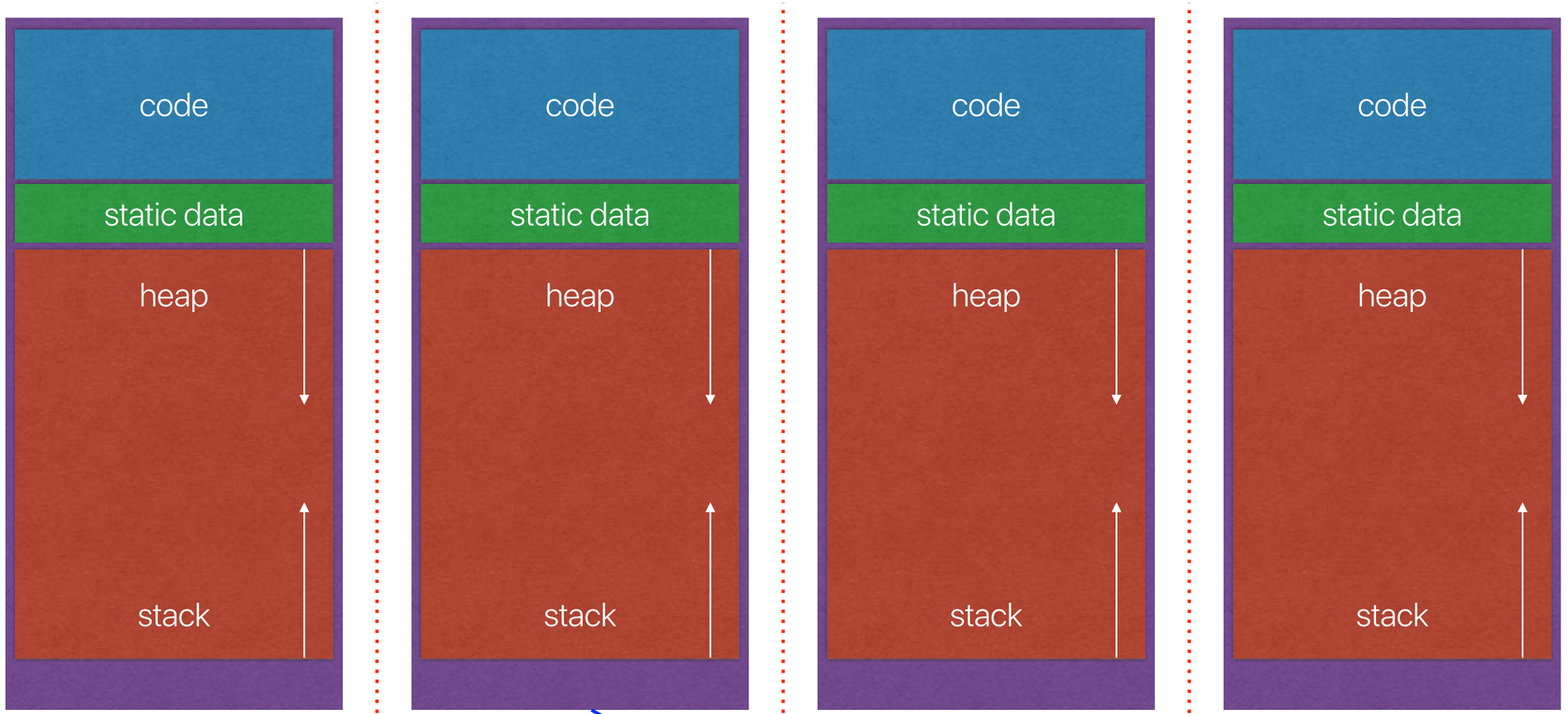
Process/Thread/Task Scheduling

Hung-Wei Tseng

Recap: Each process has a separate virtual memory space

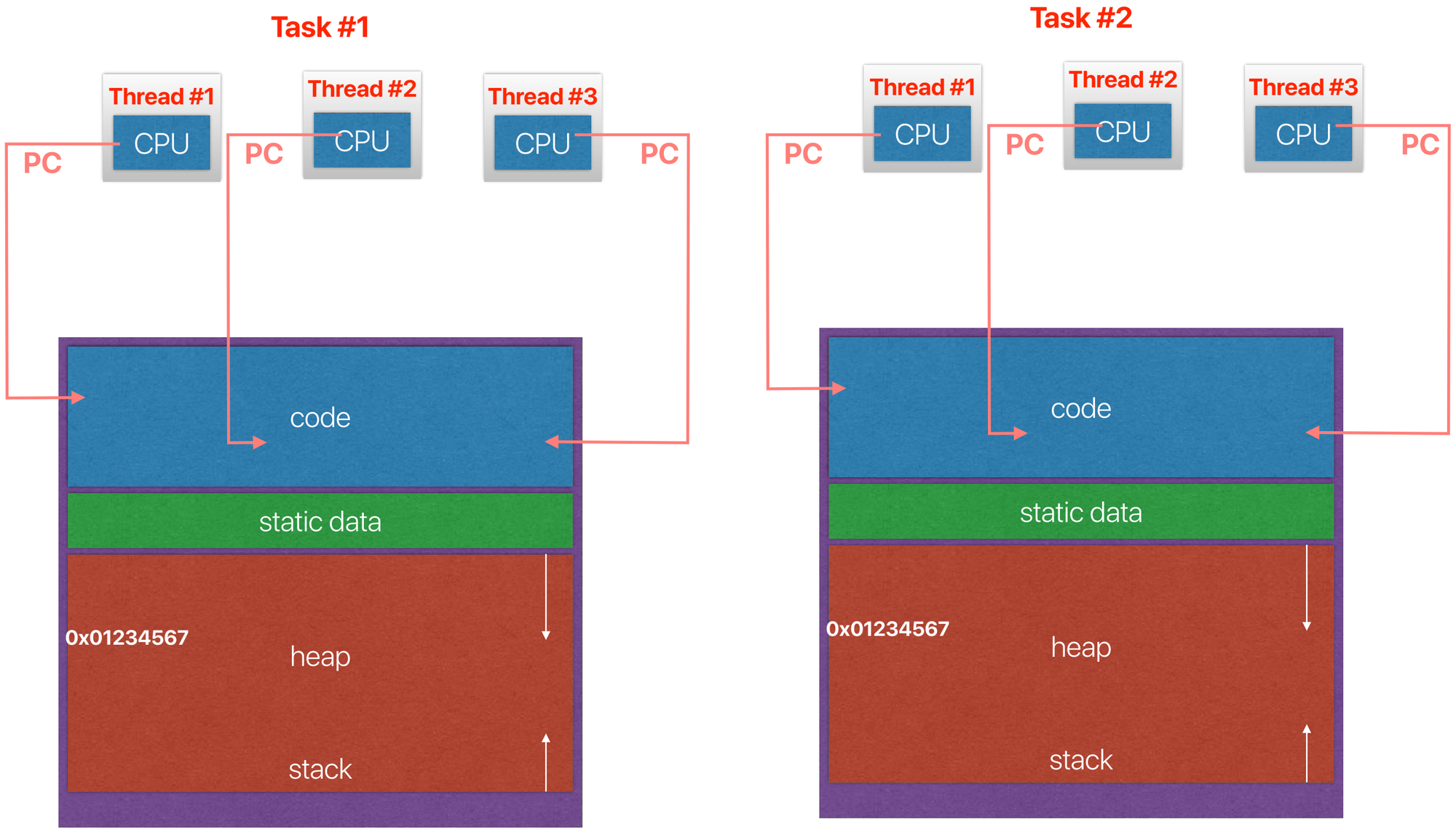


They are isolated from one another. Each of them is not supposed to know what happens to another one

Processor

Virtually, every process seems to have a processor, but only a few of them are physically executing.

Recap: Threads



Recap: Why Threads?

- Process is an abstraction of a computer
  - When you create a process, you duplicate everything
  - However, you only need to duplicate CPU abstraction to parallelize computation tasks
- Threads as lightweight processes
  - Thread is an abstraction of a CPU in a computer
  - Maintain separate execution context
  - Share other resources (e.g. memory)

Joint Banking

- If the shared variable, balance, initially has the value of 40, what value(s) might it hold after threads A and B finish after we call `deposit(10)` and `withdraw(20)`?

