی طراحی شده است. در نسخه اولیه Pintos ، سیستم مدیریت نخ ها بسیار ساده و ابتدایی پیاده سازی شده بود و همین موضوع سبب بروز مشکلاتی در کارایی و عدالت در اجرای نخ ها میشد.
هدف پروژه اول این بود که توانایی های سیستم نخ ها افزایش یابد و سیستم به رفتار سیستم های واقعی نزدیکتر شود.

تمرکز پروژه روی دو بخش اساسی است :

۱. پیاده سازی خواب واقعی نخ ها بدون استفاده از Busy Waiting
۲. پیاده سازی زمانبندی بر اساس اولویت همراه با Donation Priority برای رفع مشکل Priority Inversion

مشکلات و چالش های سیستم اولیه :

در نسخه خام Pintos ، تابع timer_sleep() از روش busy waiting استفاده می‌کرد.
در این روش، نخ هر بار بررسی می‌کرد که آیا مدت خواب آن تمام شده است یا نه و در طول این زمان دائما CPU را اشغال میکرد. این رفتار به بررسی می هیچ عنوان با مفهوم sleep واقعی سازگار نبود.

در نتیجه :

- نخ ها حتی هنگامی که باید در حالت خواب باشند، CPUرا اشغال میکردند.
- مقدار idle_ticks سیستم عملا صفر بود.
- سیستم رفتار نادرستی در تست های زمان بندی داشت.

مشکل بعدی نبود زمانبندی مبتنی بر اولویت بود. در نسخه اولیه، تمام نخ ها با روش Round Robinاجرا می‌شدند و هیچ اهمیتی به اولویت نخ ها داده نمی‌شد. طبیعی است که در سیستم هایی مانند سیستم عامل، عملیات حیاتی باید زودتر از عملیات کم اهمیت اجرا شوند. اما این امکان در Pintos در ابتدا وجود نداشت.

سومین مشکل مربوط به Priority Inversion بود. این مشکل معمولا هنگام استفاده از Lock ها رخ می‌دهد. اگر نخ با اولویت پایین قفلی را گرفته باشد و نخ با اولویت بالا نیز نیازمند همان قفل باشد، نخ با اولویت بالا انتظار می‌کشد، حتی اگر از نظر منطقی باید زودتر اجرا شود. اگر در همین حین نخ دیگری با اولویت متوسط شروع به اجرا کند، اجرای نخ با اولویت بالا متوقف میشود و سیستم دچار پدیده‌ای می‌شود که به Priority Inversion معروف است. این مشکل در سیستم واقعی میتواند باعث توقف های جدی و غیر قابل قبول شود.

فاز ۳ پروژه (Alarm Clock) :

پیاده سازی جدید اصلاح تابع timer_sleep() منجر به بهینه سازی مصرف CPU شده است. در رویکرد قبلی که از Busy Waiting استفاده می‌شد، نخ‌های در حال خواب به صورت فعالانه و پی در پی وضعیت زمان را بررسی می‌کردند و با فراخوانی thread_yield()خود را به صف آماده ها اضافه می کردند، که این فرآیند باعث هدررفت قابل توجه منابع پردازشی می‌شد.

تغییرات در timer.c – تابع timer_sleep() :

در نسخه اصلی، timer_sleep() :

- زمان شروع را می‌گرفت (start = timer_ticks())
- تا وقتی timer_elapsed(start) < ticks بود در یک حلقه while فقط thread_yield() می‌کرد.
- یعنی busy waiting : نخ بیدار می‌ماند و فقط مدام CPU را به بقیه می‌داد.

در نسخه جدید اگر ticks > 0 باشد :

- وقفه ها را غیرفعال می‌کند.
- روی نخ فعلی یک فیلد جدید blocked_ticks را برابر تعداد تیک‌هایی که باید بخوابد قرار می‌دهد.
- نخ را با thread_block() واقعا بلاک می‌کند.
- سپس وقفه ها را برمی‌گرداند.

```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();
  ASSERT (intr_get_level () == INTR_ON);

  /* ++ Use block to handle sleep */
  if (ticks > 0) {
    enum intr_level old_intr_level = intr_disable();
    struct thread *t = thread_current();
    t->blocked_ticks = ticks;
    thread_block();
    intr_set_level(old_intr_level);
  }
  /* -- Old Implementation */
  // while (timer_elapsed (start) < ticks)
  //   thread_yield ();
}
```

