Laboratorio 1 - Sistemas Operativos

—Integrantes—

Angela Jara < angela.jara@unmsm.edu.pe >

Alwin Dávila < alwin.davila@unmsm.edu.pe >

Shamir Mantilla < shamir.mantilla@unmsm.edu.pe >

—Preliminares—

https://www.scs.stanford.edu/24wi-cs212/pintos/pintos.html

https://grail.eecs.csuohio.edu/~cis345s/PintosCSU_Ref.pdf

==========

ALARM CLOCK

==========

—DATA STRUCTURES—

```
struct thread{
...
  int64_t wakeup_tick;
...
}
```

La estructura struct thread requiere un atributo int64_t wakeup_tick, que representa el tiempo específico en el que un hilo debe despertarse. Este campo es clave para determinar cuándo el hilo debe moverse de la lista de sueño (sleep_list) a la lista de hilos listos para ejecutarse.

```
static struct list sleep_list;
```

La variable global static struct list sleep_list se utiliza para mantener una lista ordenada de todos los hilos en estado de sueño, permitiendo una gestión eficiente de sus tiempos de despertar.

—ALGORITHMS—

```
void wakeup_threads(void) {
    while (!list_empty(&sleep_list)) {
        struct thread *wakeup_head = list_entry(list_front(&sleep_list), struct thread,
elem);
```

```
        if (timer_ticks() >= wakeup_head->wakeup_tick) {
            list_pop_front(&sleep_list);  // Remove the thread from the sleep list.
            list_push_back(&ready_list, &(wakeup_head->elem));  // Add it to the ready
list.
        } else {
            // If the head thread's wakeup time has not arrived, exit the loop.
            break;
        }
    }
}
```

Recorre la lista de hilos en estado de sueño (sleep_list) para comprobar si alguno de ellos debe ser despertado. Compara el tiempo actual (timer_ticks()) con el campo wakeup_tick de la cabeza de la lista. Si el tiempo actual es mayor o igual que el tiempo de despertar del hilo, este se elimina de sleep_list y se agrega a la lista de hilos listos para ejecutarse (ready_list). Si el tiempo actual aún no ha alcanzado el momento de despertar, la función termina el bucle.

```
void
thread_sleep(int64_t ticks)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;
  old_level = intr_disable ();
  if(cur!=idle_thread){
    cur->status=THREAD_BLOCKED;
    cur->wakeup_tick=ticks;
```

```
    list_insert_ordered(&sleep_list,&cur->elem,thread_wakeup_tick_cmp,NULL);
    schedule();
  }
  intr_set_level (old_level);
}
```

Permite que el hilo actual entre en estado de sueño por un número específico de ciclos de temporizador (ticks). Se desactiva temporalmente la interrupción para evitar condiciones de carrera y asegurar la operación atómica. Luego, si el hilo actual no es el hilo inactivo (idle_thread), se establece su estado como THREAD_BLOCKED, se asigna el tiempo de despertar a wakeup_tick y se inserta de forma ordenada en la lista sleep_list según su tiempo de despertar usando la función de comparación thread_wakeup_tick_cmp. Finalmente, se invoca la función schedule() para permitir que otros hilos puedan ejecutarse mientras el hilo actual está dormido. Se restauran las interrupciones al final de la operación.

—RATIONALE—

El diseño se basa en una cola ordenada (sleep_list) para rastrear el tiempo de despertar de los hilos. El método thread_sleep coloca hilos en esta lista con su tiempo de sueño, mientras que wakeup_threads verifica periódicamente los ticks actuales para mover los hilos listos a la cola de ejecución (ready_list) cuando es necesario. Esto asegura una operación eficiente y libre de condiciones de carrera.

==========

PRIORITY SCHEDULING

==========

—DATA STRUCTURES—

```
struct
thread
{
  ...
  int base_priority;
  int priority;
  struct list donors;
  struct list_elem donation_elem;
  struct lock *wait_lock;
  ...
}
```

Se agregan los atributos base_priority para la prioridad base del hilo, priority para la prioridad actual, y donors, una lista que rastrea las donaciones de prioridad. Además, se incluye wait_lock para gestionar la sincronización mediante bloqueo.

—ALGORITHMS—

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);

  t->status = THREAD_READY;
  list_insert_ordered(&ready_list,&t->elem,thread_priority_great,NULL);

  if (thread_current() != idle_thread &&thread_current()->priority < t->priority ){
    thread_yield();
  }

  intr_set_level (old_level);
}
```

Esta función se encarga de desbloquear un hilo que estaba en estado de bloqueo (THREAD_BLOCKED). Desactiva temporalmente las interrupciones para garantizar una operación segura. Cambia el estado del hilo a THREAD_READY y lo inserta en la lista de hilos listos para ejecutar (ready_list) de manera ordenada según su prioridad usando thread_priority_great. Si el hilo desbloqueado tiene una prioridad mayor que el hilo actual, se ejecuta un cambio de contexto con thread_yield().