A. 30
B. 20 or 30
C. 20, 30, or 50
D. 10, 20, or 30

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Thread B

```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Critical sections

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Thread B

```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

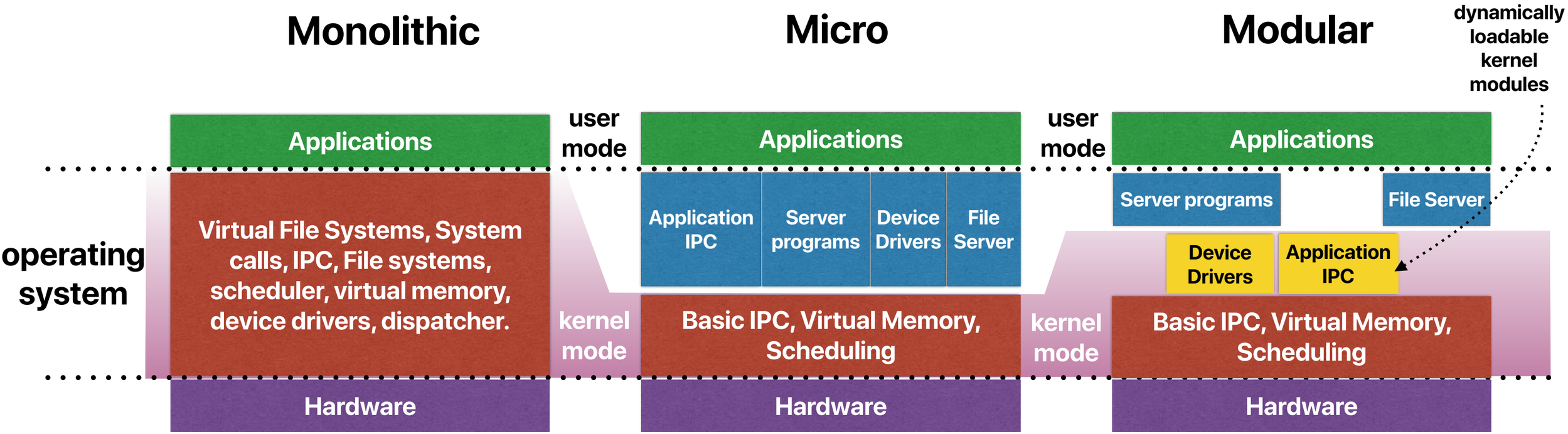
Use pthread_lock

```
int max;
volatile int balance = 0; // shared global variable
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first <loopcount>\n");
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      Pthread_mutex_lock(&lock);
      balance++;
      Pthread_mutex_unlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

Kernel: Types of Kernels



Original UNIX

Hydra, Mach

Linux, Windows, MacOS

Current scoreboard

Red	Blue
6	7

Outline

- Mechanisms of changing processes
- Basic scheduling policies
- An experimental time-sharing system — The Multi-Level Scheduling Algorithm
- Scheduler Activations
- Getting locks done correctly with modern OS scheduling

The mechanisms of changing processes

The mechanisms of changing the running processes

- Cooperative Multitasking (non-preemptive multitasking)
- Preemptive Multitasking

Poll close in 1:30

Cooperative multitasking

- How many of the following statements about cooperative multitasking is/are correct?
  - ① The OS can change the running process if the current process give up the resource
  - ② The OS can change the running process if the current process traps into OS kernel
  - ③ The OS can change the running process if the current process raise an exception like divide by zero
  - ④ The OS can actively change the running process if the current process is running for a long enough time
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Poll close in 1:30

Cooperative multitasking

- How many of the following statements about cooperative multitasking is/are correct?
  - ① The OS can change the running process if the current process give up the resource
  - ② The OS can change the running process if the current process traps into OS kernel
  - ③ The OS can change the running process if the current process raise an exception like divide by zero
  - ④ The OS can actively change the running process if the current process is running for a long enough time
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Cooperative multitasking

- How many of the following statements about cooperative multitasking is/are correct?
  - ① The OS can change the running process if the current process give up the resource
  - ② The OS can change the running process if the current process traps into OS kernel
  - ③ The OS can change the running process if the current process raise an exception like divide by zero **Anytime if we make a system call to the OS, the OS can potentially switch a proces**
