CS 423
Operating System Design:
MP2 Walkthrough

Professor Adam Bates
Spring 2018

MP2: Rate-Monotonic Scheduling

- MP2 will be out at the end of the week
- We are currently grading MP1
- Reminder
  - Please do not touch your VMs until MP2 is out

A Note About Piazza

- “My code is not running, why?” is not very helpful
    - Be more specific when dealing with failures so we can help
- Use private posts if you are not comfortable sharing details of your implementation
    - Or office hours
- Be careful not to remove `/var/log/sssd` as this is will brick authentication

Purpose of MP2

- Understand real time scheduling concepts
- Design a real time schedule module in the Linux kernel
- Learn how to use the kernel scheduling API, timer, procfs
- Test your scheduler by implementation a user level application

Reuse of MP1

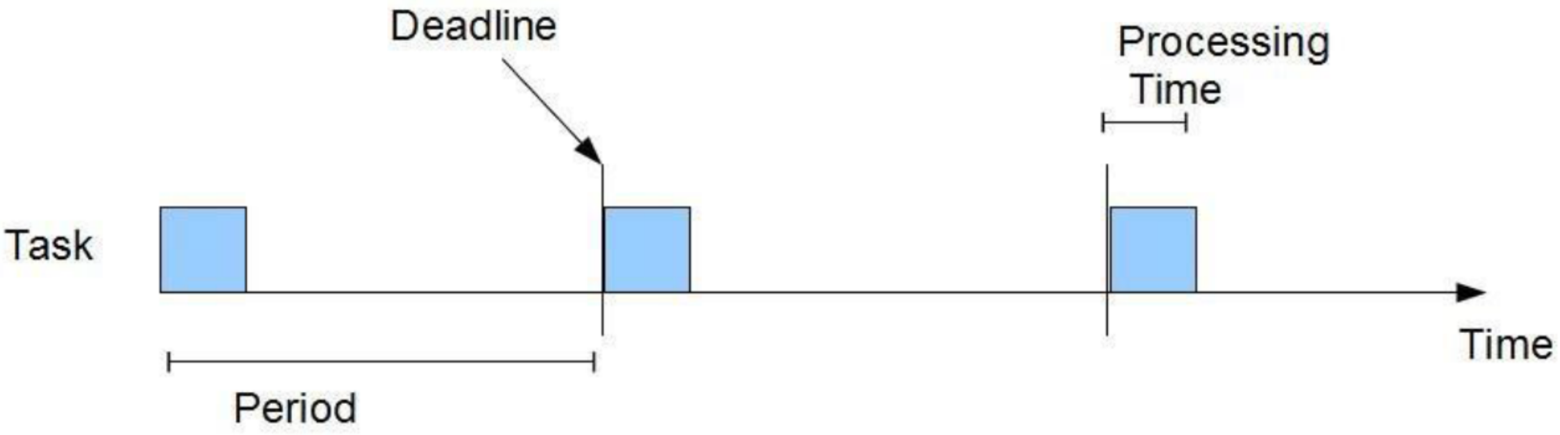
- MP1 was focused on getting you familiar with kernel programming
  - Code/Makefile from MP1 can be reused for MP2
- MP2 is aimed at developing useful kernel code
  - Develop a scheduler as a kernel module
  - Implement a task admission control policy
  - Use procfs to communicate with user programs

Introduction

- Real-time systems have requirements in terms of response time and predictability
  - Think video surveillance systems
- We will be dealing with periodic tasks
  - Constant period
  - Constant running time
- We will assume tasks are independent

Periodic Tasks Model

- Liu and Layland [1973] model, each task i has
  - Period $P\downarrow i$
  - Deadline $D\downarrow i$
  - Runtime $C\downarrow i$



Rate Monotonic Scheduler (RMS)

- A static scheduler has complete information about all the incoming tasks
  - Arrival time, deadline, runtime, etc.
- RMS assigns higher priority for tasks with higher rate/shorter period
  - It always picks the task with the highest priority
  - It is preemptive

Optimality of RMS

- RMS is optimal for hard-real time systems
- If RMS cannot schedule it, then no other algorithm can!
- If any other scheduler algorithm can scheduler a set of tasks, then RMS can do it too!

MP2 Overview

- We will implement RMS with an admission control policy as a kernel module
- The scheduler provides the following interface
  - Registration: save process info like pid, P, D, etc.
  - Yield: process notifies RMS that is has completed its period
  - De-Registration: process notifies RMS that it has completed all its tasks
- We will use procfs to communicate between the modules and the processes

Admission Control

- We only register a process if it passes admission control
- The module will answer this question every time
  - *Can the new set of processes still be a scheduled on a single processor ?*
  - Yes iff $$\boxed{\sum_{i \in T} \frac{C_i}{P_i} \leq 0.693}$$
- Assume always that $C_i < P_i$

Admission Control

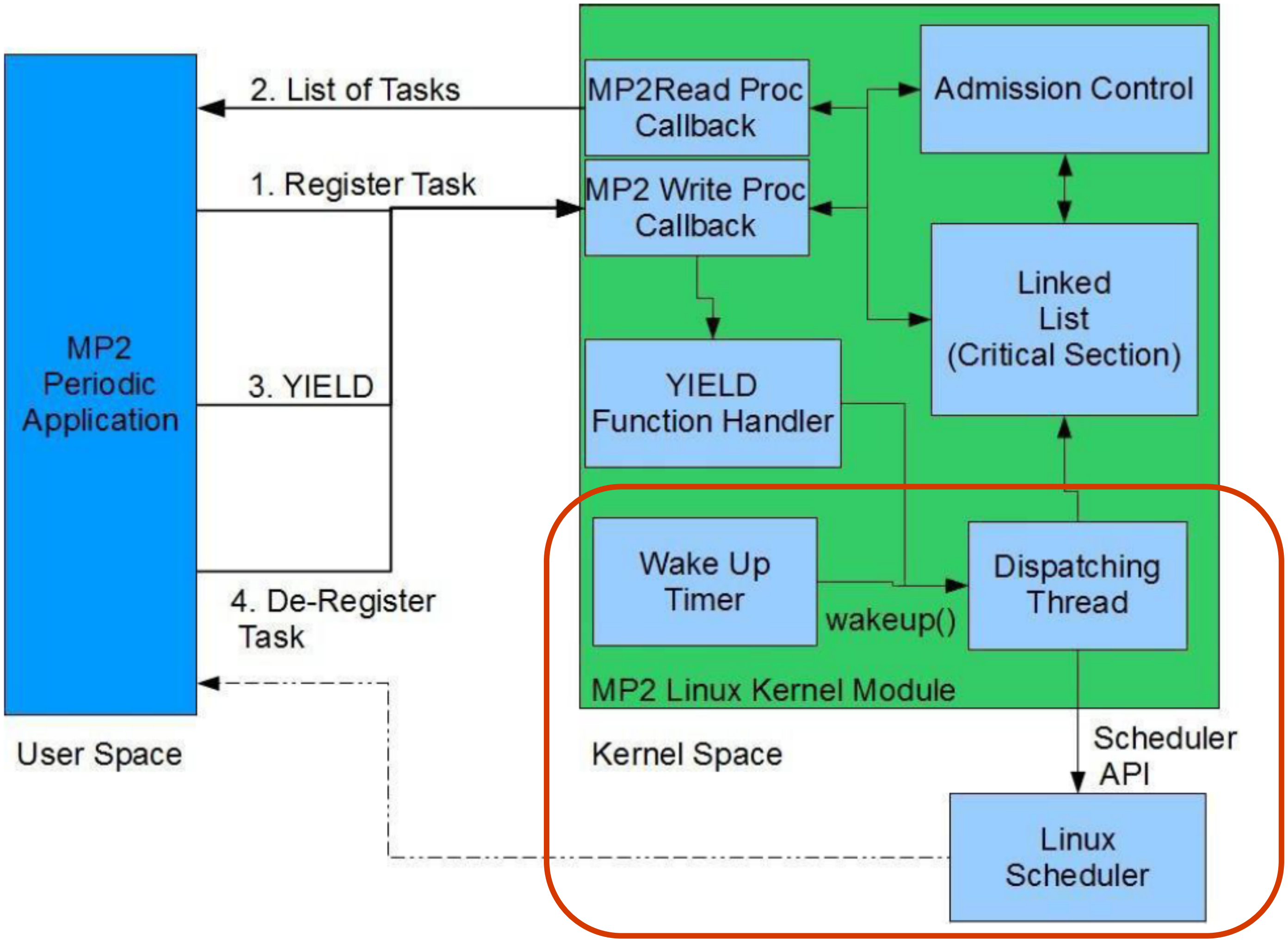
- We only register a process if it passes admission control
- The module will answer this question every time

