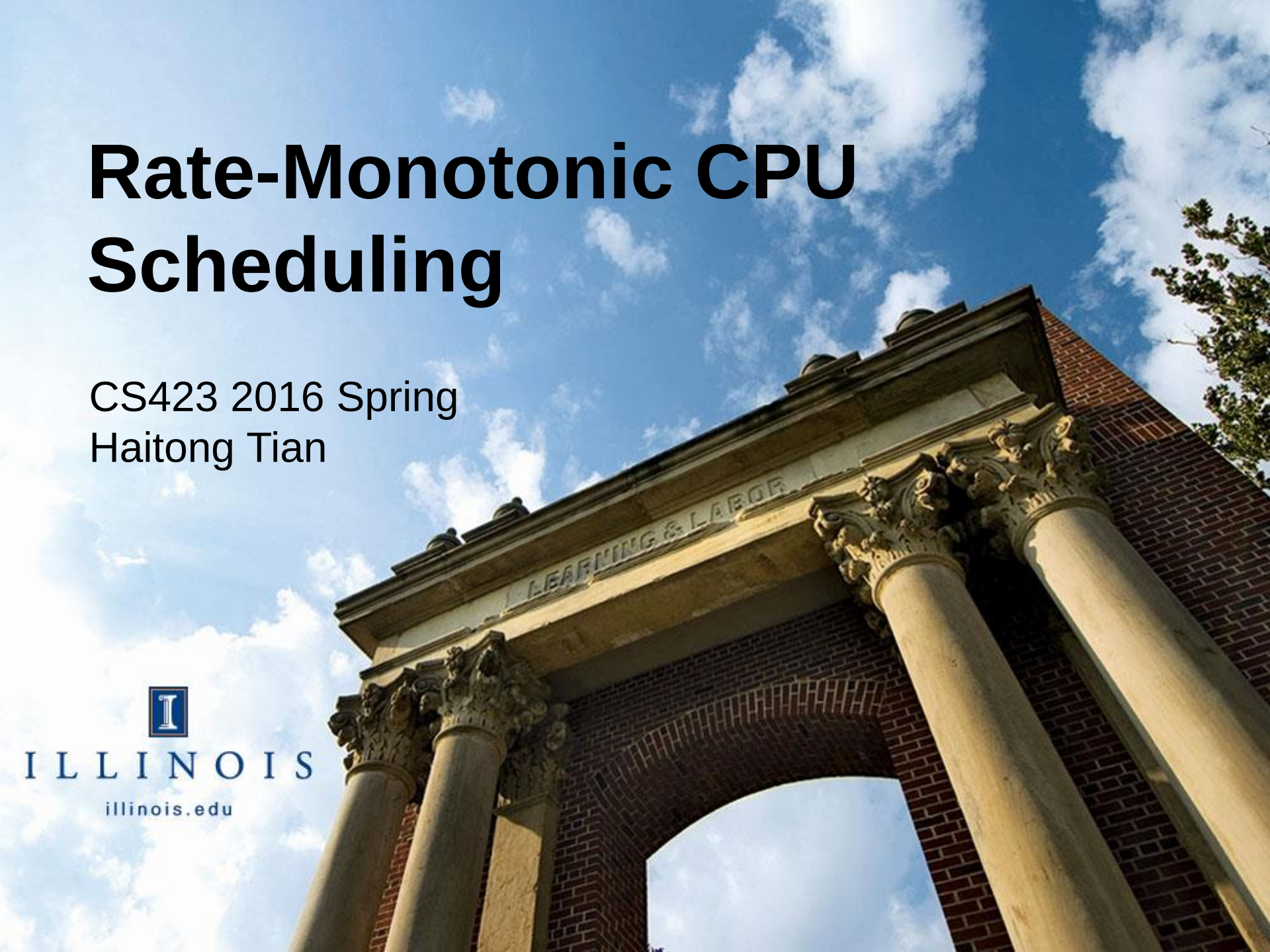
Rate-Monotonic CPU Scheduling

CS423 2016 Spring
Haitong Tian



ILLINOIS
illinois.edu

MP2: Rate-Monotonic CPU Scheduling

- Due 11:59pm March 9 2016
  - No late submission are accepted!
- Submit it on compass
  - Includes all your source code, Makefile, and report
  - Clearly list all your group members in the report
- Grading
  - Demo your code and answer questions
  - Signup sheet for MP demo
    - First come, first serve
    - We will make announcement when it is ready

