

Project #3: Threads

[CSE4070]

Fall 2019

Hyeongu Kang

Contents

1. Process Scheduling
2. Threads
3. Synchronization
4. Requirements
5. Evaluation
6. Documentation
7. Submission

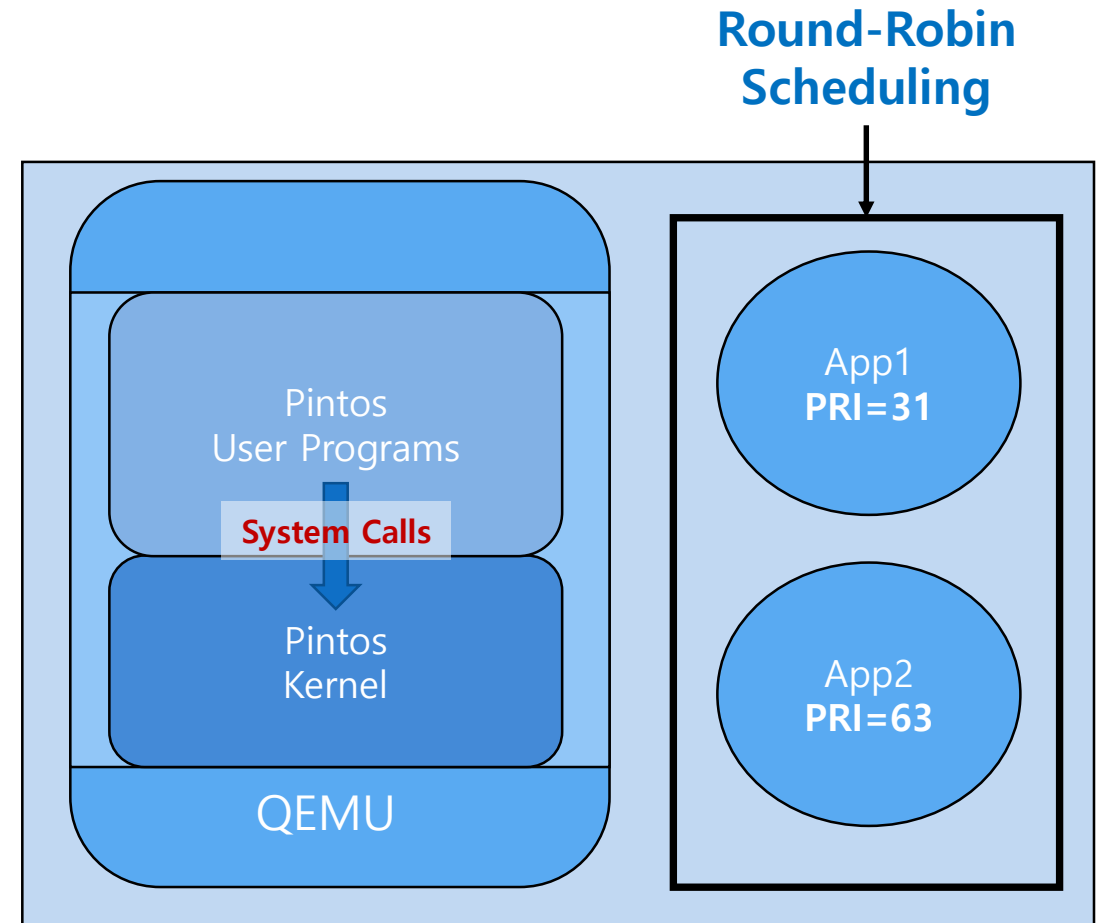
Notes

- "Project #3: Threads" in the class matches "Project 1: Threads" in Pintos document.
- You can find the information of this project in Chapter 2 of Pintos document.

Process Scheduling

Sketch of Pintos

- Until project #2, we focused on the things related with user programs.
 - User stack and argument passing
 - System call handler and system calls (using file system API)
 - Protecting inappropriate memory access
- Due to your efforts, current Pintos can run most of user programs which resides in src/examples.
- However, Pintos uses simple scheduler, round-robin scheduler.
- **It means Pintos doesn't consider the priority of each process or thread.**



Schedulers

- We've learned variety of schedulers in Chapter 5.
 - FIFO (First In, First Out)
 - SJF (Shortest Job First)
 - STCF (Shortest Time-to-Completion First)
 - RR (Round-Robin)
- Pintos uses **round-robin scheduler** as default scheduler.

Default Scheduler in Pintos

```
309 void
310 thread_yield (void)
311 {
312     struct thread *cur = thread_current ();
313     enum intr_level old_level;
314
315     ASSERT (!intr_context ());
316
317     old_level = intr_disable ();
318     if (cur != idle_thread)
319         list_push_back (&ready_list, &cur->elem);
320     cur->status = THREAD_READY;
321     schedule ();
322     intr_set_level (old_level);
323 }
```

thread_yield() push current thread at the end of the ready list and calls **schedule()**.