  - ④ The OS ~~can~~ actively change the running process if the current process is running for a long enough time **Unfortunately, the OS cannot — if the process never traps**

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Cooperative multitasking — processes voluntarily yield control periodically or when idle in order to enable multiple applications to be run simultaneously

Preemptive Multitasking

- The OS controls the scheduling — can change the running process even though the process does not give up the resource
- But how?

Poll close in 1:30

How to achieve preemptive multitasking

- Which of the following mechanism are used to support preemptive multitasking?
  A. Exception
  B. Interrupt
  C. System calls

Poll close in 1:30

How to achieve preemptive multitasking

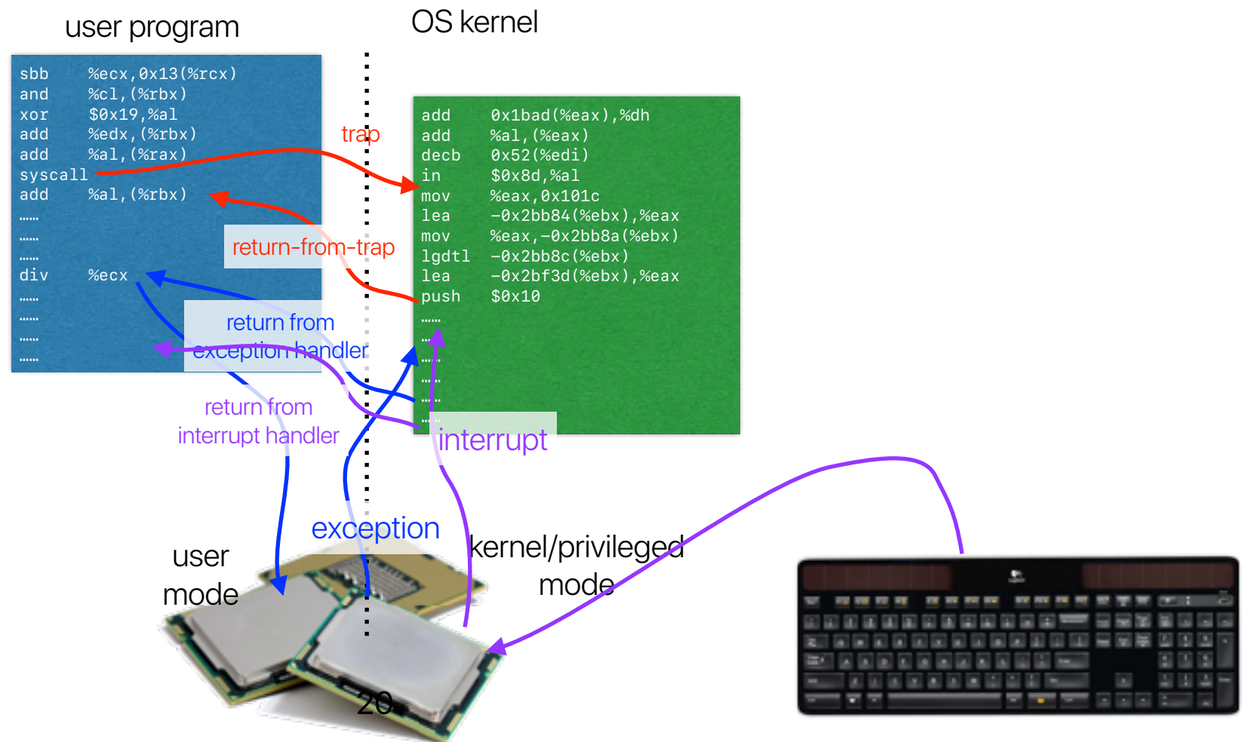
- Which of the following mechanism are used to support preemptive multitasking?
  A. Exception
  B. Interrupt
  C. System calls

How to achieve preemptive multitasking

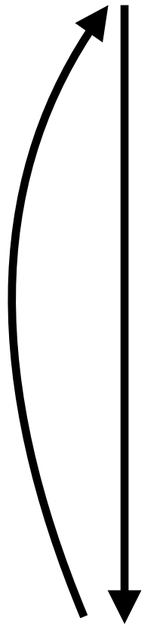
- Which of the following mechanism are used to support preemptive multitasking?
  A. Exception
  B. Interrupt
  C. System calls

Three ways to invoke OS handlers

- System calls / trap instructions — raised by applications
  - Display images, play sounds
- Exceptions — raised by processor itself
  - Divided by zero, unknown memory addresses
- Interrupts — raised by hardware
  - Keystroke, network packets

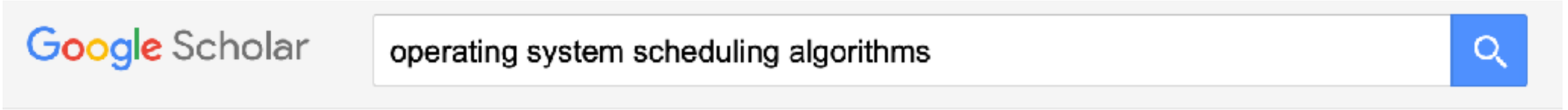


How preemptive multitasking works



- Setup a **timer** (a hardware feature by the processor)event before the process start running
- After a certain period of time, the **timer** generates **interrupt** to force the running process transfer the control to OS kernel
- The OS kernel code decides if the system wants to continue the current process
  - If not — context switch
  - If yes, return to the process

Scheduling Policies from Undergraduate OS classes



Articles

About 2,380,000 results (**0.10** sec)

CPU Scheduling

- Virtualizing the processor
  - Multiple processes need to share a single processor
  - Create an illusion that the processor is serving my task by rapidly switching the running process
- Determine which process gets the processor for how long

What you learned before

- Non-preemptive/cooperative: the task runs until it finished
  - FIFO/FCFS: First In First Out / First Come First Serve
  - SJF: Shortest Job First
- Preemptive: the OS periodically checks the status of processes and can potentially change the running process
  - STCF: Shortest Time-to-Completion First
  - RR: Round robin

Poll close in 1:30

Best for turn-around time

- Assume that we have the following 3 processes