Recall that floating point operations are very expensive in the kernel. You should NOT use them.

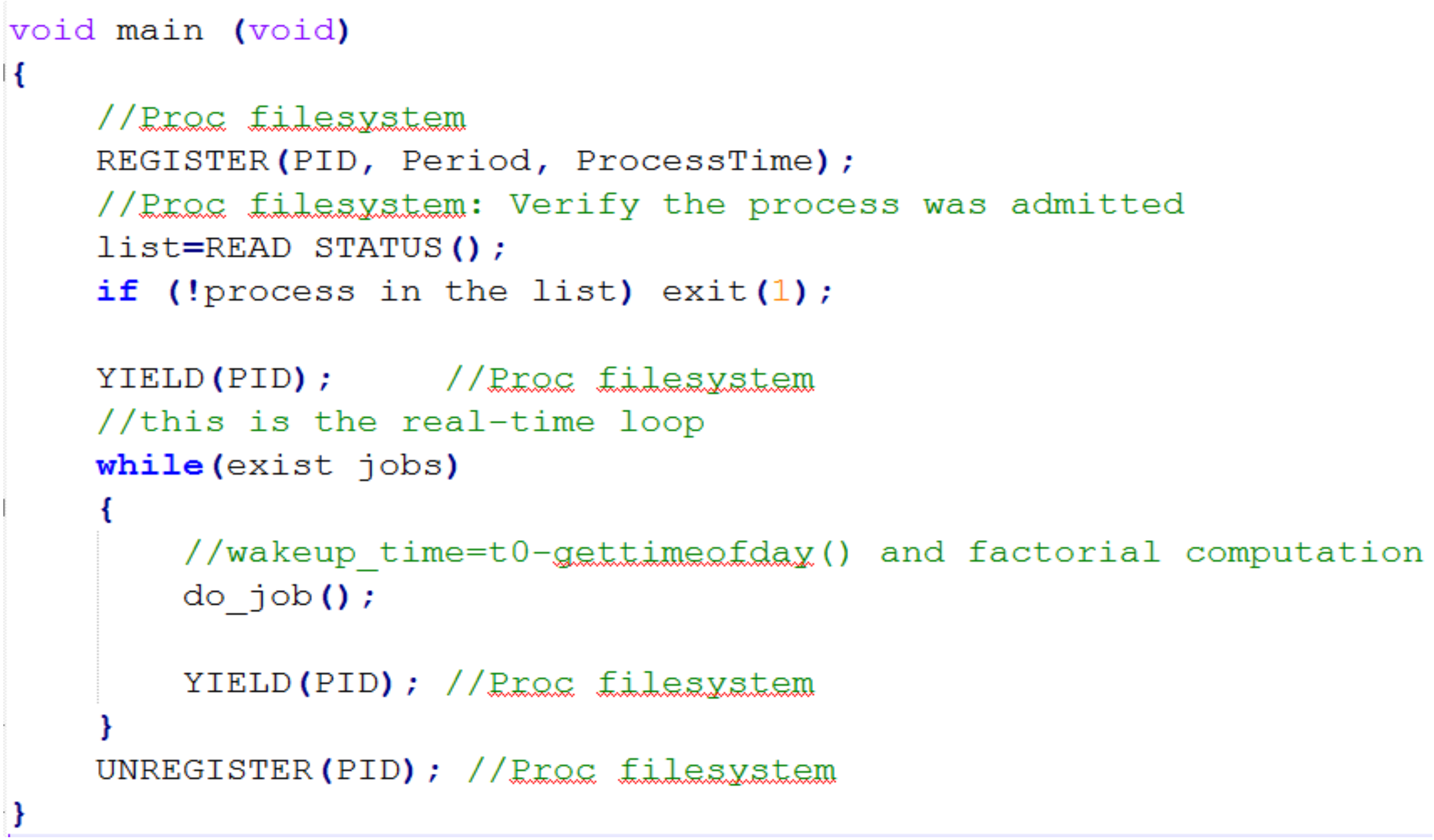
Instead use **Fixed-Point arithmetic**

- Assume always that $C_i < P_i$

MP2 Building Blocks

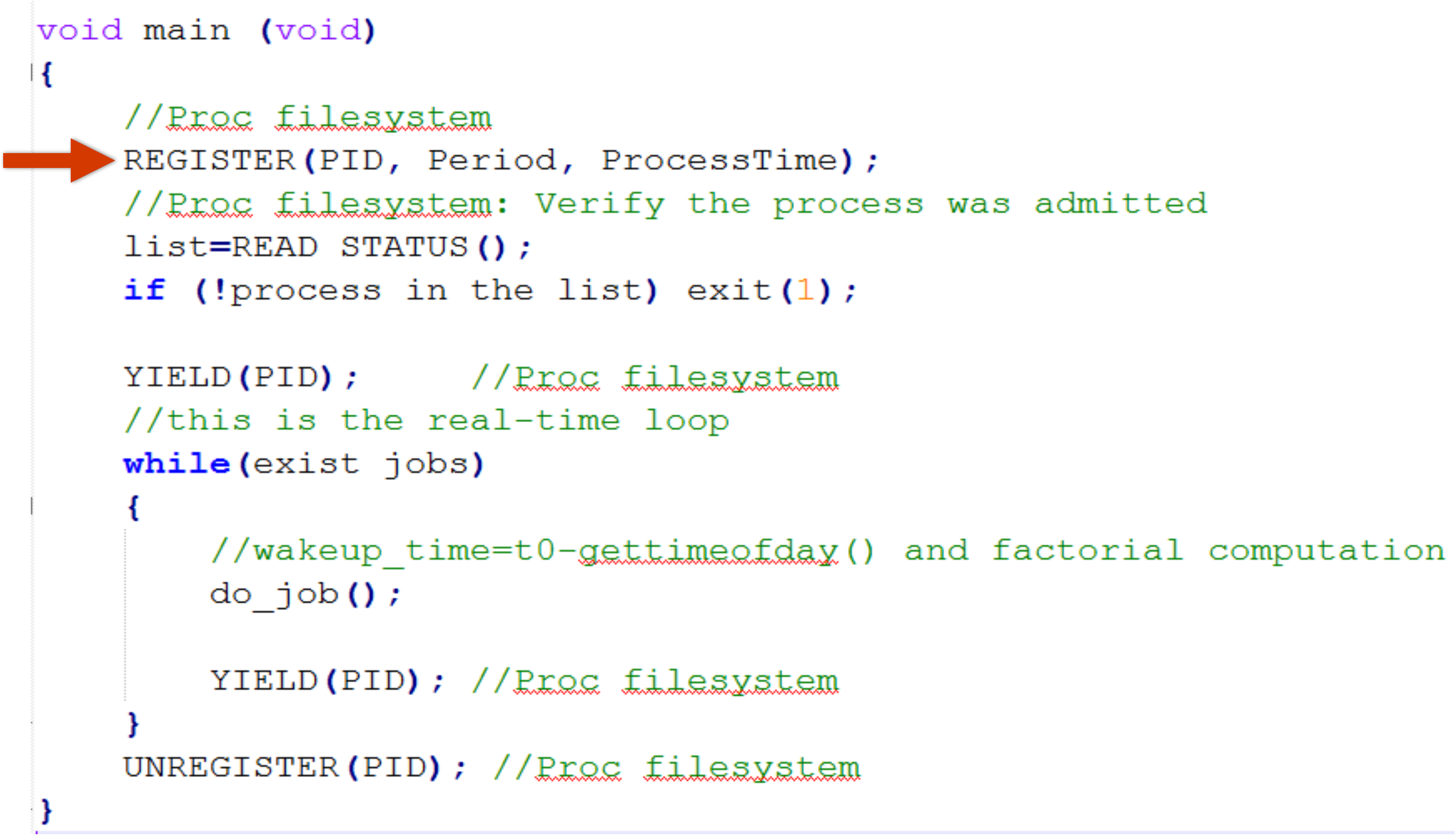


MP2 User Process Behavior