Default Scheduler in Pintos

```
555 static void
556 schedule (void)
557 {
558     struct thread *cur = running_thread ();
559     struct thread *next = next_thread_to_run ();
560     struct thread *prev = NULL;
561
562     ASSERT (intr_get_level () == INTR_OFF);
563     ASSERT (cur->status != THREAD_RUNNING);
564     ASSERT (is_thread (next));
565
566     if (cur != next)
567         prev = switch_threads (cur, next);
568     thread_schedule_tail (prev);
569 }
```

`schedule()` calls `next_thread_to_run()` to find next thread to be run

Default Scheduler in Pintos

```
493 static struct thread *  
494 next_thread_to_run (void)  
495 {  
496     if (list_empty (&ready_list))  
497         return idle_thread;  
498     else  
499         return list_entry (list_pop_front (&ready_list), struct thread, elem);  
500 }
```

`next_thread_to_run()` pops the first thread in the ready list

Default Scheduler in Pintos

```
555 static void
556 schedule (void)
557 {
558     struct thread *cur = running_thread ();
559     struct thread *next = next_thread_to_run ();
560     struct thread *prev = NULL;
561
562     ASSERT (intr_get_level () == INTR_OFF);
563     ASSERT (cur->status != THREAD_RUNNING);
564     ASSERT (is_thread (next));
565
566     if (cur != next)
567         prev = switch_threads (cur, next);
568     thread_schedule_tail (prev);
569 }
```

schedule() calls **switch_threads()** to switch process

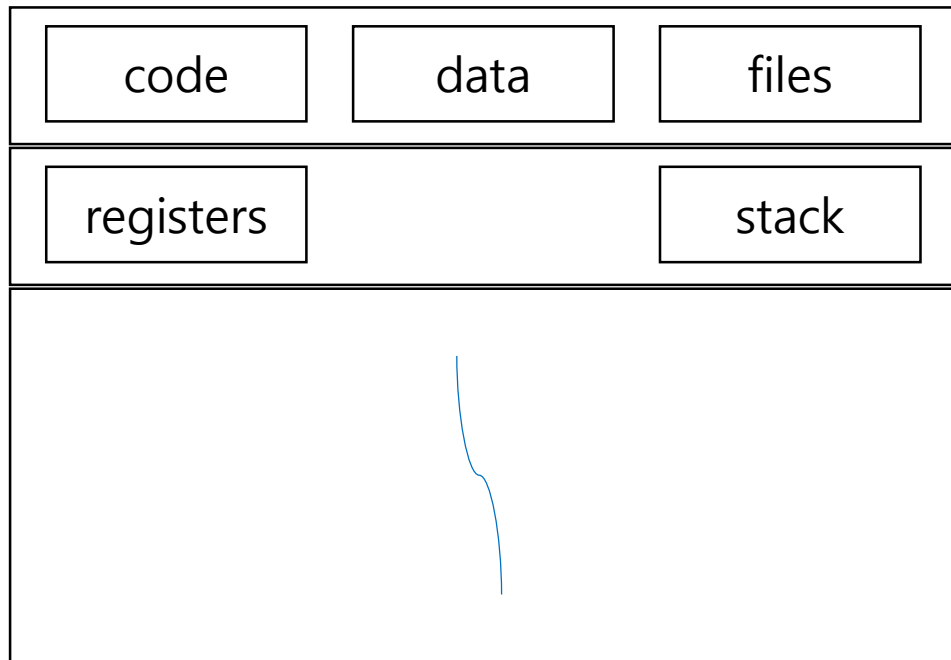
Schedulers

- Implement complex scheduler which considers thread's priority
- For scheduler, relevant components are threads and synchronization techniques.
- Threads are the objects of scheduling.
- Synchronization such as semaphores or locks should be used in the scheduler to arrange order of thread execution.