تغییرات timer_interrupt() :

در هر وقفه تایمر، بعد از thread_tick() روی همه نخ‌ها thread_check_blocked را صدا می‌زند تا نخ‌های خواب را مدیریت کند.

```
thread_tick ();

 /* ++ Check block status for each thread */
 thread_foreach(thread_check_blocked, NULL);
```

تغییرات در thread.c – مدیریت blocked-ticks :

۱.مقداردهی اولیه‌ی blocked_ticks در thread_create :

```
/* Initialize thread. */
 init_thread (t, name, priority);
 tid = t->tid = allocate_tid ();

 /* ++ Set blocked ticks to 0 */
 t->blocked_ticks = 0;
```

یعنی هر نخ جدید با مقدار blocked_ticks = 0 شروع می‌شود.

۲.تابع جدید thread_check_blocked :

```
/* ++ Check if the thread should be unblocked as it's
    blocked ticks reached to 0. */
void thread_check_blocked (struct thread *t, void *aux UNUSED) {
  if (t->status == THREAD_BLOCKED && t->blocked_ticks > 0) {
    t->blocked_ticks--;
    if (t->blocked_ticks == 0) {
      thread_unblock(t);
    }
  }
}
```

اگر نخ در وضعیت THREAD_BLOCKED است و blocked_ticks > 0 :

- یک تیک از آن کم می‌کند.
- وقتی به صفر رسید، با thread_unblock نخ را به صف آماده‌ها برمی‌گرداند.

تغییرات در struct thread (در thread.h) :

فیلد اضافه شده :

```
uint64_t blocked_ticks;   /* ++ Blocked ticks. */
```

جمع بندی فاز ۳ :

هدف :حذف busy waiting در timer_sleepو استفاده از مکانیزم بلوک کردن نخ‌ها.

پیاده سازی:

- به هر نخ یک شمارنده‌ی blocked_ticks داده شد.
- در timer_sleep به جای حلقه‌ی while نخ را بلاک می‌کند.
- در timer_interrupt روی همه نخ‌ها می‌چرخد و هر نخ بلاک شده را هر تیک یک واحد کم می‌کند و در زمان صفر آن را unblock می‌کند.

فاز ۴ پروژه (Priority Donation و Priority Scheduler) :

زمان‌بندی سادهٔ FIFO باعث می‌شود نخ‌های با اولویت بالا منتظر بمانند و مساله *priority inversion* رخ دهد.

هدف فاز: تبدیل scheduler به اولویت‌محور و پیاده‌سازی مکانیزم donation تا inversion رفع شود.

۱.ساختار داده‌ها - struct thread و struct lock

فیلد های اضافه شده در thread.h :

```
int base_priority;          /* Base priority. */
struct list locks;            /* Locks that the thread is holding. */
struct lock *lock_waiting;  /* The lock that the thread is waiting for. */
```

فیلد های اضافه شده در synch.h برای struct lock :

```
int max_priority;
struct list_elem elem;
```

هدف:

- base_priority : اولویت اصلی نخ قبل از هرdonation

- locks : لیست قفل‌هایی که نخ نگه داشته تا بتوانیم donation را از روی آن‌ها حساب کنیم.
- lock_waiting : قفلی که نخ فعلی در صف آن منتظر است، برای پیاده‌سازی donation چند سطحی.
- max_priority داخل lock : بیشترین اولویت تردهایی که منتظر این lock هستند، برای این‌که holder بداند تا چه سطحی باید donation بگیرد.

۲.زمان بند – صف آماده ها و تابع thread_set_priority :

الف) آماده ها به صورت مرتب بر اساس اولویت

در نسخه اصلی ready_list به صورت FIFO با list_push_back مدیریت می‌شد.

در نسخه تغییر یافته :

```
/* thread_unblock: */
list_insert_ordered(&ready_list, &t->elem,
                    (list_less_func *)&compare_priority, NULL);

/* thread_yield: */
if (cur != idle_thread)
  list_insert_ordered(&ready_list, &cur->elem,
                      (list_less_func *)&compare_priority, NULL);
```

و تابع مقایسه :

```
bool compare_priority(const struct list_elem *a,
                      const struct list_elem *b,
                      void *aux UNUSED) {
  return list_entry(a, struct thread, elem)->priority >
         list_entry(b, struct thread, elem)->priority;
}
```

یعنی:

- هر نخ آماده در ready_list وارد می‌شود، ولی همیشه مرتب بر اساس priority نگه داشته می‌شود.
- نخ با **بیشترین اولویت همیشه ابتدای لیست** است.