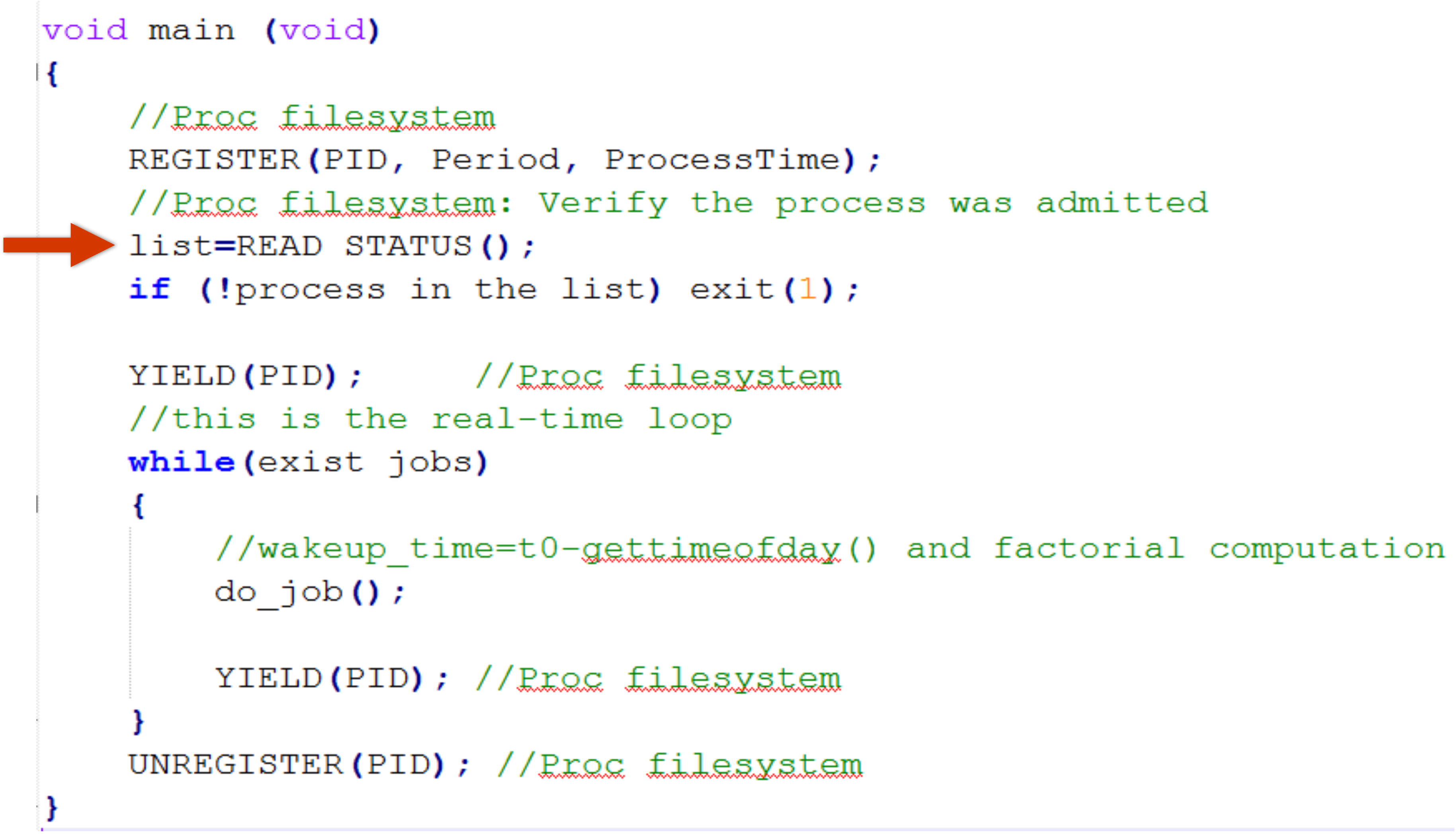
```
void main (void)
{
    //Proc filesystem
    REGISTER(PID, Period, ProcessTime);
    //Proc filesystem: Verify the process was admitted
    list=READ STATUS();
    if (!process in the list) exit(1);

    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
}
```