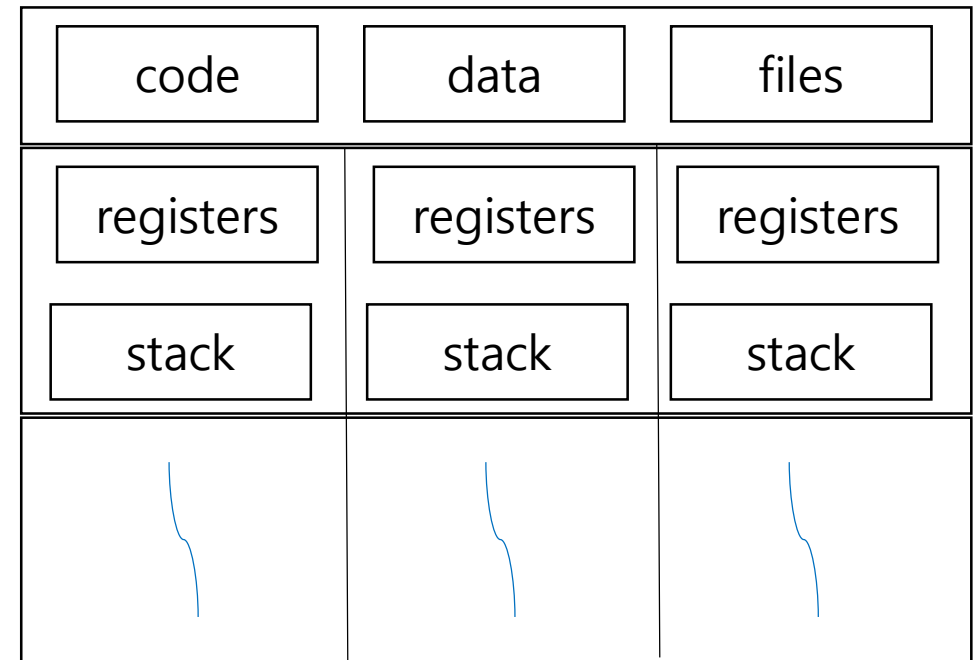
Threads

Threads

- A thread is a basic unit of CPU utilization.
- It shares code, data and other resources with other threads belonging to the same process.
- If a process only has one thread in it, we can consider this thread as a process.



