POSTECH

CSED 312:

Operating System Lab

Project 1: Threads

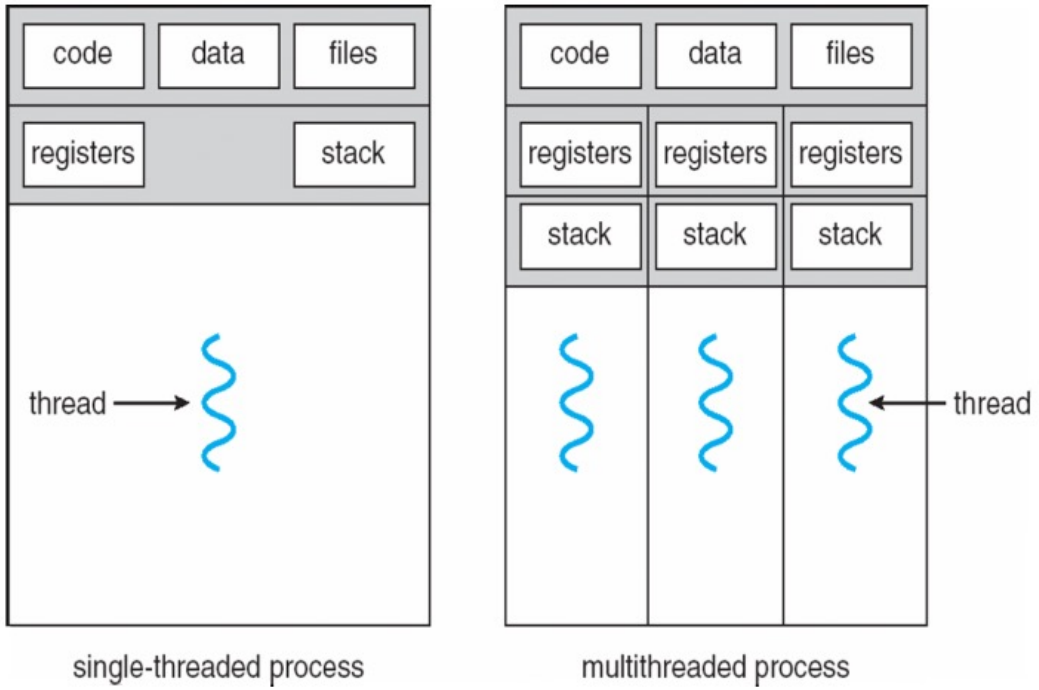
Autumn 2023

Overview

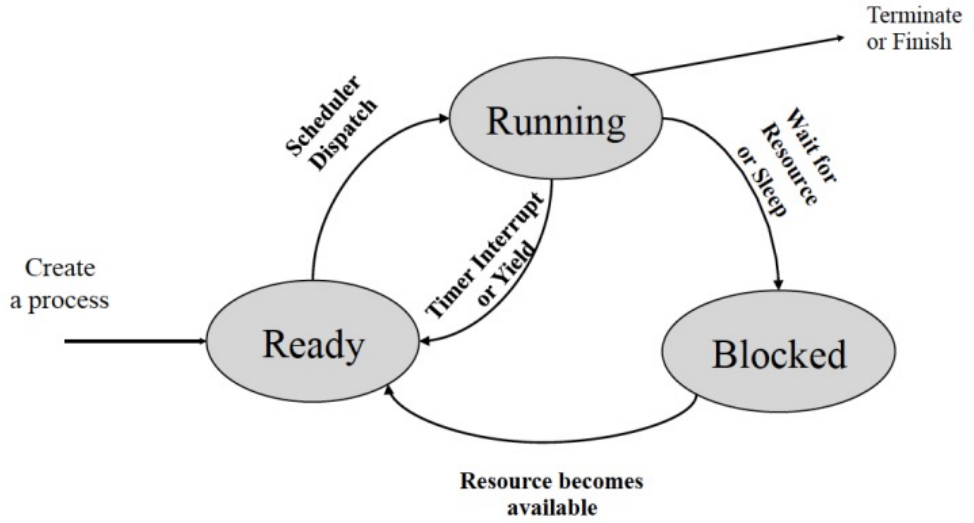
- **Goal: Implement the improved thread system in pintos**
  - Alarm clock
  - Priority scheduling
  - Advanced scheduler