MP2 User Process Behavior



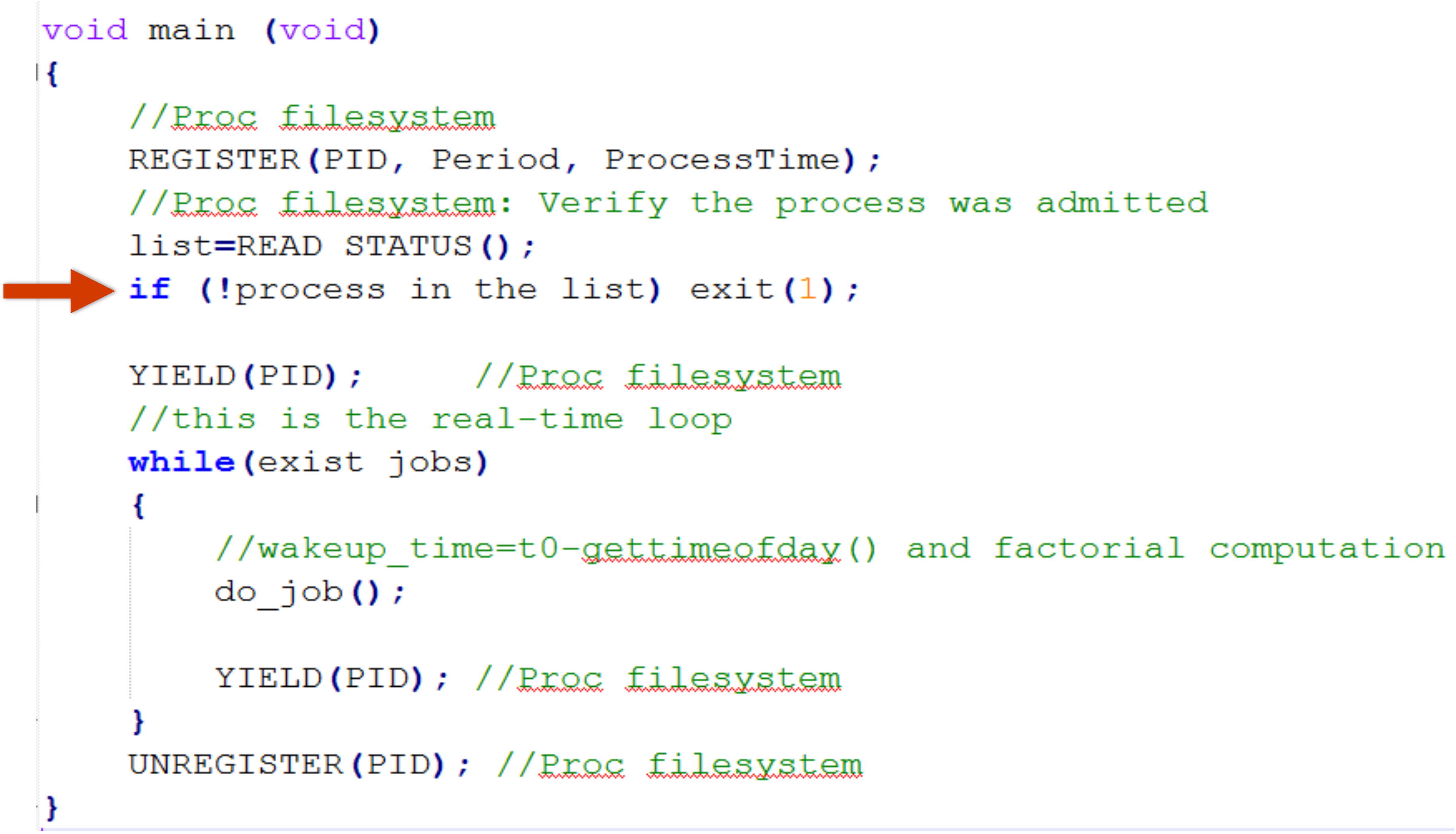
```
void main (void)
{
    //Proc filesystem
    REGISTER(PID, Period, ProcessTime);
    //Proc filesystem: Verify the process was admitted
    list=READ STATUS();
    if (!process in the list) exit(1);

    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
}
```

MP2 User Process Behavior



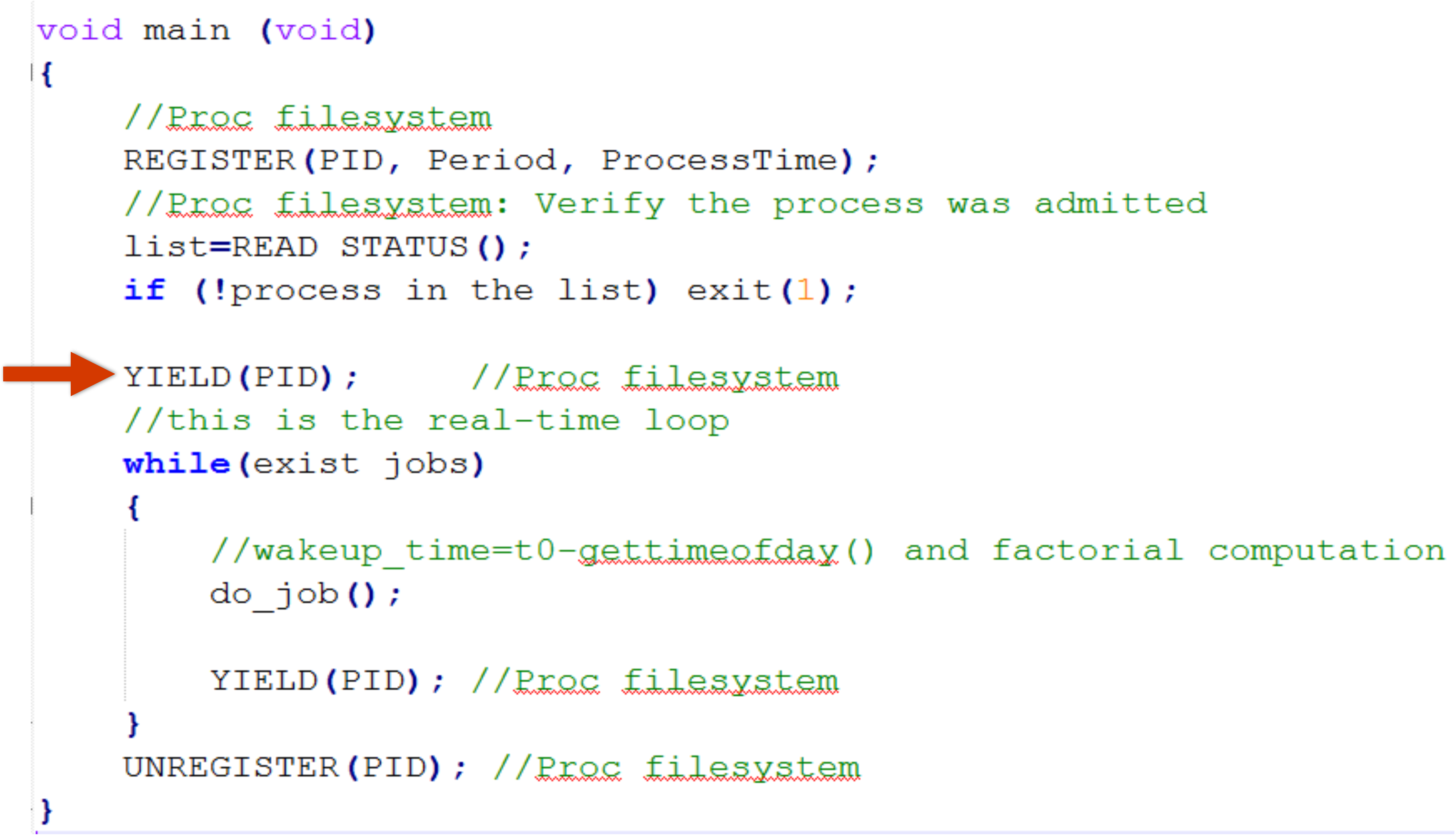
```
void main (void)
{
    //Proc filesystem
    REGISTER(PID, Period, ProcessTime);
    //Proc filesystem: Verify the process was admitted
    list=READ STATUS();
    if (!process in the list) exit(1);

    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
}
```