Single-threaded process

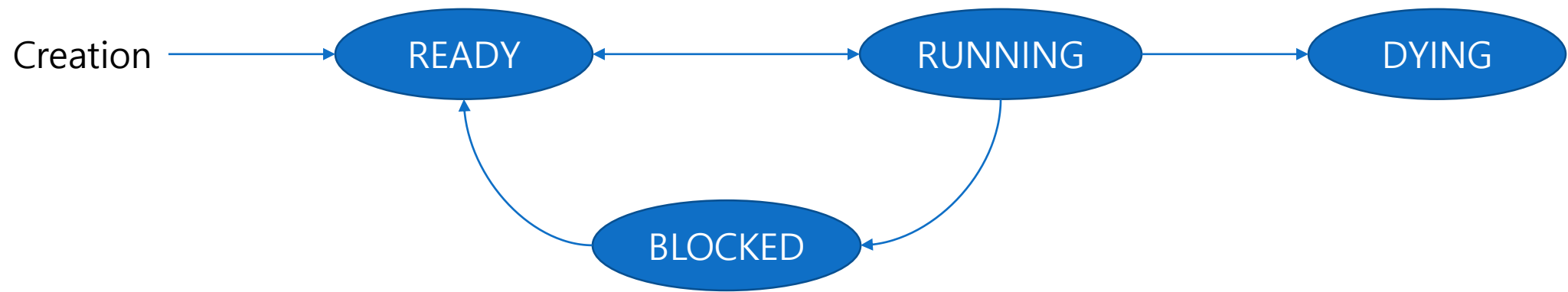


Multithreaded process

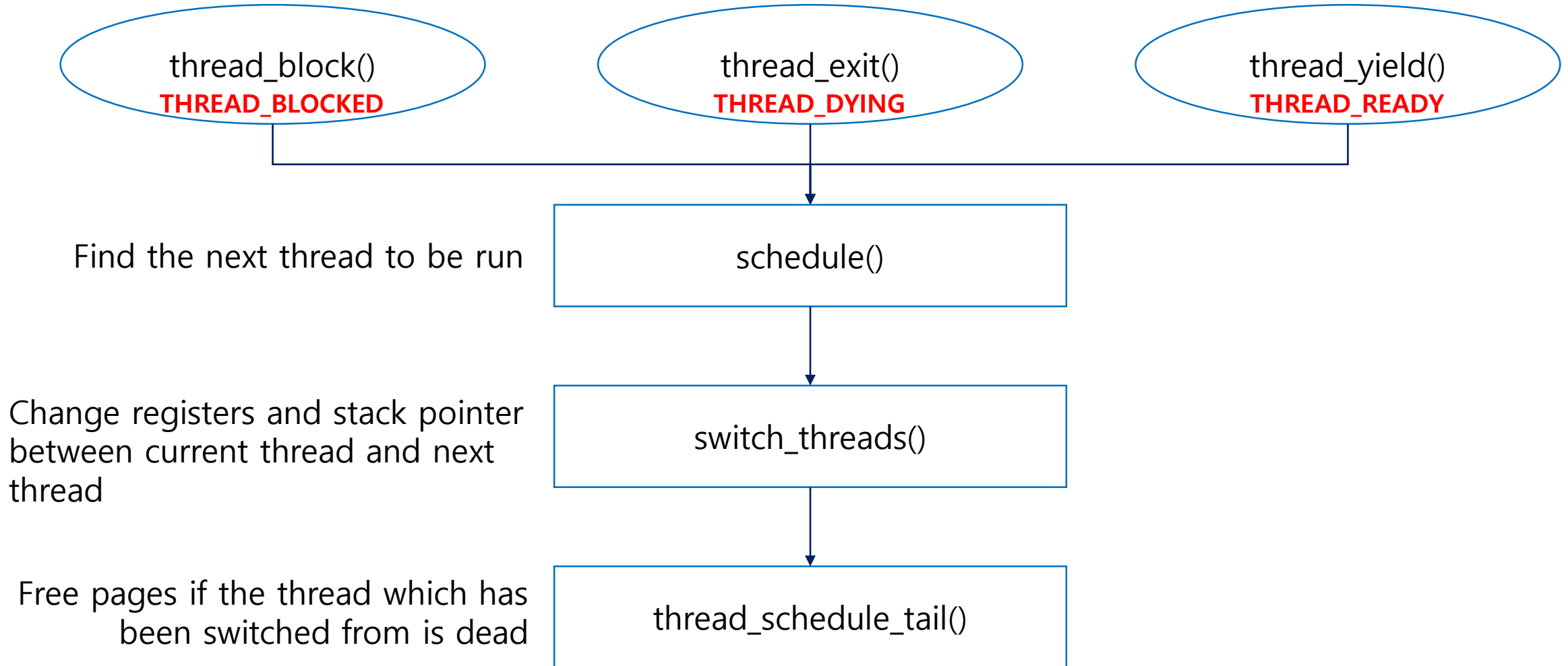
Thread Status

- A thread can have one of four status.

```
/* States in a thread's life cycle. */  
enum thread_status  
{  
    THREAD_RUNNING,    /* Running thread. */  
    THREAD_READY,      /* Not running but ready to run. */  
    THREAD_BLOCKED,    /* Waiting for an event to trigger. */  
    THREAD_DYING       /* About to be destroyed. */  
};
```



Thread Switching



Synchronization

Semaphores

- **A semaphore is a nonnegative integer** with down and up operators.
- When `sema_down()` is called, if the value of semaphore is 0, the thread that calls `sema_down()` is blocked.
- `sema_up()` wakes up one of blocked threads.

```
void  
sema_init (struct semaphore *sema, unsigned value)  
{  
    ASSERT (sema != NULL);  
  
    sema->value = value;  
    list_init (&sema->waiters);  
}
```

Semaphores

- A semaphore is a nonnegative integer with down and up operators.
- When `sema_down()` is called, **if the value of semaphore is 0, the thread that calls `sema_down()` is blocked.**
- `sema_up()` wakes up one of blocked threads.

```
void
sema_down (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);
    ASSERT (!intr_context ());

    old_level = intr_disable ();
    while (sema->value == 0)
    {
        list_push_back (&sema->waiters, &thread_current ()->elem);
        thread_block ();
    }
    sema->value--;
    intr_set_level (old_level);
}
```

Semaphores

- A semaphore is a nonnegative integer with down and up operators.
- When `sema_down()` is called, if the value of semaphore is 0, the thread that calls `sema_down()` is blocked.
- `sema_up()` **wakes up one of blocked threads.**

```
void
sema_up (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);

    old_level = intr_disable ();
    if (!list_empty (&sema->waiters))
        thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                                struct thread, elem));
    sema->value++;
    intr_set_level (old_level);
}
```

Locks

- Lock is a specialization of a semaphore with an **initial value of 1**.
- It also has two operators, acquire (equivalent of down in semaphore) and release (equivalent of up)

```
void  
lock_init (struct lock *lock)  
{  
    ASSERT (lock != NULL);  
  
    lock->holder = NULL;  
    sema_init (&lock->semaphore, 1);  
}
```

Locks

- Lock is a specialization of a semaphore with an initial value of 1.
- It also has two operators, **acquire (equivalent of down in semaphore)** and release (equivalent of up)

```
void
lock_acquire (struct lock *lock)
{
    ASSERT (lock != NULL);
    ASSERT (!intr_context ());
    ASSERT (!lock_held_by_current_thread (lock));

    sema_down (&lock->semaphore);
    lock->holder = thread_current ();
}
```

Locks

- Lock is a specialization of a semaphore with an initial value of 1.
- It also has two operators, acquire (equivalent of down in semaphore) and **release (equivalent of up)**

```
void
lock_release (struct lock *lock)
{
    ASSERT (lock != NULL);
    ASSERT (lock_held_by_current_thread (lock));

    lock->holder = NULL;
    sema_up (&lock->semaphore);
}
```

Requirements

- Alarm Clock
- Priority Scheduling
- Advanced Scheduler (BSD Scheduler)
 - > Additional

Alarm Clock

- timer_sleep() is used to let the thread fall asleep.
- It is implemented with busy waiting technique.
- Though it calls thread_yield() immediately when the timer is not expired, it's not efficient since the thread iterates between RUNNING state and READY state.
- Thus, we will modify this to avoid busy waiting.

```
87 /* Sleeps for approximately TICKS timer ticks.  Interrupts must
88    be turned on. */
89 void
90 timer_sleep (int64_t ticks)
91 {
92     int64_t start = timer_ticks ();
93
94     ASSERT (intr_get_level () == INTR_ON);
95     while (timer_elapsed (start) < ticks)
96         thread_yield ();
97 }
```