- **Documents**
  - Project 1.docx - Requirements of project 1, brief explanation and requirements
  - Section 2, A.2, A.3, in pintos homepage (more detailed)
    (http://web.stanford.edu/class/cs140/projects/pintos/pintos.html)
    - Detailed explanation of project 1 and preliminaries
    - Description of thread system (A.2), synchronization (A.3), 4BSD scheduler (Appendix B)

Background: Thread



Process State Transition



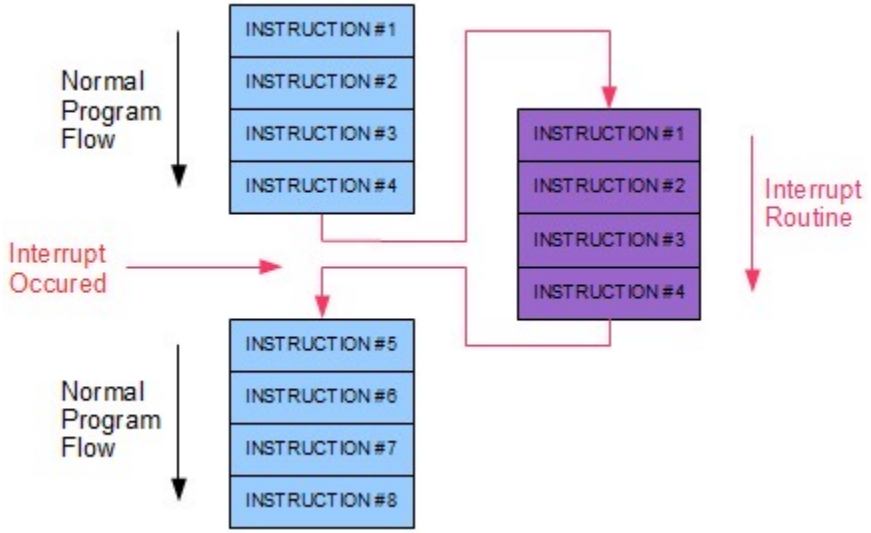
Running : executing now
Ready : waiting for CPU (everything is ready except CPU)
Blocked : waiting for I/O completion or Lock release

Background: Thread

- **Understanding threads**
  - Pintos already implements...
    - thread creation
    - thread completion
    - a simple scheduler to switch between threads (Round-Robin)
    - synchronized primitives (semaphores, locks, ...)
  - When a thread is created, you are creating a new context to be scheduled
    - provide function to be run in this context as an arguments to thread_create(...)
    - the given function acts like main() in that context
  - In pintos, each thread is assigned a small, fixed-size execution stack just under 4KB in size
    - You may cause problems, if you declare large data structures!

Background: Interrupt

- **See A.4 Interrupt handling**
- **Internal (synchronous) interrupts, that is, interrupts caused directly by CPU instructions.**
  - system calls, page faults, and divide-by-zero, and so on.
- **External (asynchronous) interrupts, that is, interrupts originating outside the CPU.**
  - timer, keyboard, serial ports, and so on.
  - we are interested in timer interrupts.