ب) تابع thread_set_priority :

در نسخه اصلی thread_set_priority فقط priority را عوض می‌کرد.

در نسخه تغییر یافته:

```
void
thread_set_priority (int new_priority)
{
  if (thread_mlfqs)
    return;

  enum intr_level old_level = intr_disable();

  struct thread *current_thread = thread_current();
  int old_priority = current_thread->priority;
  current_thread->base_priority = new_priority;

  if (list_empty(&current_thread->locks) || new_priority > old_priority) {
    current_thread->priority = new_priority;
    thread_yield();
  }

  intr_set_level(old_level);
}
```

نکته‌ها:

- اگر thread_mlfqs فعال باشد، طبق صورت پروژه نباید اجازه دهیم کاربر اولویت را دستی عوض کند، پس فقط return می‌کند.
- در غیر این صورت:
  - base_priority را آپدیت می‌کند.
  - اگر نخ قفلی در دست ندارد یا اولویت جدید بالاتر است، همان اولویت جاری را عوض می‌کند و ممکن است thread_yield() بدهد تا نخ با اولویت بالاتر CPU را بگیرد.
  - اگر نخ قفل‌هایی دارد و اولویت را کم می‌کنیم، اولویت واقعی از طریق donation و thread_update_priority محاسبه می‌شود.

ج) تابع های کمکی برای donation :

نکات:

۱.وقتی نخ holder یک lock است و نخ دیگری با اولویت بالاتر منتظر همان lock است، از طریق lock->max_priority و thread_donate_priority اولویت holder را بالا می‌برد.

۲.thread_update_priority :

- از base_priority شروع می‌کند.
- اگر نخ قفل‌هایی دارد، لیست قفل‌ها را بر اساس max_priority مرتب می‌کند و اگر قفلی با max_priority بالاتر از base باشد، اولویت نخ را به آن مقدار بالا می‌برد.

۳.thread_hold_the_lock :

- قفل تازه گرفته‌شده را در current->locks اضافه می‌کند.
- اگر lock->max_priority از اولویت فعلی نخ بیشتر باشد، نخ را donate می‌کند و thread_yield() می‌کند.

٤.thread_remove_lock :

- موقع lock_release قفل را از لیست حذف می‌کند و دوباره thread_update_priority را صدا می‌زند تا اولویت نخ به مقدار درست برگردد.

۳. تغییرات در synch.c – semaphore / lock / cond :

الف) اولویت در semaphore ها

در sema_down قبلا این اتفاق می‌افتاد :

list_push_back (&sema->waiters, &thread_current ()->elem)

و در نسخه جدید:

```
list_insert_ordered(&sema->waiters, &thread_current()->elem,
                    compare_priority, NULL);
thread_block();
```

یعنی نخ‌های منتظر روی semaphore هم با ترتیب اولویت نگه داری می‌شوند.

در sema_up :

```
old_level = intr_disable ();
if (!list_empty (&sema->waiters)) {

  list_sort(&sema->waiters, compare_priority, NULL);
  thread_unblock(list_entry(list_pop_front(&sema->waiters),
                            struct thread, elem));
}
sema->value++;
thread_yield();
intr_set_level(old_level);
```

ب) Priority Donation در lock ها

در lock_init :

```
lock->holder = NULL;
sema_init (&lock->semaphore, 1);

lock->max_priority = PRI_MIN;
```

در lock_acquire :

```
struct thread *current_thread = thread_current();
struct lock *l;
enum intr_level old_level;

if (lock->holder != NULL && !thread_mlfqs) {
  current_thread->lock_waiting = lock;
  l = lock;
  while (l && current_thread->priority > l->max_priority) {
    l->max_priority = current_thread->priority;
    thread_donate_priority(l->holder);
    l = l->holder->lock_waiting;
  }
}

old_level = intr_disable();
sema_down(&lock->semaphore);

current_thread = thread_current();
if (!thread_mlfqs) {
  current_thread->lock_waiting = NULL;
  lock->max_priority = current_thread->priority;
  thread_hold_the_lock(lock);
}
lock->holder = current_thread;
intr_set_level(old_level);
```

این دقیقا donation چند سطحی را پیاده‌سازی می‌کند:

اگر قفل holder دارد و mlfqs خاموش است:

- نخ فعلی lock_waiting را روی این lock تنظیم می‌کند.
- یک حلقه روی زنجیره‌ی lock ها می‌رود:
  - max_priority را بالا می‌برد.
  - روی holder فعلی donation می‌دهد(thread_donate_priority)
  - به قفلی که holder خودش منتظرش است (l = holder->lock_waiting) حرکت می‌کند.