← Get current time

← Yield CPU until the "ticks" has gone by

Alarm Clock

- How to avoid busy waiting?
 - After checking that "ticks" has gone by, if not, block the thread (BLOCKED state).
 - To manage these threads, **create a new queue** to store it.
 - When the thread is inserted into the queue, wake up time should be saved as well.
 - When time is up, wake up the thread and insert it into ready queue (ready_list).
- How can you check that the time is up after the thread is blocked?
 - **timer_interrupt()** is called every tick.
 - Find the threads that need to be woken up in this function.
 - If it is the case, insert it into ready queue.

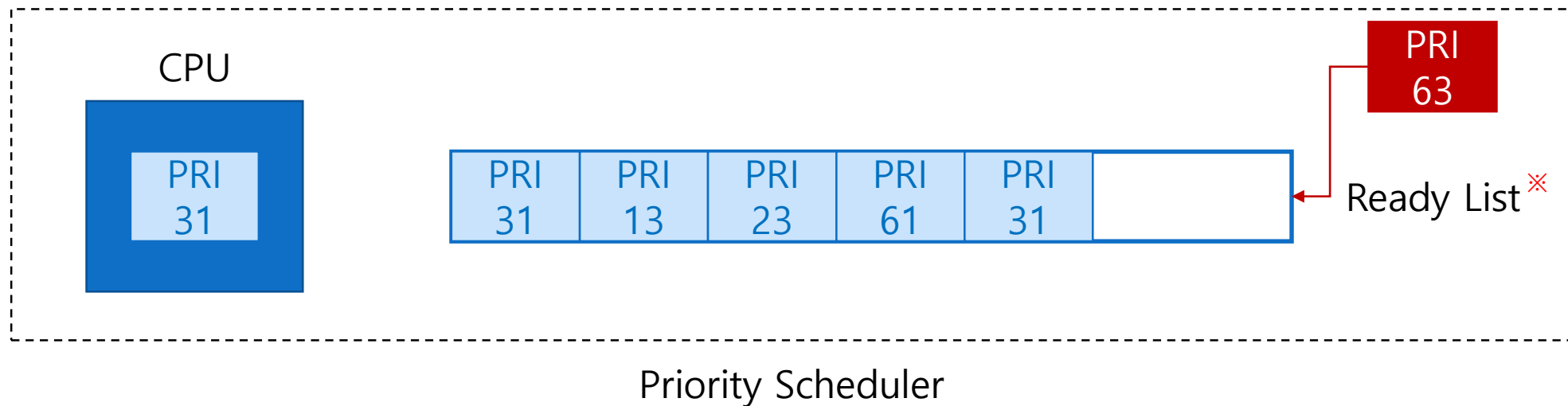
```
169 /* Timer interrupt handler. */
170 static void
171 timer_interrupt (struct intr_frame *args UNUSED)
172 {
173     ticks++;
174     thread_tick ();
175 }
```

Priority Scheduling

- Until now, Pintos performs round-robin scheduling for threads.
- When `thread_yield()` or `thread_unblock()` is called, the current thread or unblocked thread are inserted at the end of the ready list regardless of its priority.