Purpose of MP2

- **Understand** real time scheduling concepts
- **Design** a real time scheduler module in Linux kernel
- **Learn** how to use the kernel scheduling API, timer, proc file system
- **Test** your scheduler by implementing an user level application

MP2 & MP1

- MP1 is focused on getting familiar with Linux kernel programming (some of the data structures/proc file system/module install/uninstall etc.)
  - Code/Makefile from MP1 could be reused
- MP2 is aimed at developing useful kernel code, a Rate Monotonic Scheduler
  - Develop the scheduler as a Linux kernel module
  - Implement the admission control policy
  - Use Proc File System to communicate with user program
    - /proc/mp2/status

MP2: Rate Monotonic Scheduler

Real-time scheduling algorithm

- Static scheduling
  - Scheduler has complete knowledge of tasks
    - Task arrival time
    - Deadlines
    - Computation time
    - Future release
    - Precedence constraints
- Rate monotonic (RM) scheduling is the best solution

MP2: Rate Monotonic Scheduler

Real-time scheduling algorithm

- Dynamic scheduling
  - No complete knowledge of the tasks and their constraints
  - Arrival of tasks could be
    - Unknown duration
    - At unknown time
    - With unknown constraints
  - Scheduler can only schedule the current set of tasks
- Two types of dynamic scheduling
  - Resource sufficient environment
  - Resource insufficient environment

MP2: Rate Monotonic Scheduler

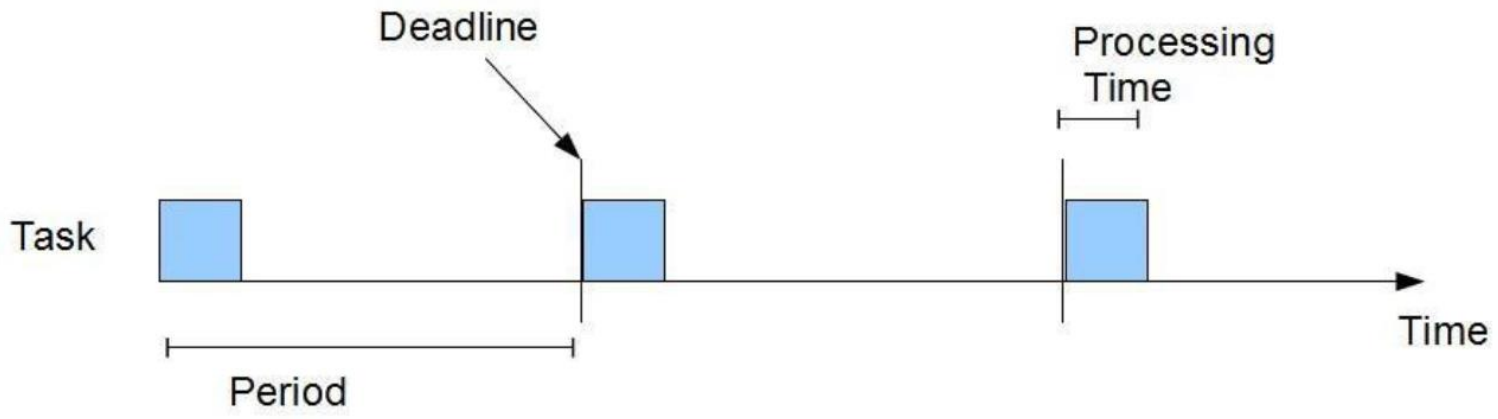
Real-time scheduling algorithm

- Resource sufficient environment
  - System resources are sufficient to a priori guarantee
  - New task might arrive at any time
  - Always enough resources exists to schedule them
  - Earliest deadline first (EDF) is the optimal solution
- Resource insufficient environment
  - No guarantee of the sufficiency of resources
  - Might have to delay/reject some tasks
  - Not suitable for hard real-time environment