Background: Interrupt

```
33 /* Sets up the timer to interrupt TIMER_FREQ times per second,
34    and registers the corresponding interrupt. */
35 void
36 timer_init (void)
37 {
38   pit_configure_channel (0, 2, TIMER_FREQ);
39   intr_register_ext (0x20, timer_interrupt, "8254 Timer");
40 }
```

devices/timer.c

timer_interrupt handler function is registered as an external interrupt handler

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

```
177 /* Registers external interrupt VEC_NO to invoke HANDLER, which
178    is named NAME for debugging purposes.  The handler will
179    execute with interrupts disabled. */
180 void
181 intr_register_ext (uint8_t vec_no, intr_handler_func *handler,
182                    const char *name)
183 {
184   ASSERT (vec_no >= 0x20 && vec_no <= 0x2f);
185   register_handler (vec_no, 0, INTR_OFF, handler, name);
186 }
```

```
344 void
345 intr_handler (struct intr_frame *frame)
346 {
347   bool external;
348   intr_handler_func *handler;
349
```

```
364   /* Invoke the interrupt's handler. */
365   handler = intr_handlers[frame->vec_no];
366   if (handler != NULL)
367     handler (frame);
```

```
386       if (yield_on_return)
387         thread_yield ();
388     }
389 }
```

```
40 /* External interrupts are those generated by devices outside the
41    CPU, such as the timer.  External interrupts run with
42    interrupts turned off, so they never nest, nor are they ever
43    pre-empted.  Handlers for external interrupts also may not
44    sleep, although they may invoke intr_yield_on_return() to
45    request that a new process be scheduled just before the
46    interrupt returns. */
47 static bool in_external_intr;   /* Are we processing an external interrupt? */
48 static bool yield_on_return;    /* Should we yield on interrupt return? */
```

threads/interrupt.c

Background: Thread and Timer

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

devices/timer.c

timer_interrupt will be called externally per every tick and increases “ticks” by 1.

```
53 /* Scheduling. */
54 #define TIME_SLICE 4            /* # of timer ticks to give each thread. */
55 static unsigned thread_ticks;   /* # of timer ticks since last yield. */
```

```
120 /* Called by the timer interrupt handler at each timer tick.
121    Thus, this function runs in an external interrupt context. */
122 void
123 thread_tick (void)
124 {
125   struct thread *t = thread_current ();
126
127   /* Update statistics. */
128   if (t == idle_thread)
129     idle_ticks++;
130 #ifdef USERPROG
131   else if (t->pagedir != NULL)
132     user_ticks++;
133 #endif
134   else
135     kernel_ticks++;
136
137   /* Enforce preemption. */
138   if (++thread_ticks >= TIME_SLICE)
139     intr_yield_on_return ();
140 }
```

threads/thread.c

thread_tick increases “thread_ticks” and if “thread_ticks” exceeds TIME_SLICE, then yield.