MP2 User Process Behavior



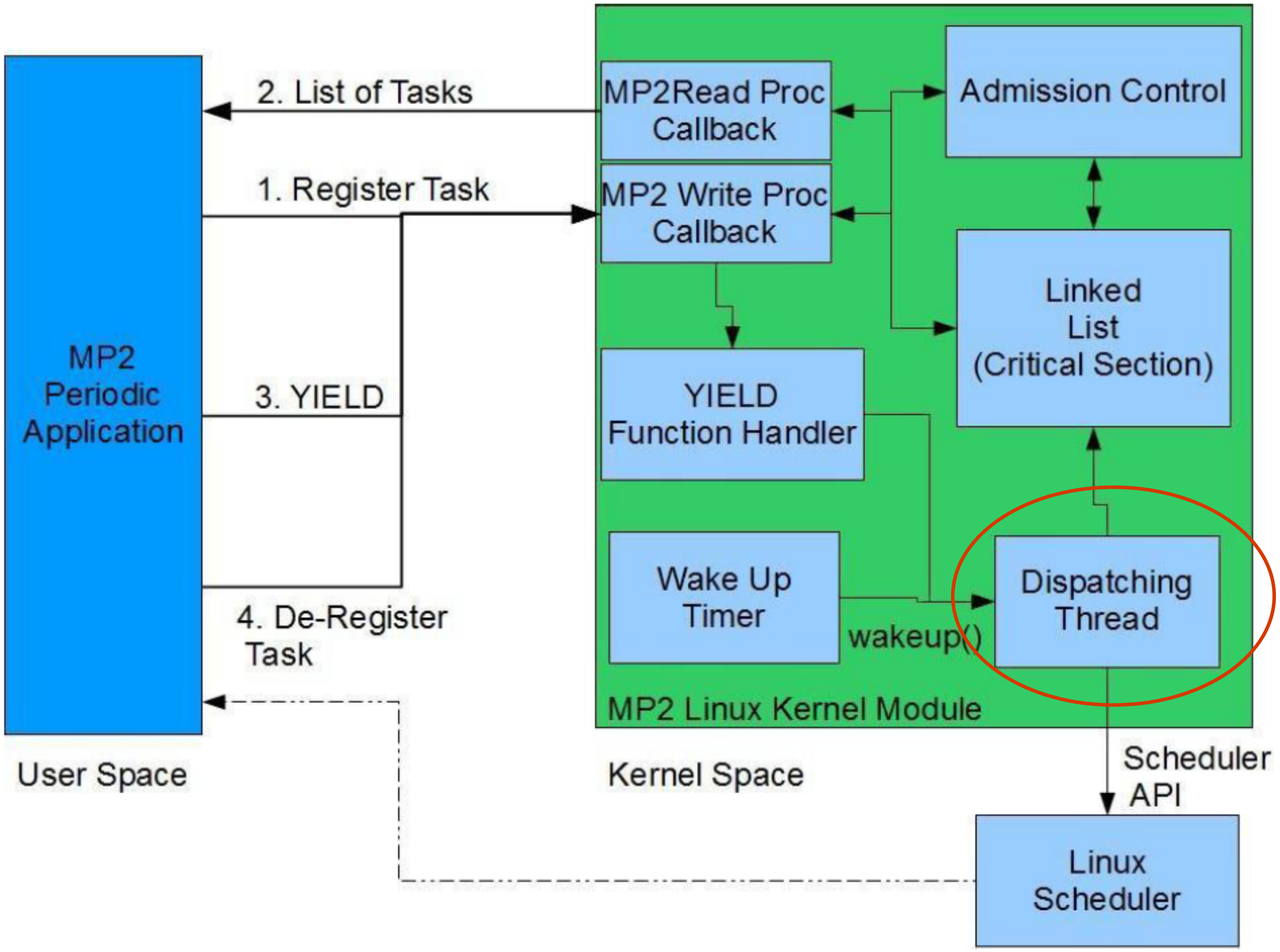
```
void main (void)
{
    //Proc filesystem
    REGISTER(PID, Period, ProcessTime);
    //Proc filesystem: Verify the process was admitted
    list=READ STATUS();
    if (!process in the list) exit(1);

    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
}
```

MP2 User Process Behavior

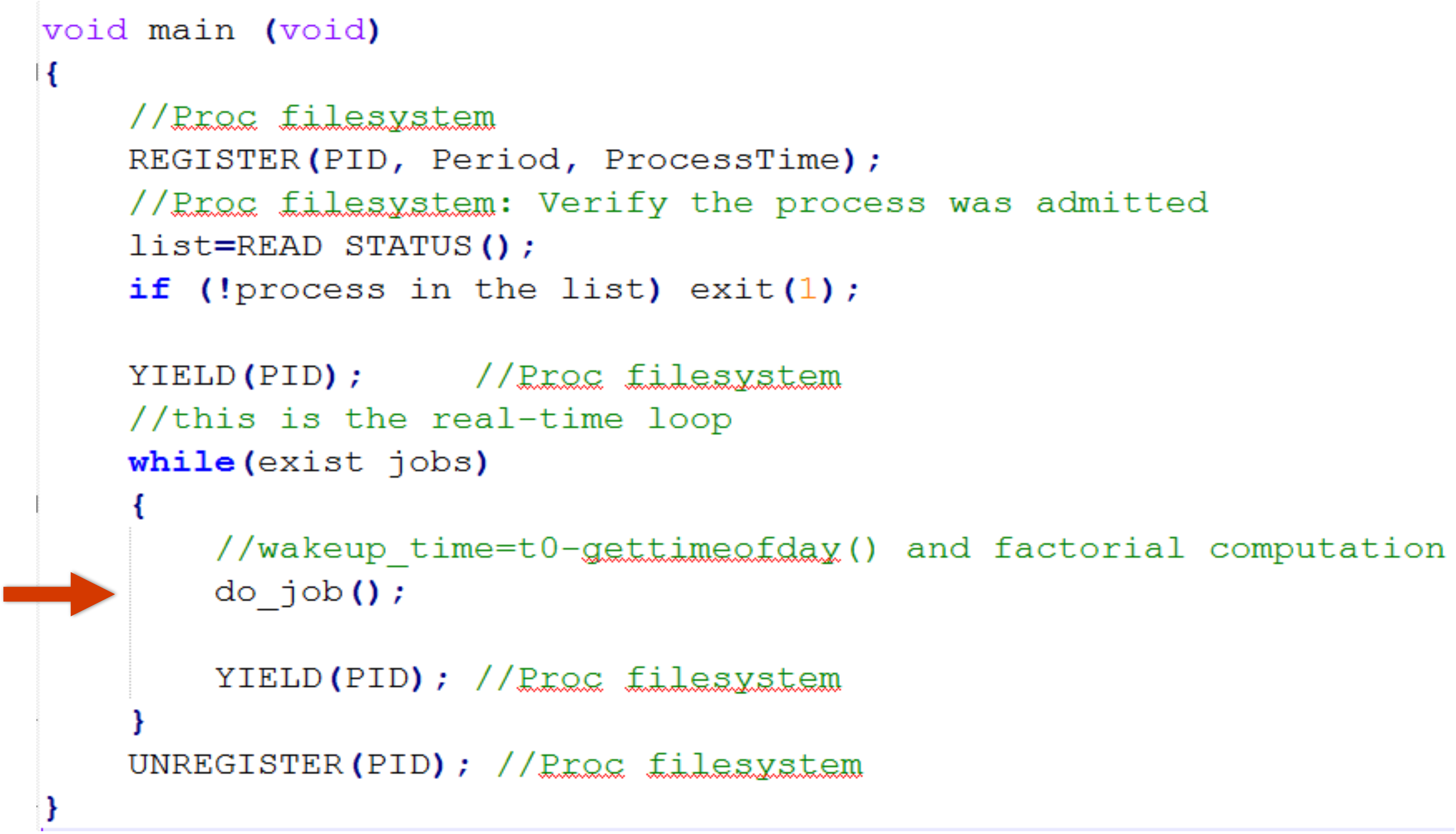


```
void main (void)
{
    //Proc filesystem
    REGISTER(PID, Period, ProcessTime);
    //Proc filesystem: Verify the process was admitted
    list=READ STATUS();
    if (!process in the list) exit(1);

    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
}
```