بعد از گرفتن قفل:

- lock_waitingرا خالی می‌کند.

- قفل فعلی را به عنوان قفل تحت مالکیت این نخ ثبت می‌کند (thread_hold_the_lock)

در lock_release :

```
if (!thread_mlfqs) {
  thread_remove_lock(lock);
}
lock->holder = NULL;
sema_up (&lock->semaphore);
```

- قفل را از لیست locks نخ حذف می‌کند.
- اولویت نخ را دوباره با توجه به base و سایر قفل‌ها تنظیم می‌کند.
- بعد semaphore را up می‌کند تا نخ‌های منتظر بیدار شوند.

ج) اولویت در condition variable :

در cond_signal :

```
if (!list_empty (&cond->waiters))
  list_sort(&cond->waiters, cond_sema_cmp_priority, NULL);
sema_up (&list_entry (list_pop_front (&cond->waiters),
                      struct semaphore_elem, elem)->semaphore);
```

و تابع مقایسه‌ی جدید :

```
bool cond_sema_cmp_priority(const struct list_elem *a,
                            const struct list_elem *b,
                            void *aux UNUSED) {
  struct semaphore_elem *sa = list_entry(a, struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry(b, struct semaphore_elem, elem);

  return list_entry(list_front(&sa->semaphore.waiters),
                    struct thread, elem)->priority
       > list_entry(list_front(&sb->semaphore.waiters),
                    struct thread, elem)->priority;
}
```

وقتی روی یک condition، signal می‌دهد، آن semaphore_elem را انتخاب می‌کند که داخلش نخ با بالاترین priority در انتظار است.

جمع‌بندی فاز ۴ :

- صف آماده‌ها، صف منتظرهایsemaphore ، و صف منتظرهای condition همگی بر اساس priority مرتب شده‌اند.
- به هر threadو lock فیلدهایی برای donation اضافه شد (base_priority، locks، lock_waiting، max_priority)
- در lock_acquireو donation، lock_release تک‌سطحی و چندسطحی پیاده‌سازی شد.
- نتیجه :زمان‌بندی بر اساس اولویت **و** حل مشکل priority inversion

فاز ۵ پروژه Advanced Scheduler (MLFQS) :

زمان‌بندی مبتنی بر اولویت دستی برای برخی بارها و تست‌ها مناسب نیست؛ MLFQS، که اولویت را بر اساس recent_cpu و nice و load_avg محاسبه می‌کند، رفتار عادلانه‌تری برای بارهای mix CPU/I-O ارائه می‌دهد.

هدف فاز: پیاده‌سازی فرمول‌های MLFQS با عدد ثابت (fixed-point) و به‌روزرسانی‌های زمان‌بندی دوره‌ای.

۱.Fixed-Point و فیلد های جدید در thread :

در struct thread :

```
int nice;           /* Niceness. */
fixed_t recent_cpu; /* CPU usage in fixed point. */
```

و در init_thread :

```
t->nice = 0;
t->recent_cpu = FP_CONST(0);
```

این‌ها پایه‌ی فرمول‌های MLFQS هستند.

۲.توابع API mlfqs: nice / load_avg / recent_cpu :

در thread.c :

```
/* Sets the current thread's nice value to NICE. */
void thread_set_nice(int nice) {
  thread_current()->nice = nice;
  thread_mlfqs_update_priority(thread_current());
  thread_yield();
}

/* Returns the current thread's nice value. */
int thread_get_nice(void) {
  return thread_current()->nice;
}

/* Returns 100 times the system load average. */
int thread_get_load_avg(void) {
  return FP_ROUND(FP_MULT_MIX(load_avg, 100));
}

/* Returns 100 times the current thread's recent_cpu value. */
int thread_get_recent_cpu(void) {
  return FP_ROUND(FP_MULT_MIX(thread_current()->recent_cpu, 100));
}
```