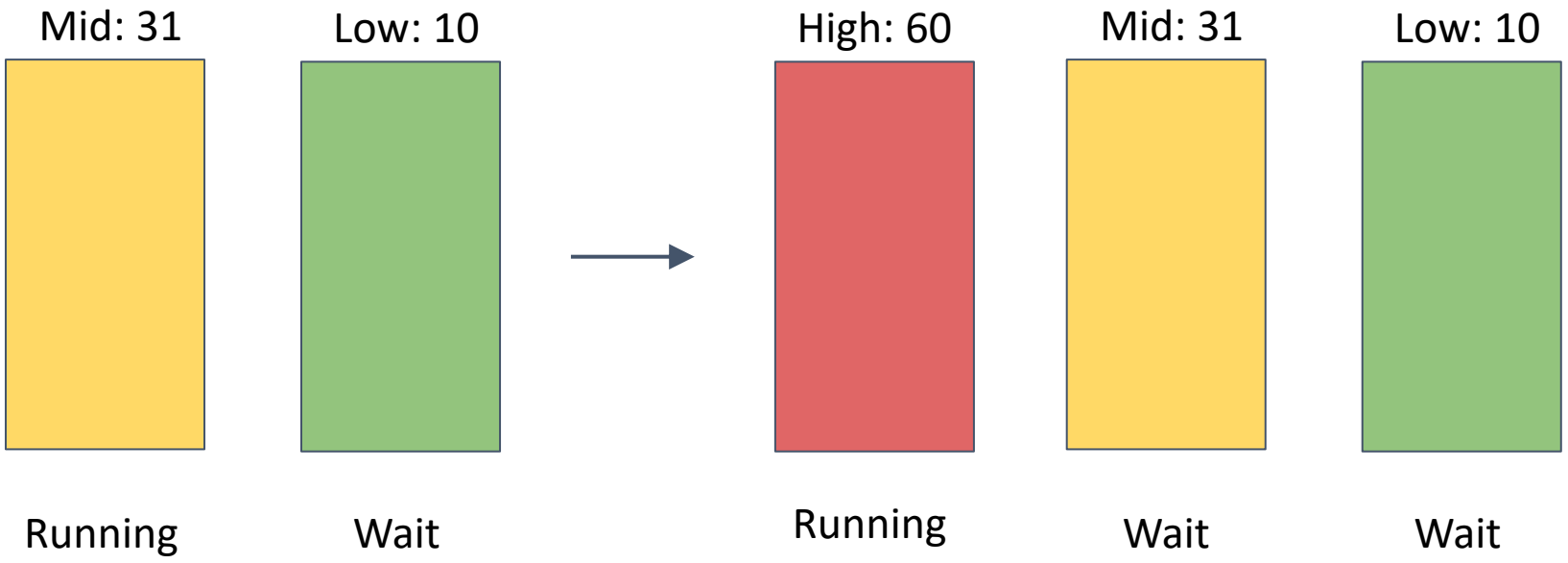
Alarm clock

- **Re-implement timer_sleep(x) in "devices/timer.c"**
  - Functionality
    - suspends execution of the calling thread for x timer ticks
  - Current Implementation: busy waits
    - spins in a loop checking the current time and calling thread_yield() until x timer ticks have gone by
    - thread_yield() locates the thread at the last of ready queue (ready_list in "threads/thread.c")
  - New implementation
    - Unless the system is idle, put the thread at the ready queue after they have waited for the right amount of time

Priority scheduling

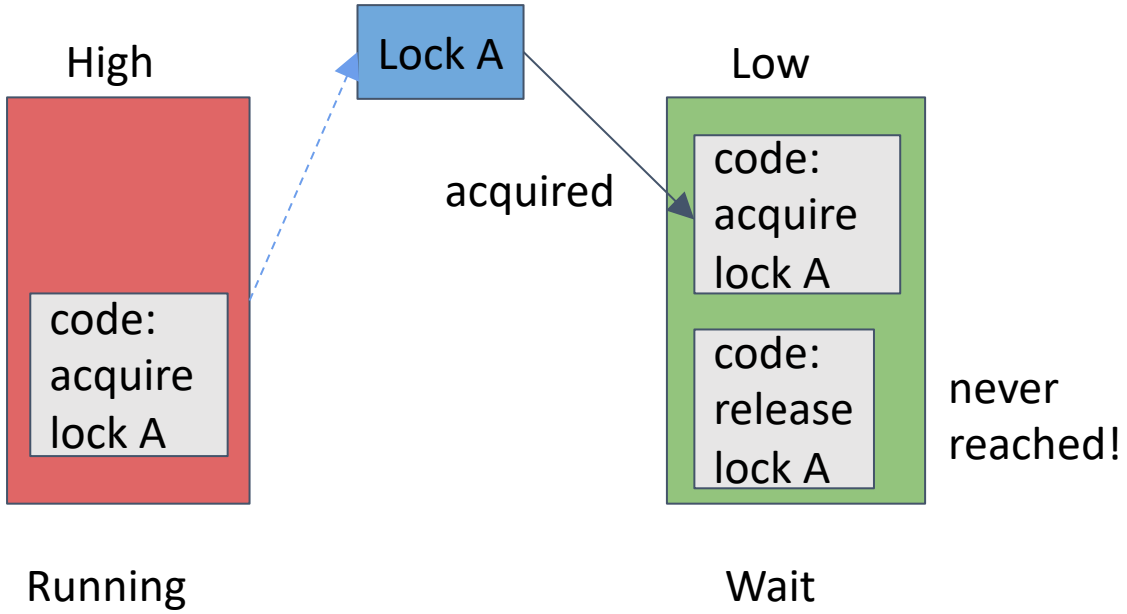
- **Modify the current round-robin scheduler to *priority* scheduler.**
- **Guidelines**
  - Among threads that are **in the ready list**, the highest-priority thread should be **scheduled to execute first**.
  - When threads are **waiting for a lock, semaphore, or condition variables**, the thread that has the highest priority should be **unblocked first**.
  - When a higher-priority thread than the currently running thread is **added to the ready list**, the current thread must **immediately yield** the processor.
  - A thread may raise or lower its own priority at any time, but lowering its priority such that it **no longer has the highest priority** must cause it to **immediately yield** the CPU.
    - **void** thread_set_priority(**int** new_priority)
    - **int** thread_get_priority(**void**)