MP2 Dispatching Thread

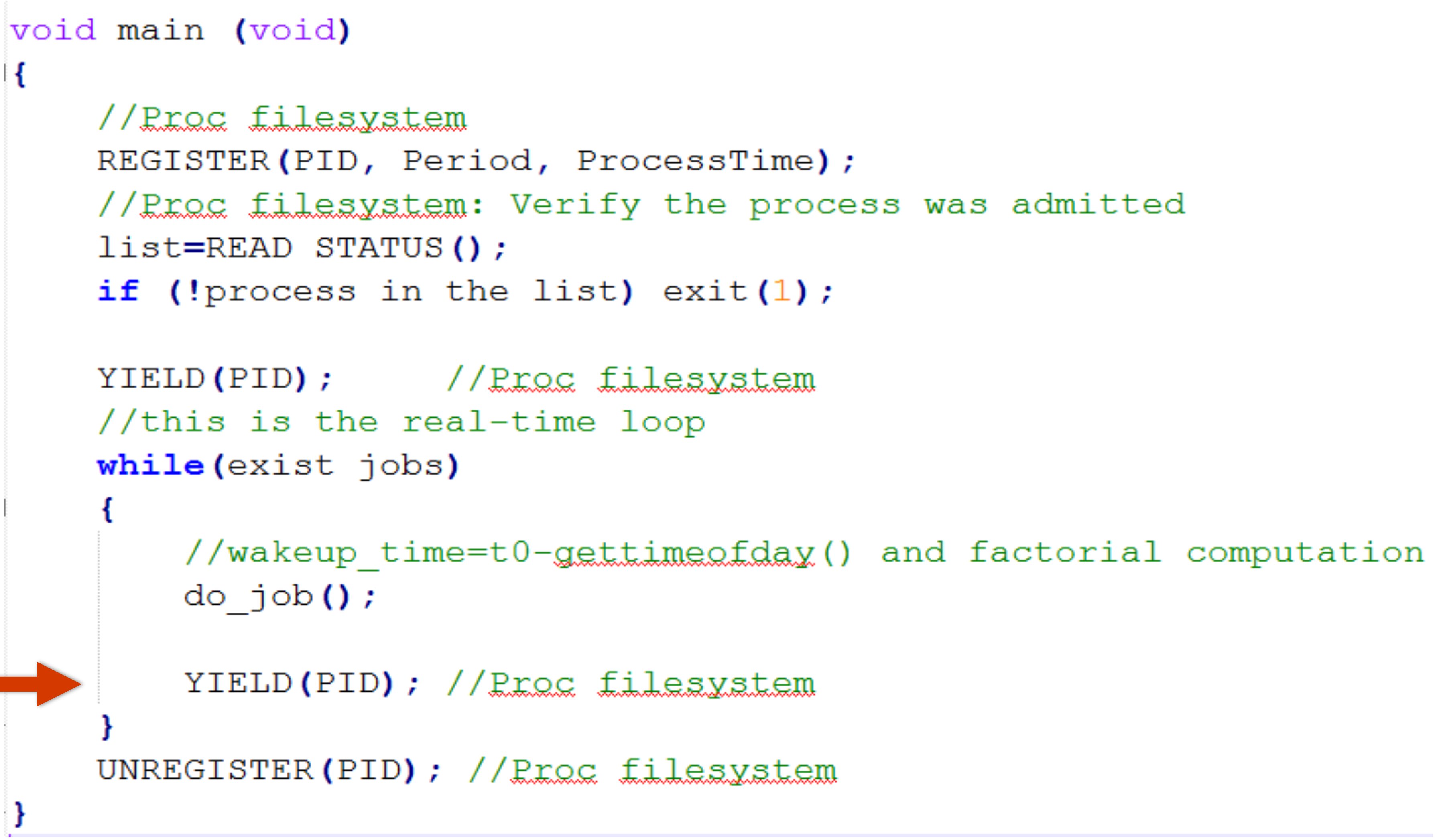


MP2 User Process Behavior



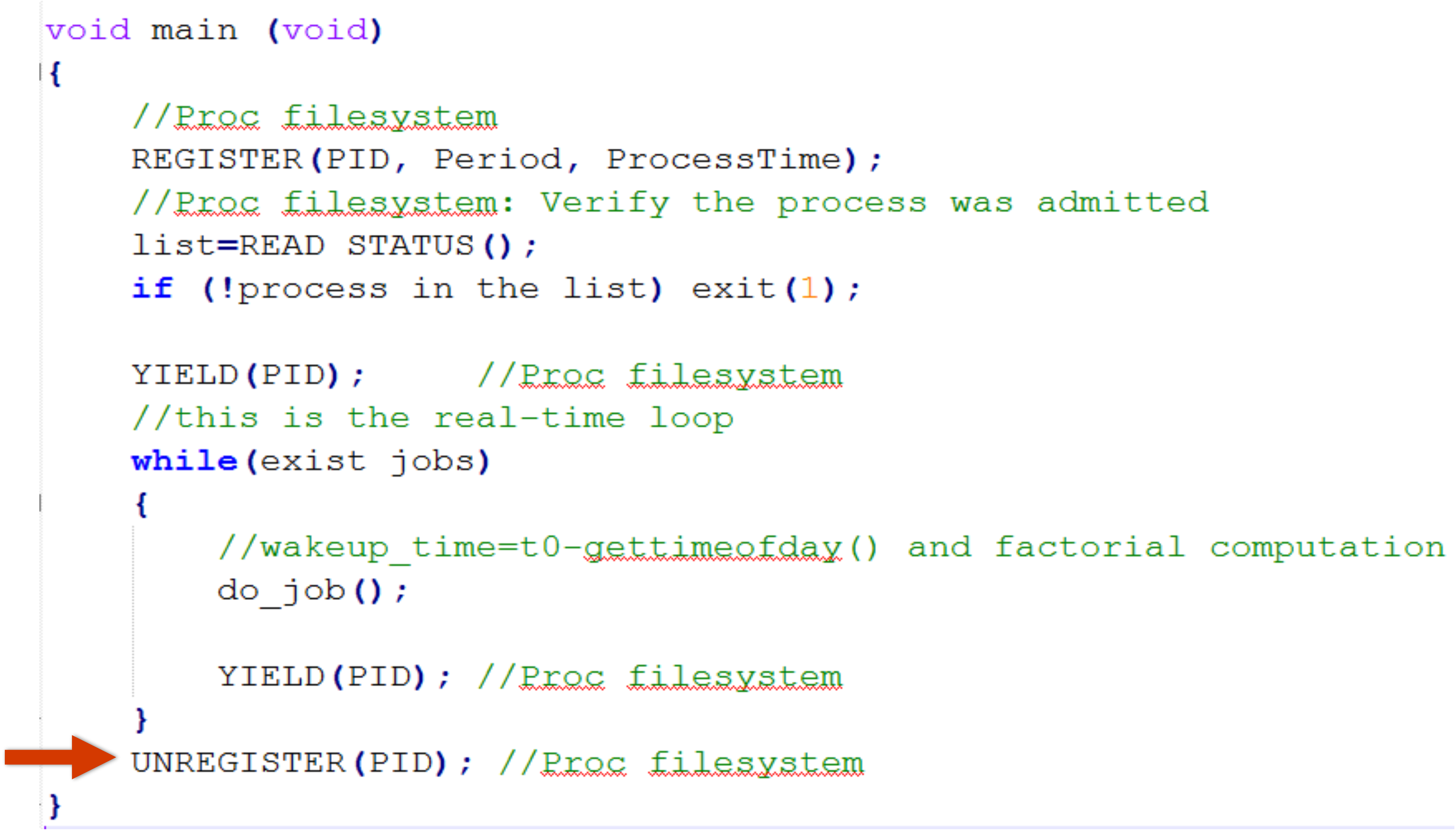
```
void main (void)
{
    //Proc filesystem
    REGISTER(PID, Period, ProcessTime);
    //Proc filesystem: Verify the process was admitted
    list=READ STATUS();
    if (!process in the list) exit(1);

    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
}
```

MP2 User Process Behavior



```