۳.به‌روزرسانی های دوره ای در timer_interrupt :

در timer.c :

```
if (thread_mlfqs) {
  thread_mlfqs_increase_recent_cpu_by_one();
  if (ticks % TIMER_FREQ == 0)
    thread_mlfqs_update_load_avg_and_recent_cpu();
  else if (ticks % 4 == 0)
    thread_mlfqs_update_priority(thread_current());
}
```

- هر تیک (به جز idle) recent_cpu نخ جاری را ۱ واحد زیاد می‌کند.
- هر ثانیه :(ticks % TIMER_FREQ == 0) :
  - load_avg را آپدیت می‌کند.
  - recent_cpu تمام نخ‌ها را از نو حساب می‌کند.

- هر **۴ تیک**:
  - اولویت نخی که در حال اجرا است با فرمول MLFQS حساب می‌شود.

۴. توابع هسته ای MLFQS در thread.c :

الف) افزایش recent_cpu :

```
/* Increase recent_cpu by 1 */
void thread_mlfqs_increase_recent_cpu_by_one(void) {
  ASSERT(thread_mlfqs);
  ASSERT(intr_context());

  struct thread *current_thread = thread_current();
  if (current_thread == idle_thread)
    return;
  current_thread->recent_cpu = FP_ADD_MIX(current_thread->recent_cpu, 1);
}
```

ب) محاسبه load_avg و recent_cpu همه نخ‌ها :

```
/* Every per second to refresh load_avg and recent_cpu of all threads */
void thread_mlfqs_update_load_avg_and_recent_cpu(void) {
  ASSERT(thread_mlfqs);
  ASSERT(intr_context());

  size_t ready_threads = list_size(&ready_list);
  if (thread_current() != idle_thread)
    ready_threads++;

  load_avg = FP_ADD(
               FP_DIV_MIX(FP_MULT_MIX(load_avg, 59), 60),
               FP_DIV_MIX(FP_CONST(ready_threads), 60)
             );

  struct thread *t;
  struct list_elem *e = list_begin(&all_list);
  for (; e != list_end(&all_list); e = list_next(e)) {
    t = list_entry(e, struct thread, allelem);
    if (t != idle_thread) {
      /* recent_cpu = (2*load_avg/(2*load_avg + 1)) * recent_cpu + nice */
      t->recent_cpu = ...;
      thread_mlfqs_update_priority(t);
    }
  }
}
```

ج) محاسبه priority بر اساس MLFQS :

```
/* Update priority. */
void thread_mlfqs_update_priority(struct thread *t) {
  if (t == idle_thread)
    return;

  ASSERT(thread_mlfqs);
  ASSERT(t != idle_thread);

  t->priority = FP_INT_PART(
                  FP_SUB_MIX(
                    FP_SUB(FP_CONST(PRI_MAX),
                           FP_DIV_MIX(t->recent_cpu, 4)),
                    2 * t->nice)
                );

  if (t->priority < PRI_MIN) t->priority = PRI_MIN;
  if (t->priority > PRI_MAX) t->priority = PRI_MAX;
}
```

$$priority = PRI\_MAX - (recent\_cpu / 4) - (nice \times 2)$$

۵. هماهنگی MLFQS با priority scheduler :

- در thread_set_priority اگر thread_mlfqs == true فقط returnمی‌کند، یعنی در حالت MLFQS اولویت دستی بی‌معنی است.
- همچنان از ready_list مرتب بر اساس priority استفاده می‌کند، فقط این‌بار priority نه از thread_set_priority بلکه از این فرمول MLFQS می‌آید.

جمع‌بندی فاز ۵ :

- فیلدهای nice، recent_cpu و متغیر سراسری load_avg اضافه شد و با fixed-point پیاده‌سازی شد.
- در timer_interrupt روال استاندارد MLFQS پیاده شد: افزایش recent_cpu هر تیک، آپدیت load_avgو recent_cpuهمه نخ‌ها هر ثانیه، و آپدیت priority هر ۴ تیک.

- توابع thread_set_nice, thread_get_nice, thread_get_load_avg, thread_get_recent_cpu طبق صورت پروژه پیاده سازی شد.
- در حالت MLFQS ، thread_set_priority عملا غیرفعال است و اولویت فقط از روی MLFQS می‌آید.

فاز ٦ پروژه – تست و ارزیابی :

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```