MP2: Rate Monotonic Scheduler

Real-time scheduling assumptions

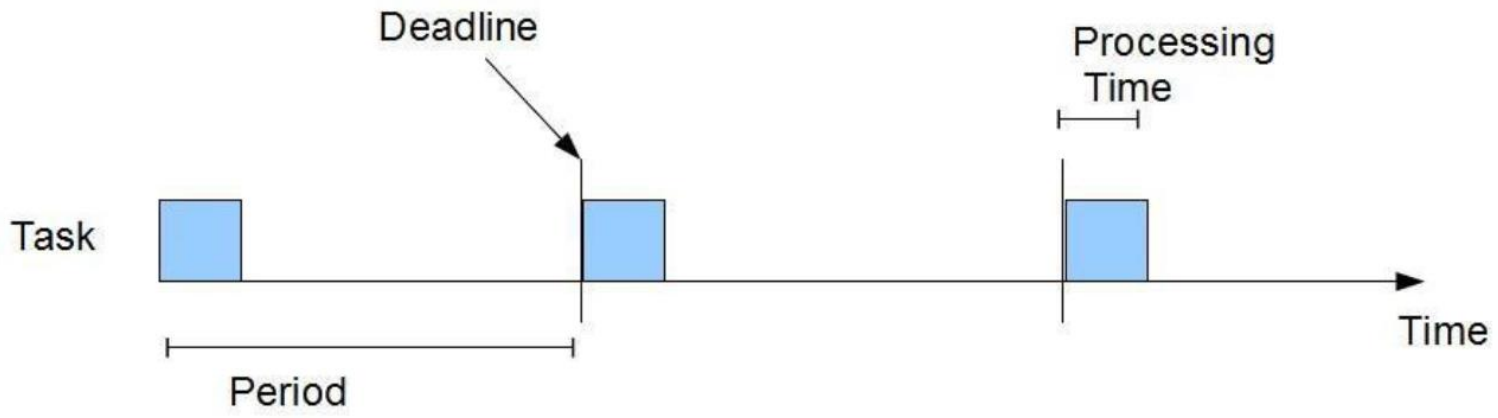
- Tasks are periodic. Constant interval between tasks.
- Each task mush be completed before the next one
- Tasks are independent
- Runtime of each task is constant



MP2: Rate Monotonic Scheduler

Rate monotonic scheduling (static scheduling)

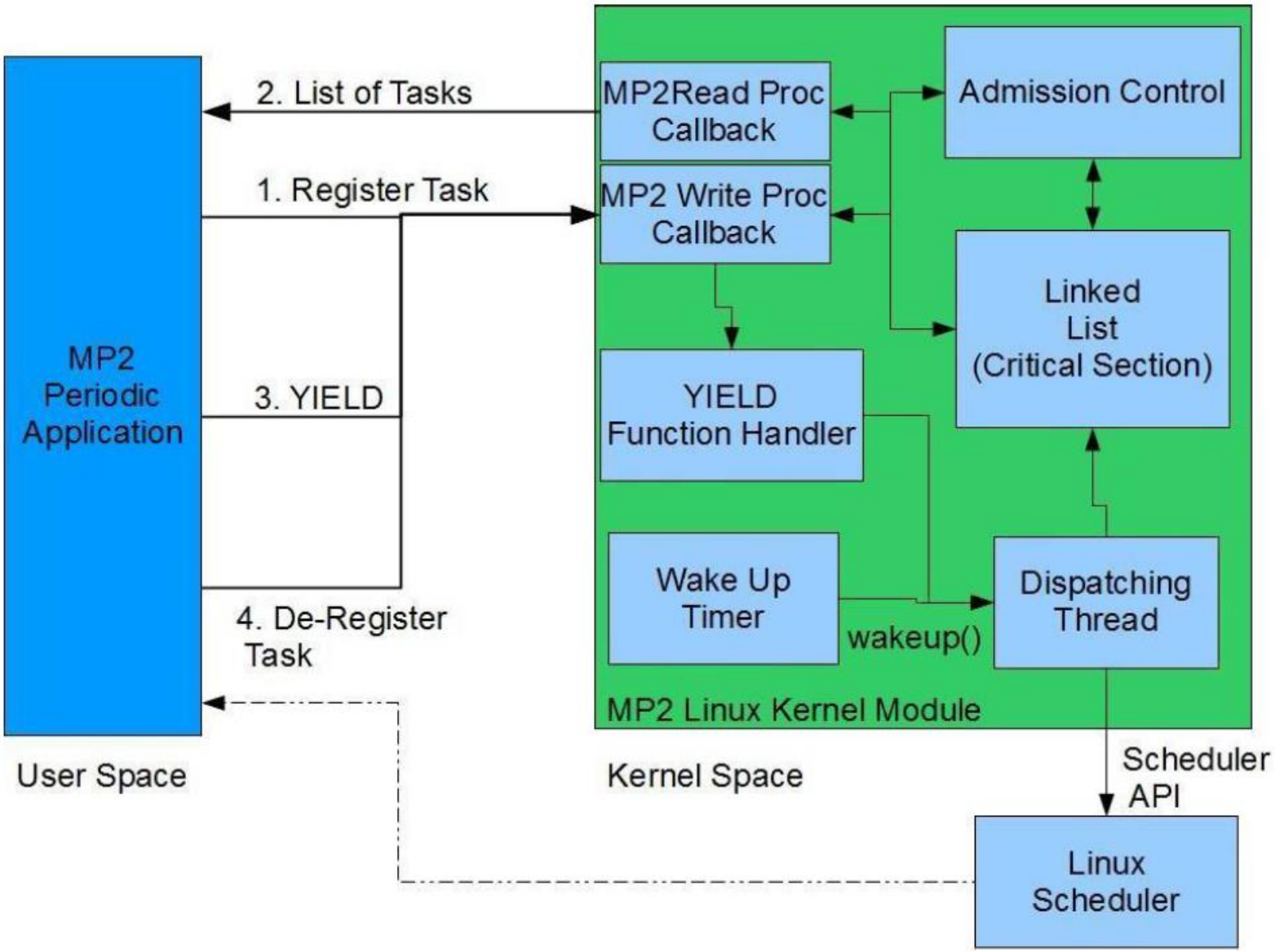
- Higher priority for tasks with smaller period.
  - Regardless of their execution time
- Always pick the task with the highest priority
  - Preemptive
- Priority does not change over time
- Admission control: $\sum_{i \in T} \frac{C_i}{P_i} \leq 0.693$