Priority



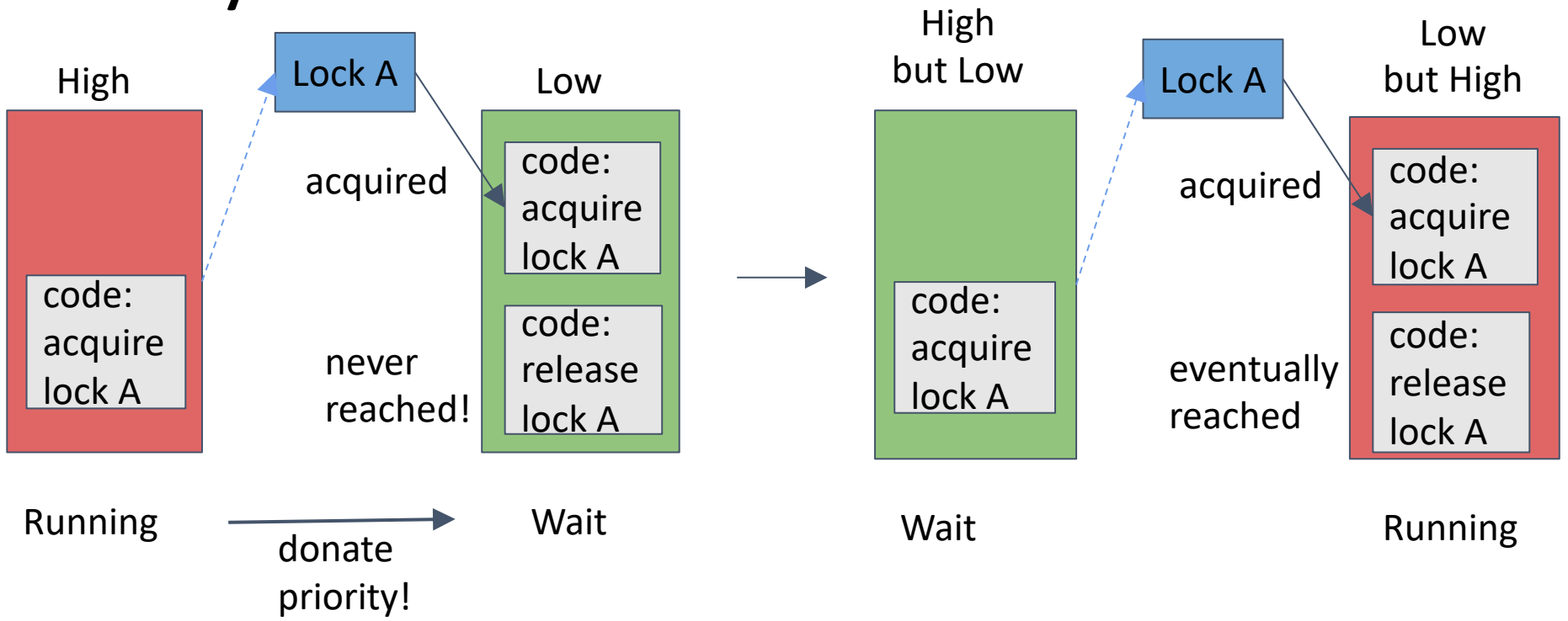
- Implementing the aforementioned four conditions cause priority inversion problems.