```
void thread_donate_priority(struct thread *recipient, struct thread *donor) {
  enum intr_level old_level;
```

```
  old_level = intr_disable ();


  recipient->priority = donor->priority;

  list_insert_ordered(&recipient->donors,
&donor->donation_elem,thread_priority_great,NULL);


  //Check if after donation yield is needed

  if (recipient == thread_current()) {

    struct thread *t_max_ready = list_entry(list_begin(&ready_list), struct thread,
elem);

    if (t_max_ready != NULL && t_max_ready->priority > recipient->priority) {

      thread_yield();

    }

  }


  intr_set_level(old_level);

}
```

La función thread_donate_priority() permite transferir la prioridad de un hilo donante (donor) a un hilo receptor (recipient). Se desactiva temporalmente las interrupciones para asegurar que la operación sea segura. Luego, se actualiza la prioridad del hilo receptor y se agrega el hilo donante a la lista de donaciones del receptor (recipient->donors). Si el hilo receptor es el hilo actual y su prioridad resulta ser menor que la del hilo con la prioridad más alta en la cola de hilos listos (ready_list), se ejecuta un cambio de contexto con thread_yield() para permitir la reprogramación de hilos.

—RATIONALE—

El diseño implementa una estrategia de prioridad dinámica con donaciones, lo que permite una asignación de recursos más flexible en escenarios donde la espera por recursos puede llevar a la inversión de prioridades. El uso de atributos como base_priority, priority, y donors es clave para implementar este

comportamiento, ya que permiten rastrear cambios en la prioridad de un hilo y aplicar mecanismos de donación. La función thread_donate_priority() es eficiente, ya que ajusta la prioridad de un hilo de forma segura y verifica si es necesario cambiar el contexto mediante thread_yield(). Sin embargo, la gestión de las donaciones puede ser compleja si la lista donors crece mucho, ya que implicaría realizar operaciones sobre múltiples elementos. En general, el diseño es efectivo pero puede requerir optimización en aplicaciones con una alta cantidad de hilos bloqueados para evitar un alto costo computacional.

==========

SCHEDULER 4BSD

==========

—DATA STRUCTURES—

```
struct
thread
{
  ...
  int nice;
  fixed_point recent_cpu;
  ...
}
```

Se agregan atributos adicionales para la planificación 4BSD. El atributo nice permite ajustar la prioridad de un hilo según la configuración de la política de planificación. recent_cpu (de tipo fixed_point) mide el tiempo de CPU recientemente utilizado por el hilo, lo que ayuda en el cálculo dinámico de prioridades dentro del esquema MLFQS.

—ALGORITHMS—

```
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;

  if (thread_mlfqs)
```

```
  {
    thread_mlfqs_inc_recent_cpu ();
    if (ticks % TIMER_FREQ == 0){
      mlfqs_updt_load_average();
      thread_mlfqs_updt_recent_cpu ();
    }
    if (ticks % 4 == 0){
      thread_mlfqs_updt_priority ();
    }
  }
  thread_tick ();
}
```

La función timer_interrupt se ejecuta periódicamente en cada interrupción de temporizador. Incrementa el contador de ticks y, si la planificación MLFQS está habilitada (thread_mlfqs), actualiza los valores relacionados con la CPU y la prioridad. Cada cierto número de ticks (TIMER_FREQ) actualiza la carga promedio (mlfqs_updt_load_average) y recalcula el uso reciente de CPU para los hilos (thread_mlfqs_updt_recent_cpu). También verifica cada 4 ticks si es necesario ajustar las prioridades de los hilos listados en la cola de ejecución.

```
void thread_mlfqs_updt_priority(void) {
  struct list_elem *e;
  struct thread *t;
  for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e)) {
    t = list_entry(e, struct thread, allelem);
    if (t == idle_thread) {
      continue;
```

```
    }

    thread_mlfqs_priority(t);

  }

  list_sort(&ready_list, thread_priority_great, NULL);

  if (!list_empty(&ready_list) &&

      thread_current() != idle_thread &&

      thread_current()->priority < list_entry(list_front(&ready_list), struct thread,
elem)->priority) {

    thread_yield();

  }

}
```

Esta función actualiza la prioridad de todos los hilos en la lista all_list, exceptuando el hilo inactivo (idle_thread). Luego ordena la cola de hilos listos (ready_list) según sus prioridades usando la función list_sort. También verifica si el hilo actualmente ejecutándose tiene una prioridad más baja que el hilo con mayor prioridad en la cola de ejecución, y en ese caso, cede la CPU a otro hilo mediante thread_yield(). Esto asegura que los hilos con mayor prioridad puedan ejecutarse oportunamente.

—RATIONALE—

El diseño implementa el MLFQS para asignar dinámicamente prioridades a los hilos según su uso de CPU, con el fin de equilibrar eficiencia y equidad en la asignación de recursos. La función timer_interrupt() actualiza periódicamente las métricas clave, como la carga promedio y el uso reciente de CPU. Por su parte, thread_mlfqs_updt_priority() ajusta las prioridades de los hilos en la cola de ejecución, garantizando que los hilos con mayor prioridad puedan ejecutarse primero. Aunque funcional, el enfoque podría tener sobrecostos en sistemas con muchos hilos activos, especialmente por el recorrido completo de la lista de hilos. Sin embargo, es modular y eficiente en escenarios estándar.