MP2 Overview

- Design a rate-monotonic scheduler with admission control policy
- Scheduler performs the following
  - Registration (define your own message type)
    - With info like PID, period, computation time
  - Yield
  - De-registration
- Scheduler only register a process if it passes admission control policy
- Simple test application

MP2 Overview



/proc file system

- Use /proc file system to communication between the user space program and the kernel module

- Process
  - Register: PID, period, computation time
  - Yield: finished current task. Sleep until next task arrives
  - De-register

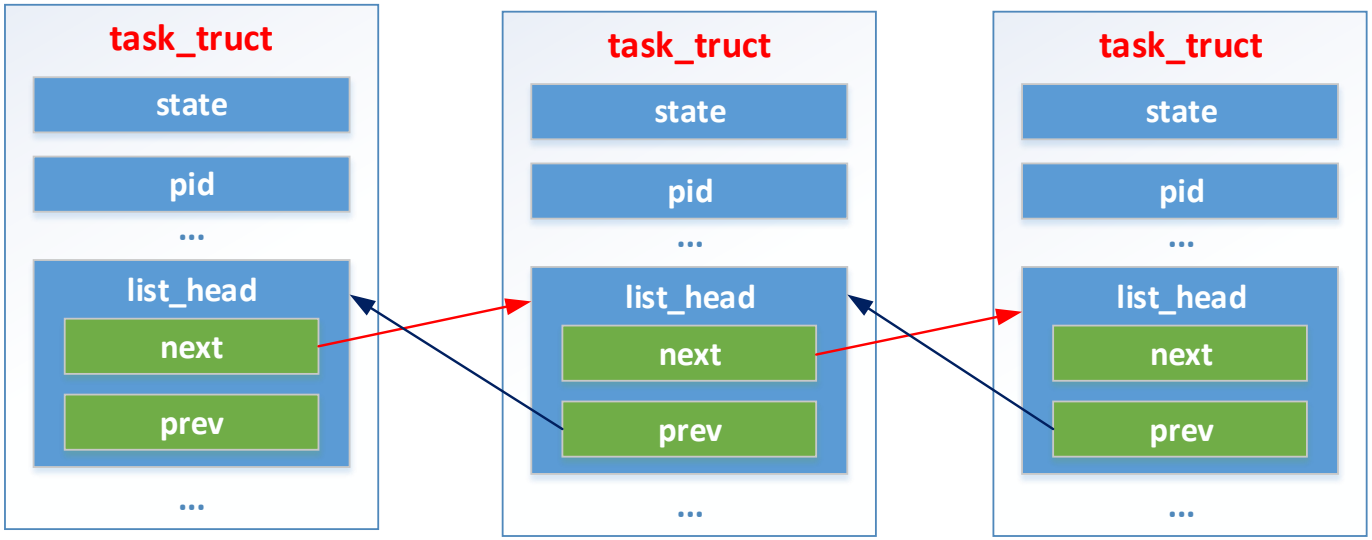
Context Switch

Utilize Linux scheduler API

- *schedule()*: trigger scheduling
  - Runs after changing the state of a process
- *wake_up_process()*
  - Can be called in interrupt context
- *sched_setscheduler()*: set scheduling parameters
  - FIFO, NORMAL, etc.
- *set_current_state()*: changes the state
  - Can be used to put a process to sleep
- *set_task_state()*: changes the task state