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
}
```

MP2 User Process Behavior



```
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
}
```

MP2 Process State

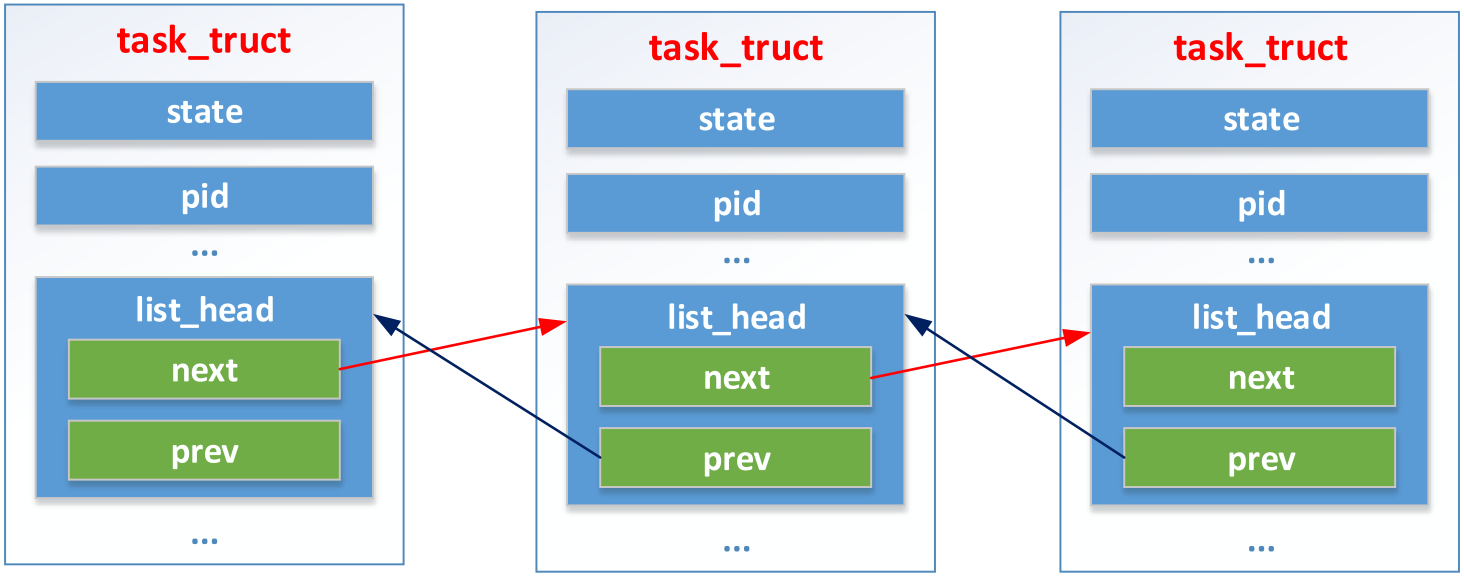
- A process in MP2 can be in one of three states

1. READY: a new job is ready to be scheduled
2. RUNNING: a job is currently running and using the CPU
3. SLEEPING: job has finished execution and process is waiting for the next period