	Arrival time	Task length
P1	0	5
P2	1	4
P3	4	1

which of the following scheduling policy yields the best average turn-around time? (assume we prefer not to switch process if two process have the same criteria)

A. FIFO/FCFS: First In First Out / First Come First Serve
B. SJF: Shortest Job First
C. STCF: Shortest Time-to-Completion First
D. RR: Round robin
E. Two of the above

Poll close in 1:30

Best for turn-around time

- Assume that we have the following 3 processes

	Arrival time	Task length
P1	0	5
P2	1	4
P3	4	1

which of the following scheduling policy yields the best average turn-around time? (assume we prefer not to switch process if two process have the same criteria)

A. FIFO/FCFS: First In First Out / First Come First Serve
B. SJF: Shortest Job First
C. STCF: Shortest Time-to-Completion First
D. RR: Round robin
E. Two of the above

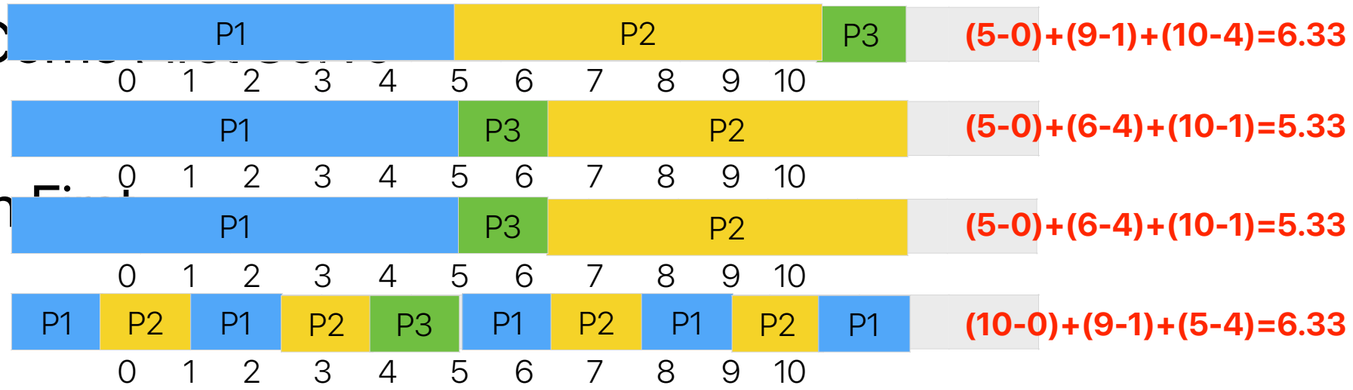
Best for turn-around time

- Assume that we have the following 3 processes

	Arrival time	Task length
P1	0	5
P2	1	4
P3	4	1

which of the following scheduling policy yields the best average turn-around time? (assume we prefer not to switch process if two process have the same criteria)

A. FIFO/FCFS: First In First Out / First C
B. ✓ SJF: Shortest Job First
C. ✓ STCF: Shortest Time-to-Completion Fi
D. RR: Round robin
E. Two of the above



28

Poll close in 1:30

Parameters for policies

- How many of the following scheduling policies require knowledge of process run times before execution?
  - ① FIFO/FCFS: First In First Out / First Come First Serve
  - ② SJF: Shortest Job First
  - ③ STCF: Shortest Time-to-Completion First
  - ④ RR: Round robin

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Poll close in 1:30

Parameters for policies

- How many of the following scheduling policies require knowledge of process run times before execution?
  - ① FIFO/FCFS: First In First Out / First Come First Serve
  - ② SJF: Shortest Job First
  - ③ STCF: Shortest Time-to-Completion First
  - ④ RR: Round robin

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Parameters for policies

- How many of the following scheduling policies require knowledge of process run times before execution?
  - ① FIFO/FCFS: First In First Out / First Come First Serve
  - ② SJF: Shortest Job First
  - ③ STCF: Shortest Time-to-Completion First
  - ④ RR: Round robin

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

You can never know the execution time before executing them!
— These policies are not realistic

The best ones you learned in undergraduate OS does not even work in real!
— forget about them in real implementation

An experimental time-sharing system

Fernando J. Corbató, Marjorie Merwin-Daggett and Robert C. Daley
Massachusetts Institute of Technology, Cambridge, Massachusetts