Priority inversion



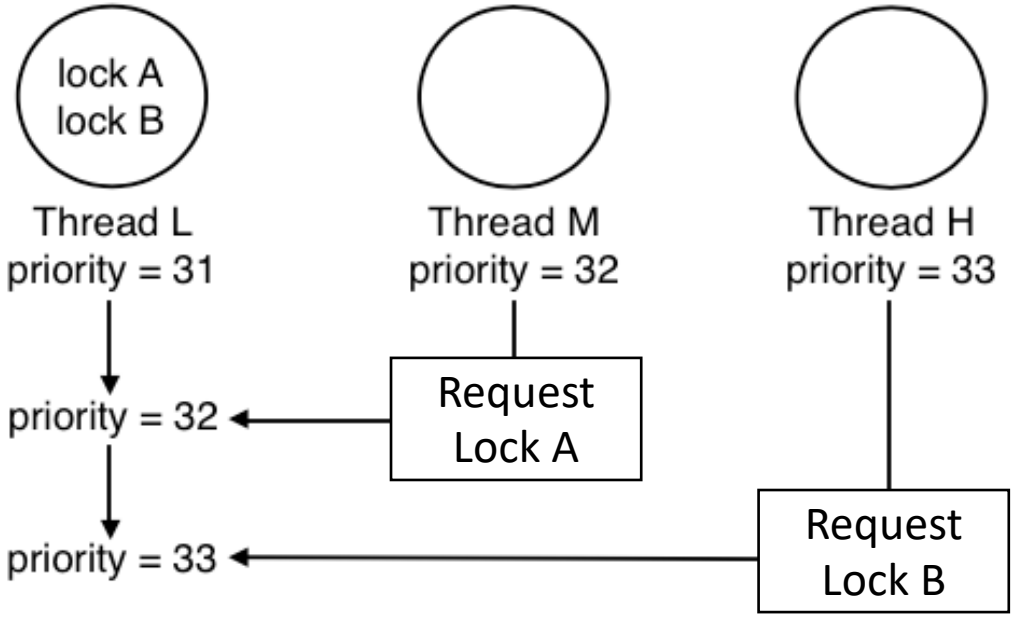
- If a thread needs to wait for thread who has lower priority, then waiting thread will never get the CPU because the thread of lower priority will not give any CPU time!

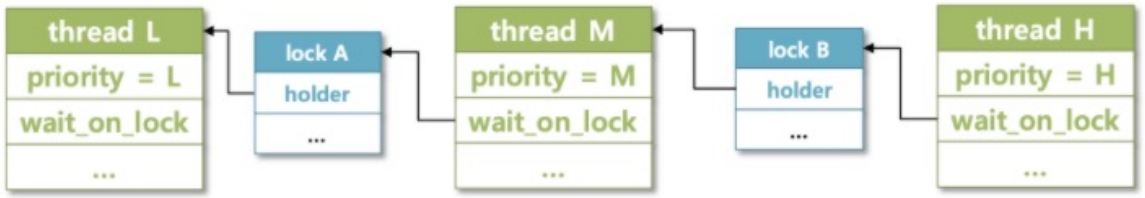
Priority donation



- You will need to account for all different situations
  - multiple donations
  - nested donations
- Implement priority donation for locks (not for semaphore, condition variable)

Multiple & Nested donation





Advanced scheduler

- **Implement multilevel feedback queue scheduler (MLFQS) similar to 4BSD**
  - by using ready queues per each priority, MLFQS manages the priority automatically
  - refresh the thread's priority per 4 ticks
  - MLFQS is selected by adding "-mlfqs" when running pintos
  - use the boolean variable thread_mlfqs in "threads/thread.c"