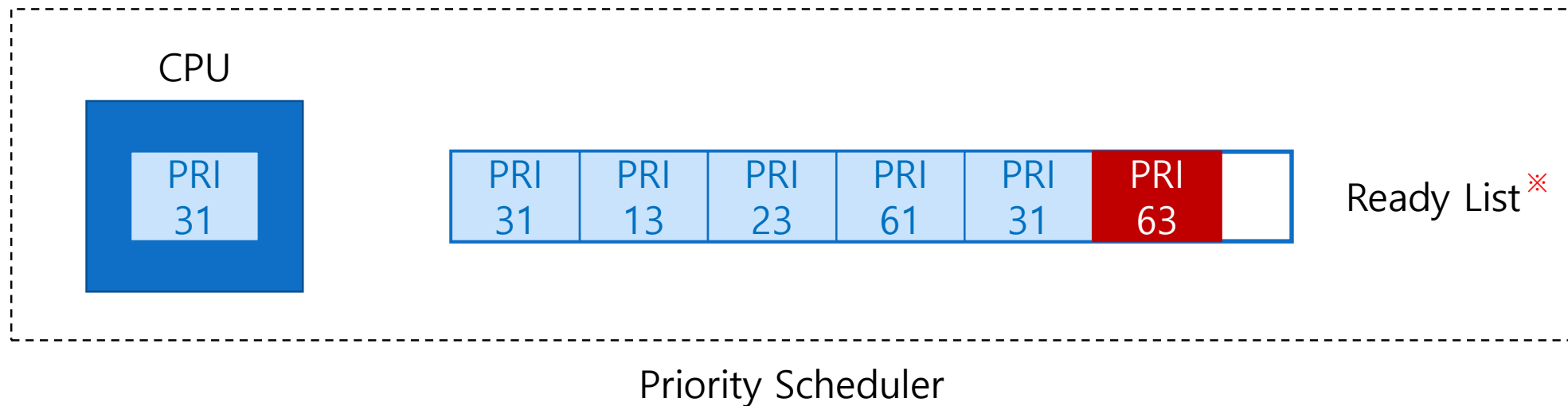
Priority Scheduling

- In this project, we are to implement complex scheduler which considers priority of the threads.
- If there comes new thread that has higher priority than current thread in ready list, the current thread should immediately yield the CPU to new thread.



Priority Scheduling

- In this project, we are to implement complex scheduler which considers priority of the threads.
- If there comes new thread that has higher priority than current thread in ready list, the current thread should immediately yield the CPU to new thread.



Priority Scheduling

- Thread priorities range from PRI_MIN (0) to PRI_MAX (63).
- Default priority value is PRI_DEFAULT (31).
- Lower numbers correspond to lower priorities, so that priority 0 is the lowest and 63 is the highest.
- The initial thread priority is passed as an argument to thread_create ().

```
/* Thread priorities. */  
#define PRI_MIN 0           /* Lowest priority. */  
#define PRI_DEFAULT 31      /* Default priority. */  
#define PRI_MAX 63         /* Highest priority. */
```

Priority Scheduling

- Thread priorities range from PRI_MIN (0) to PRI_MAX (63).
- Default priority value is PRI_DEFAULT (31).
- Lower numbers correspond to lower priorities, so that priority 0 is the lowest and 63 is the highest.
- The initial thread priority is passed as an argument to thread_create ().

```
tid_t  
thread_create (const char *name, int priority,  
               thread_func *function, void *aux)
```

Priority Scheduling - Aging

- Default priority scheduling invokes starvation of processes which have low priority.
- Thus, we need aging technique that increases the priority in proportion to the time passed after the process resides in ready list.
- **Before implementing aging technique, modify the codes as in the next page.**
- The codes indicate that aging technique is used only when Pintos kernel gets -aging option and sets aging flag to TRUE.

Priority Scheduling - Aging

```
4 #include <debug.h>
5 #include <list.h>
6 #include <stdint.h>
7 #include "threads/synch.h" /* Project #3. */
8
9 #ifndef USERPROG
10 /* Project #3. */
11 extern bool thread_prior_aging;
12 #endif
13
14 /* States in a thread's life cycle. */
15 enum thread_status
16 {
17     THREAD_RUNNING, /* Running thread. */
18     THREAD_READY,   /* Not running but ready to run. */
19     THREAD_BLOCKED, /* Waiting for an event to trigger. */
20     THREAD_DYING    /* About to be destroyed. */
21 };
```

src/threads/thread.h

```
    else if (!strcmp (name, "-rs"))
        random_init (atoi (value));
    else if (!strcmp (name, "-mlfqs"))
        thread_mlfqs = true;
    #ifndef USERPROG
    /* Project #3. */
    else if (!strcmp (name, "-aging"))
        thread_prior_aging = true;
    #endif
    #ifdef USERPROG
    else if (!strcmp (name, "-ul"))
        user_page_limit = atoi (value);
    #endif
```

src/threads/init.c

```
63 /* Scheduling. */
64 #define TIME_SLICE 4
65 static unsigned thread_ticks;
66
67 #ifndef USERPROG
68 /* Project #3. */
69 bool thread_prior_aging;
70 #endif
71
72 /* If false (default), use round-robin scheduling.
73    If true, use multi-level feedback queue scheduling.
74    Controlled by kernel command line option -mlfqs. */
75 bool thread_mlfqs;
```

```
/* Enforce preemption. */
if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return ();

#ifndef USERPROG
/* Project #3. */
thread_wake_up ();

/* Project #3. */
if (thread_prior_aging == true)
    thread_aging ();
#endif
```

src/threads/thread.c

Priority Scheduling - Aging

- **Replace tests to existing tests directory**
 - Extract threads_tests.tar and overwrite extracted directory to src/tests/threads
- How to implement aging
 - Check that `thread_prior_aging` is TRUE
 - If it is TRUE, increase the priority proportional to the time spent without occupying the CPU as tick is increased.

Advanced Scheduler - BSD Scheduler

- This is additional implementation of this project.
- You can find the information about this in Appendix B 4.4BSD Scheduler.

Advanced Scheduler - BSD Scheduler

- BSD scheduler is general purpose scheduler.
 - Multi-Level Feedback Queue (MLFQ) or Multi-Level Ready Queue (MLRQ) is generally used in general purpose scheduler.
 - Each priority has its own ready queue.
 - When `schedule()` is invoked, thread is selected from the highest priority queue.
 - Ready queue of each priority follows round robin policy.
- In this project, you can use MLFQs of 64 queues or MLFQ of 1 queue.
- MLFQs of 64 queues will be covered in this slide.