Process Control Block

- PCB is defined by task_struct structure
- PCB managed by a circular doubly linked list
- task_struct * current points to currently running process
- PCB contains process states
  - TASK_RUNNING/TASK_INTERRUPTIBLE/TASK_UNINTERRUPTIBLE…



Process Control Block

- Additional PCB info
  - priority: the static priority
  - rt_priority: the real-time priority
  - counter: the number of clock ticks left in the scheduling budget of the process
  - policy: the scheduling policy
    - SCHED_NORMAL: Normal
    - SCHED_FIFO or SCHED_RR
  - mm: the memory descriptor



Define Your Own PCB

MP2_task_struct

- struct task_struct * task
- int period
- int processing_time
- int task_state
- timer_list timer
- It's up to you!
- …

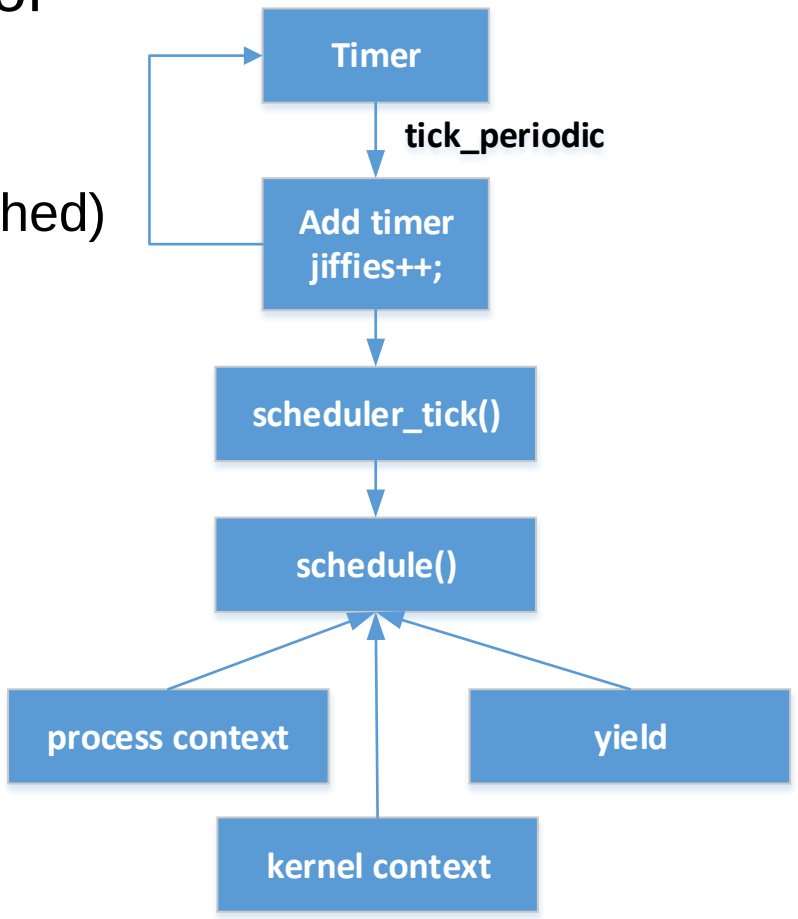
```
struct mp2_task_struct
{
    struct task_struct *task;
    struct list_head task_node;
    struct timer_list task_timer;

    unsigned int task_state;
    uint64_t next_period;
    unsigned int pid;
    unsigned long relative_period;
    unsigned long slice;
};
```