- **Implement priority scheduling in MLFQS**
  - no priority donation
  - thread_set_priority() should not modify a priority
    - thread_set_nice() can modify the priority indirectly
    - priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)

Development Suggestion

- **Read and design before implementation**
  - pintos web page will help you a lot
    - not only project 1 page, but also general introduction
  - rash implementation will cause you to initialized the project

- **Many groups divided the assignments into pieces, then combine at the end**
  - We do not recommend this approach because
    - often two changes conflict with each other ☹
    - requires lots of last-minute debugging ☹
  - Integrating your team's changes early and often
    - use git ☺
    - discuss often ☺

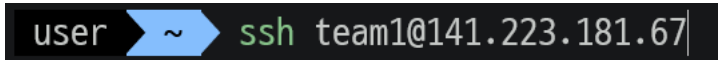
Design report should include

- **How to achieve each requirement**
  - overall structure
  - data structures, functions to be added / modified
  - algorithms, rationales, and so on.

- **Analysis of the current thread system**
  - structure, functions (thread_init(), ...), how to switch threads
  - see source codes ("threads/thread.h", "threads/thread.c", …) and section A.2

- **Analysis of synchronization**
  - meaning of semaphore, lock, and their implementation in pintos
  - see source codes ("threads/synch.h", "threads/synch.c", …) and section A.3

Submitting Project1

- **Server Information**
  - Server IP : 141.223.181.67
  - SSH account & password: “teamXX”. (XX is your team id on team_board)
  - Ex) ssh teamXX@141.223.181.67 

  - To prevent any confusion, it is recommended to change the password.

- **Whole project source code must be submitted to server**
  - Submit your entire project files at “/home/teamXX/pintos”.
  - Must include “.git” folder in the proje 

  - Ex) git clone “your git source” pintos

Submitting Project1 (cont.)

- **Git branch naming**
  - Project implementation must be submitted under “project1” branch.
  - Make sure all your features are merged in “project1” before submission.

```
team1@cse-edu ~/pintos  project1 git branch
    feature/interrupt
    feature/priority
    master
  * project1
```

- **Submission Due**
  - ~ 2023.10.09 23:59:59
  - After the deadline, all connected sessions will be terminated, and access will be denied

Announcements

- **Q&A**
  - You can use Q&A in PLMS (general rule)
  - Or email me (for private questions only): minhyeonoh@postech.ac.kr
- **Using GIT is mandatory.**
  - Your later submission of final project must include .git directory.
  - Recommended for collaboration.
  - Mandatory to ensure an even distribution of team members' contributions.
- **We can support additional servers for your projects**
  - One VM (1core / 1GB memory) can be provided per team who is not able to use their own desktop or laptop.
  - Please contact freely to TA (donghyeonryu@postech.ac.kr)
- **We will use PLMS for any additional announcements**

Supplementary

Suggested Questions

- How threads are layed out in the memory?
- How is the thread system initialized?
  - What thread_init() does?
- How to create a thread?
  - On the call to thread_create(), what happenes?
- On demand of switching the execution context, thread_switch() should be called.
  - In which concrete cases, is it called?
  - What it does?
  - You may want to consult switch.S and stack representation before and after the call to thread _switch().
  - What is the role of kernel_thread_frame, switch_entry_frame, switch_threads_frame?

Suggested Questions

- When and how does the status of threads change?
- What happens if sema_down() is implemented with if statement instead of the cu rrent while loop?
- How semaphores and locks differ?
- What is fixed point arithmetic and why is it necessary?

Development Suggestion

- **You do NOT need to include analysis of list.h and list.c in the design report.**
  - But, just anaylizing and understanding is HIGHLY recommended.
  - Extensive use of the functions therein is expected.
    - For every projects.

Relying solely on the assignment document distributed via PLMS may make it difficult to achieve a perfect understanding.

Although it might seem time-consuming at first, analyzing the code step by step will be the fastest way to grasp it comprehensively in the end.