Advanced Scheduler - BSD Scheduler

Select the thread
in the highest ready queue

CPU



...



...



MLFQ

Round Robin

Advanced Scheduler - Niceness

- Each thread in Pintos has **nice** value in the range from -20 to 20.
- Positive nice value lower the priority so that other threads can occupy CPU.
 - nice value 0 doesn't affect the priority.
- Initial nice value of the thread
 - If the thread is created initially, set nice value to 0.
 - If not, the thread starts with a nice value inherited from their parent thread.
- **Functions to implement**
 - `int thread_get_nice (void)`
 - Returns the current thread's nice value
 - `void thread_set_nice (int new_nice)`
 - Set the current thread's nice value to `new_nice`
 - Recalculates the thread's priority based on the new value
 - If the running thread no longer has the highest priority, yields

Advanced Scheduler - Calculating Priority

- Scheduler has priorities of 64 level.
 - Maximum priority: 63 (PRI_MAX)
 - Minimum priority: 0 (PRI_MIN)
 - 64 ready queues are generally used, but you can use only 1 ready queue.
- Calculating Priority
 - Initial priority is decided in `thread_create()`
 - Every 4 tick, priorities of all thread in the system are recalculated.
 - Formula for calculating priority
 - $priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)$
 - *recent_cpu*: Estimate of the CPU time the thread has used recently
 - *nice*: nice value of the thread
 - Based on the formula of BSD scheduler, the thread that had much CPU time will get lower priority in the next scheduling.

Advanced Scheduler - Calculating recent_cpu

- *recent_cpu*
 - It estimates CPU time of the thread.
 - More recent CPU time should be weighted more heavily than less recent CPU time.
 - Initial recent_cpu value of the thread:
 - If it is created first, the value is 0.
 - If not, inherits value of the parent thread.
 - Whenever time interrupt is invoked, recent_cpu value of the thread in RUNNING state is increased by 1. (Except for idle thread)
- recent_cpu value of all thread (RUNNING, READY and BLOCKED) is recalculated every second.
 - $recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice$
 - *load_avg* : average of the number of thread in READY state
- **Functions to Implement**
 - Int thread_get_recent_cpu (void)
 - Returns **100 times** the current thread's recent_cpu value
 - Rounded up to the nearest integer.

Advanced Scheduler - Calculating load_avg

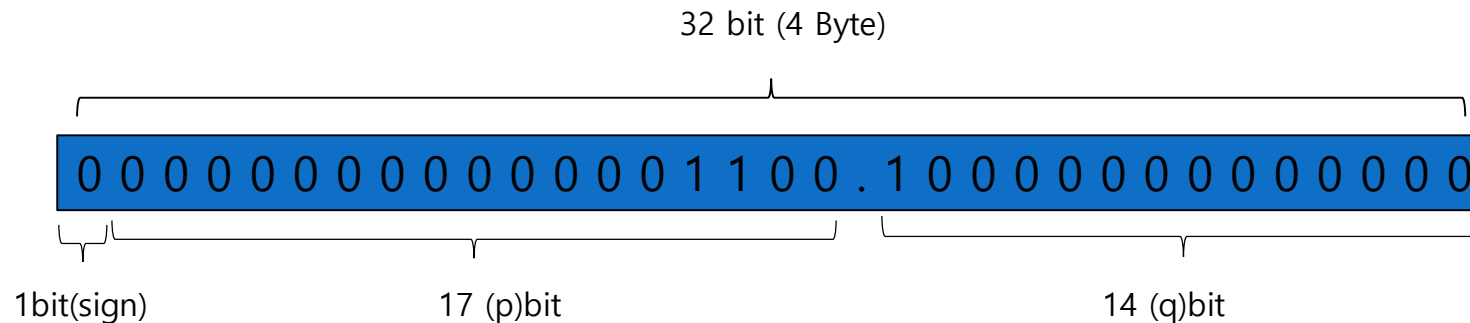
- *load_avg*
 - System-wide value
 - Initialized to 0 when system is booted.
- load_avg value is updated every second.
 - $load_avg = (59/60) * load_avg + (1/60) * ready_threads$
 - *ready_threads* : number of thread in READY or RUNNING state (Except for idle thread)
- **Functions to Implement**
 - thread_get_load_avg (void)
 - Returns 100 times the current system load average,
 - Rounded to the nearest integer.