Poll close in 1:30

Why Multi-level scheduling algorithm

- Why MIT's experimental time-sharing system proposes Multi-level schedule algorithm? How many of the followings are correct?
  - ① Optimize for the average response time of tasks
  - ② Optimize for the average turn-around time of tasks
  - ③ Optimize for the performance of long running tasks
  - ④ Guarantee the fairness among tasks
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Poll close in 1:30

Why Multi-level scheduling algorithm

- Why MIT's experimental time-sharing system proposes Multi-level schedule algorithm? How many of the followings are correct?
  - ① Optimize for the average response time of tasks
  - ② Optimize for the average turn-around time of tasks
  - ③ Optimize for the performance of long running tasks
  - ④ Guarantee the fairness among tasks
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Why Multi-level scheduling algorithm

- Why MIT's experimental time-sharing system proposes Multi-level schedule algorithm? How many of the followings are correct?
  - ① Optimize for the average response time of tasks
  - ② Optimize for the average turn-around time of tasks
  - ③ Optimize for the performance of long running tasks
  - ④ Guarantee the fairness among tasks

A. 0
B. 1
C. 2
D. 3
E. 4

4. The response time for programs of equal size, entering the system at the same time, and being run for multiple quanta, is no worse than approximately twice the response-time occurring in a single quanta round-robin procedure. If

To illustrate the strategy that can be employed to improve the saturation performance of a time-sharing system, a multi-level scheduling algorithm is presented. This algorithm also

Several important conclusions can be drawn from the above algorithm which allow the performance of the system to be bounded.

Why Multi-level scheduling algorithm?

- System **saturation** — the demand of computing is larger than the physical **processor** resource available
- Service level degrades
  - Lots of **program** swap ins-and-outs (known as **context switches** in our current terminology)
  - User interface response time is bad — you have to wait until your turn
  - Long running tasks cannot make good progress — frequent swap in-and-out

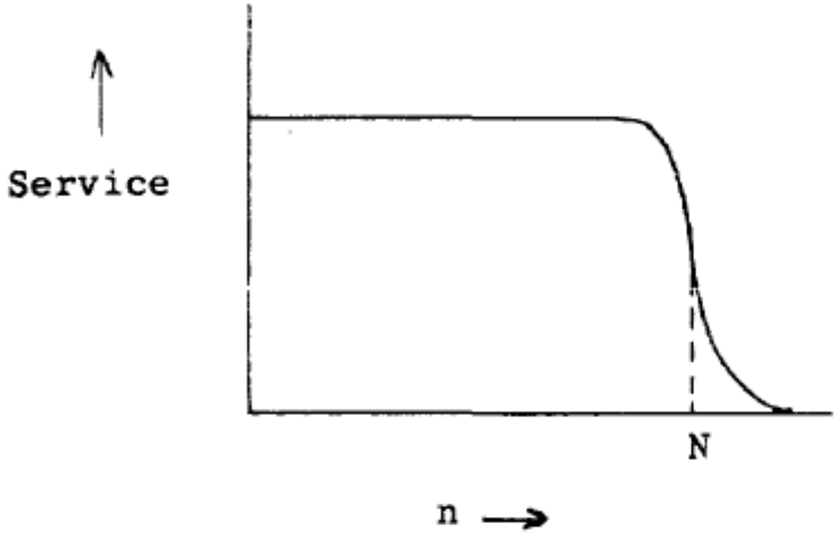
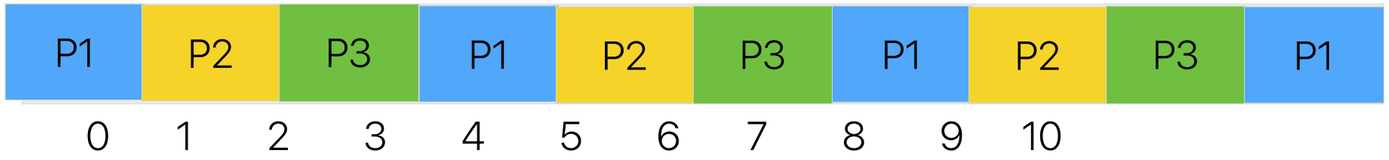


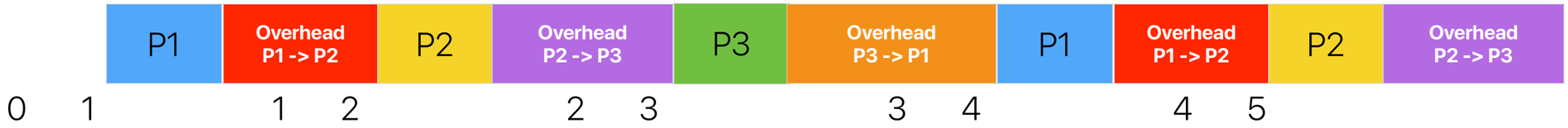
Figure 1. Service vs. Number of Active Users

Context Switch Overhead

You think round robin should act like this —



But the fact is —



- **Your processor utilization can be very low if you switch frequently**
- **No process can make sufficient amount of progress within a given period of time**
- **It also takes a while to reach your turn**

The Multilevel Scheduling Algorithm