- A job is not allowed to run before its appropriate period

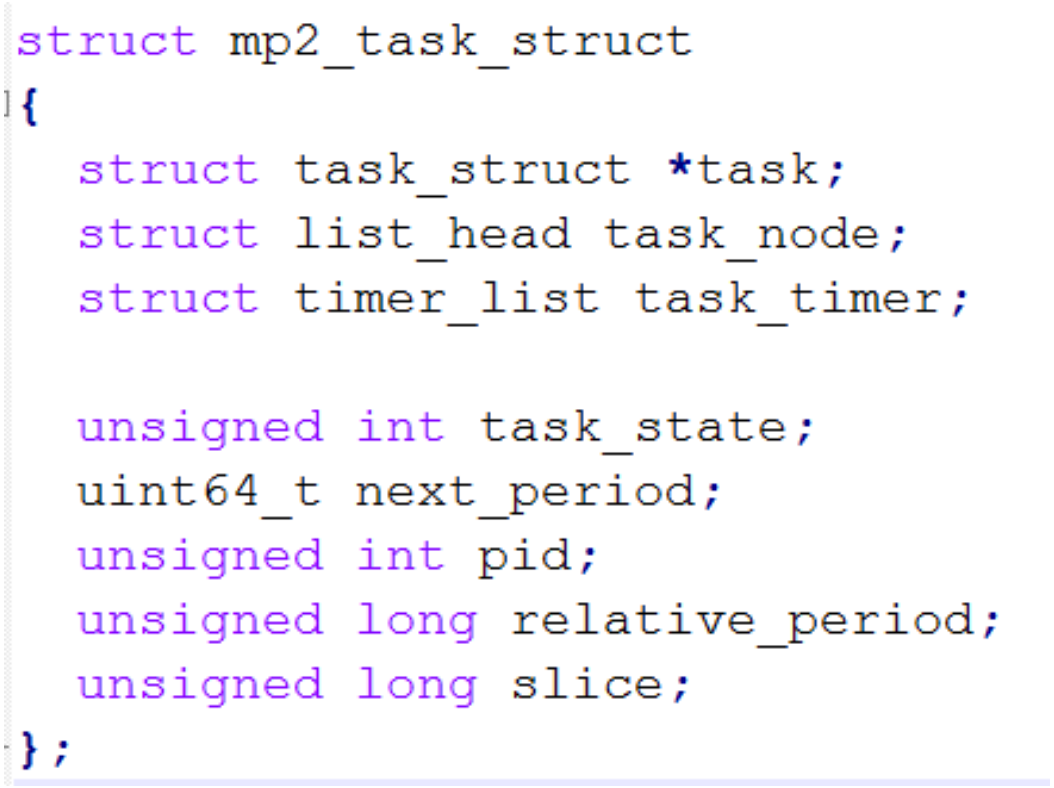
MP2 Process Control Block

- PCB is defined by `task_struct`
- PCB is manager by a circular doubly linked list
- Maintain pointer to current running state



MP2 Extending the PCB

- Extend PCB to hold MP2-specific information, example,



```
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 Logic

- We will use a kernel thread to handle the scheduling logic
- It will handle context switches as needed
- There are two cases in which a context switch is needed
  1. Received a YIELD message from an application
  2. The wakeup timer of a process has expired, i.e., its new period has started

MP2 Scheduling Logic

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

MP2 Context Switching

- We will use the kernel scheduling API
  - `schedule():` trigger the kernel scheduler
  - `wake_up_process (struct task_struct *)`
  - `sched_setscheduler():` set scheduling parameters
    - FIFO for real time scheduling, NORMAL for regular processes, etc.
  - `set_current_state()`
  - `set_task_state()`

MP2 Scheduler API Example

- To sleep and trigger a context switch

```
set_current_state(TASK_INTERRUPTIBLE);

schedule();
```

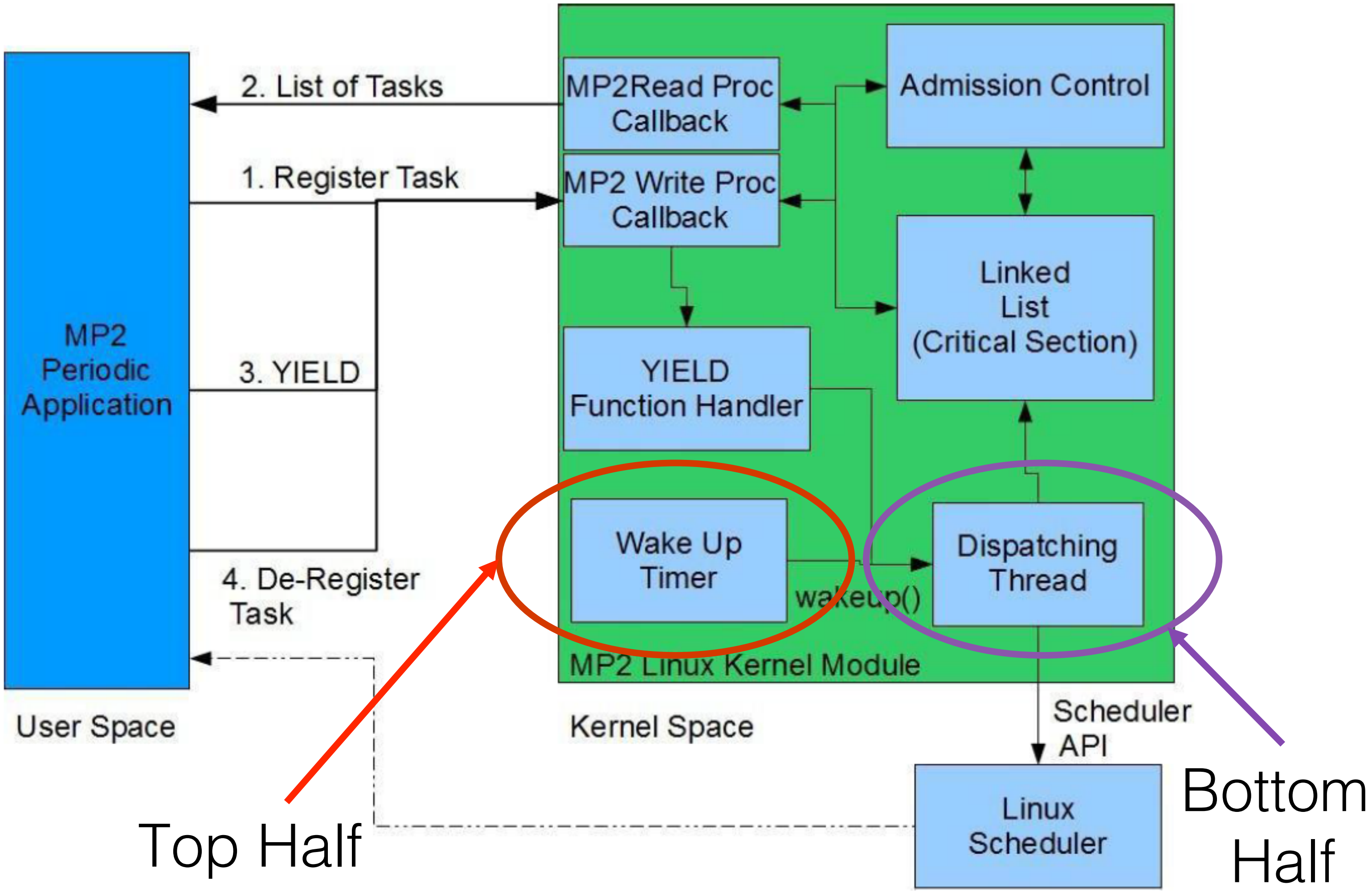
- To wake up a process

```
struct task_struct * sleeping_task;

...

wake_up_process(sleeping_task);
```

MP2 A Note About Kthreads

- You will need to explicitly put the kernel thread to sleep when you're done with your work
  - Otherwise it will keep running forever
- You also need to explicitly check for signals
  - Check if should stop working
  - `kthread_should_stop()`

MP2 Timer and Scheduler



MP2 Final Notes

- Develop things incrementally, follow the mp2 description
- Test things one at a time
- Use fixed point arithmetic
- Use global variables for persistent state
- Remember to cleanup everything
  - Failure to do so may not allow you to insert your module again