Advanced Scheduler - Summary

- Thread can manage nice value in range from -20 to 20.
- Range of priority: 0 (PRI_MIN) - 63 (PRI_MAX)
- Priority
 - Recalculate priority in every 4 ticks (same as TIME_SLICE)
 - $priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)$
- *recent_cpu*
 - *recent_cpu* indicates recent CPU time of the thread.
 - *recent_cpu* value of the thread in RUNNING state is increased by 1 in every tick.
 - *recent_cpu* value of all thread is updated in every second (1sec = TIMER_FREQ)
 - $recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice$
- *load_avg*
 - Estimate the average of number of thread in READY state.
 - Initialized to 0 when it is booted.
 - *load_avg* value is updated in every second.
 - $load_avg = (59/60) * load_avg + (1/60) * ready_threads$

Advanced Scheduler - Fixed-Point Real Arithmetic

- Pintos kernel doesn't support floating point arithmetic.
- But in BSD scheduler, real numbers such as `recent_cpu` and `load_avg` are used.
- Fixed-point format is used instead of floating point arithmetic.
 - p.q format: p is integer and q is fraction
 - For 32-bit, 1 bit for sign, 17 bits for integer (p) and 14 bits for fraction (q)
- Example of fixed-point number
- $12.50 \rightarrow 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^{-1}$
- Calculate $12.50 \cdot 2^{14}$ to represent 12.50 in fixed-point real arithmetic.



Evaluation

- 13 tests will be graded.
(Refer to the test case list in the next slide)
- Total score is 100 which consists of 80 for test cases and 20 for documentation.
- Refer to `src/tests/threads/Grading`, `src/tests/threads/Rubric.alarm` and `src/tests/threads/Rubric.priority` to check the points of each test.

Evaluation

- ✓alarm-single
- ✓alarm-multiple
- ✓alarm-simultaneous
- ✓alarm-priority
- ✓alarm-zero
- ✓alarm-negative
- ✓priority-change
- ✓priority-change-2
- ✓priority-fifo
- ✓**priority-lifo**
- ✓priority-preempt
- ✓priority-sema
- ✓priority-aging
- ✓**Total : 13 tests**

Evaluation

- priority-lifo can't be checked by make check
- Run the following command:
 - ✓ `pintos -v -- -q run priority-lifo`
- **Analyze the code and result of priority-lifo test in the documentation.**

Evaluation

- **Additional Requirement**

- 5% additional point for BSD Scheduler implementation
 - Describe in the documentation where and how you implemented the BSD scheduler.
 - Mlfs related tests will pass when implementing BSD Scheduler.

- ✓mlfs-block
- ✓mlfs-fair-2
- ✓mlfs-fair-20
- ✓mlfs-load-1
- ✓mlfs-load-60
- ✓mlfs-load-avg
- ✓mlfs-nice-10
- ✓mlfs-nice-2
- ✓mlfs-recent-1
- ✓Total : 9 tests

Documentation

- Use the document file uploaded on e-class
- Documentation accounts for 20% of total score.
(Development 80%, Documentation 20%)

Submission

- Before submission, check that you performed the following:
 1. **Make clean** before compressing **(in pintos/src directory)**
 2. Submission form (Refer to the next slide)
 3. Once you copy other's codes, you will get **F grade**

Submission

- Due date : **2019. 12. 8 23:59**
- Submission
 - The form of submission file is as follows:

Name of compressed file	Example (project 3, Group #7)
os_prj3_##.tar.gz	os_prj3_07.tar.gz

- Only one person of group should submit the file.
- **You should also submit the design document in hardcopy (AS916).**
(Due date of hardcopy is same as the compressed file)

Submission

- **Contents**

- ① Pintos source codes
- ② document: **document.doc** or **document.docx** (Other format is not allowed such as .hwp)

- **Form and way to submit**

- 1) How to make tar.gz file

- Run 'make clean' in pintos/src
 - Make new directory with your group number
 - Copy 'pintos' directory (not pintos/src) into the new directory
 - Copy the document file into the new directory
 - Compress the new directory into os_prj3_##.tar.gz **(tar -zcf os_prj3_[Group#].tar.gz ./[Group#])**
 - ✓ **For example, if your group number is 7, tar -zcf os_prj3_07.tar.gz ./07**
 - ✓ You should use -zcf options for using tar

- 2) Way to submit: Upload the tar.gz file to e-class

- **Submit Hardcopy to AS916 (You should submit softcopy and hardcopy both, if not 3% of point will be deducted)**

- 3) Due date: 2019. 12. 8. 23:59

- ❖ **5% of point will be deducted for a wrong form and way to submit**
 - ❖ **Late submission is allowed up to 3 days (~12/11) and 10% of point will be deducted per day**

Project Schedule

Projects	Points	Contents	Periods	Lectures
Project 0-1	1	Installing Pintos	9/16 – 9/22	Manual will be provided
Project 0-2	3	Pintos Data Structures	9/21 – 10/6	9/21 (Sat.)
Project 1	6	User Programs (1)	10/5 – 11/3	10/5 (Sat.)
Project 2	4	User Programs (2)	11/2 – 11/17	11/2 (Sat.)
Project 3	6	Threads	11/16 – 12/8	11/16 (Sat.)

※ Once you copy other's codes, you will get **F grade**