MP2 Scheduling

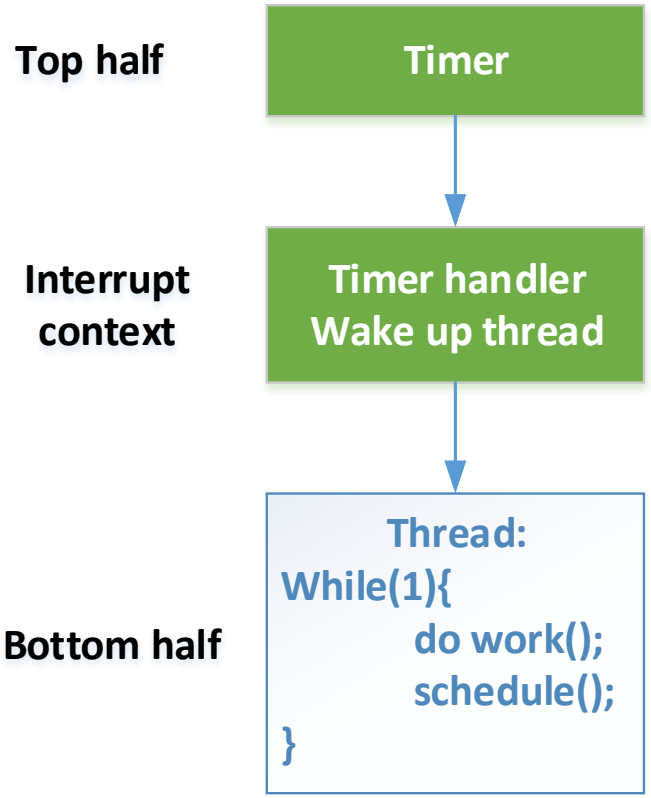
- schedule() is the entry point the of the scheduler
  - Scheduler_tick() if a reschedule is marked (task_struct => needs_resched)
  - Yield() called by application in uses space
  - Scheduler() called in kernel space

- Can schedule() be called inside interrupt handler?



Two Halves

- Kernel timers are used to create timed events
- They use jiffies to measure the system time
- Timers are interrupts
  - We can't sleep or hog CPU in them!
- Solution:
  - Divide the work into two parts
  - Uses the timer handler to signal a thread. (TOP HALF)
  - Let the kernel thread do the real job. (BOTTOM HALF)



Implementation

Yield handler

- Update timer;
- State = sleep;
- Wake up scheduler;
- Sleep;

scheduler

- Select highest priority task (smallest period)
- State = running
- Wake up process
- sleep

Timer interrupt

- State = ready
- Wake up scheduler

Locks

- Spinlocks inside timer interrupts
- *spin_lock_irqsave(spinlock_t *, unsigned)*
  - Disable interrupts: can safely implement spinlocks on single core machines
- *spin_unlock_irqrestore(spinlock_t *, unsigned int)*
- Mutex for linked list

Admission Control

$$\sum_{i \in T} \frac{C_i}{P_i} \leq 0.693$$

- Avoid floating point operations inside kernel

Test Application

```
void main (void)
{
    //Proc filesystem
    REGISTER(PID, Period, ProcessTime);
    //Proc filesystem: Verify the process was admitted
    list=READ STATUS();
    if (!process in the list) exit(1);

    YIELD(PID);      //Proc filesystem
    //this is the real-time loop
    while(exist jobs)
    {
        //wakeup_time=t0-gettimeofday() and factorial computation
        do_job();

        YIELD(PID); //Proc filesystem
    }
    UNREGISTER(PID); //Proc filesystem
}
```

Test Application

- Calculate the factorial of a fixed number
- Estimate approx. processing time
- Test with multiple applications
- Verify Pre-emption

Conclusions

- Linux uses Priority based Scheduling
- Priorities are adjusted based on the state of the process and its scheduling policy
- The PCB (task_struct) contains the information about the state of a process
- Rescheduling in Linux follows the Two-Halves approach. Reschedule can occur in process context or kernel context only.
- We can use the Scheduler API to build our own policy on top of the Linux scheduler.
- Our RMS scheduler will have a scheduler thread and will be invoked after a Job Release and after a Job Completion