- Place new process in the one of the queue
  - Depending on the program size

$$\ell_o = \left\lceil \log_2 \left(\left[\frac{w_p}{w_q} \right] + 1 \right) \right\rceil$$

w_p is the program memory size — smaller ones are assigned to lower numbered queues

Why?

  - **Smaller tasks are given higher priority in the beginning**
- Schedule processes in one of N queues
  - Start in initially assigned queue n
  - Run for 2^n quanta (where n is current depth)
  - If not complete, move to a higher queue (e.g. n +1)
  - **Larger process will execute longer before switch**
- Level m is run only when levels 0 to m-1 are empty
- **Smaller process, newer process are given higher priority**

The Multilevel Scheduling Algorithm

- Not optimized for anything — it's never possible to have an optimized scheduling algorithm without prior knowledge regarding all running processes
- It's practical — many scheduling algorithms used in modern OSes still follow the same idea

Lottery Scheduling: Flexible Proportional-Share Resource Management

Carl A. Waldspurger and William E. Weihl

Why Lottery

enormous impact on throughput and response time. Accurate control over the quality of service provided to users and applications requires support for specifying relative computation rates. Such control is desirable across a wide spectrum of systems. For long-running computations such as scientific applications and simulations, the consumption of computing resources that are shared among users and applications of varying importance must be regulated [Hel93]. For interactive computations such as databases and media-based applications, programmers and users need the ability

We want Quality of Service

ware systems. In fact, with the exception of hard real-time systems, it has been observed that the assignment of priorities and dynamic priority adjustment schemes are often ad-hoc [Dei90]. Even popular priority-based schemes for CPU allocation such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

No body knows how they work...

Few general-purpose schemes even come close to supporting flexible, responsive control over service rates.

Most approaches are not flexible, responsive

Existing *fair share* schedulers [Hen84, Kay88] and *microeconomic* schedulers [Fer88, Wal92] successfully address some of the problems with absolute priority schemes. However, the assumptions and overheads associated with these systems limit them to relatively coarse control over long-running computations. Interactive systems require

The overhead of running those algorithms are high!

Solution — Lottery and Tickets

Poll close in 1:30

What do ticket abstraction promote?

- How many of the following can the **ticket** abstraction in the lottery paper promote?
  - ① Proportional fairness
  - ② Machine-independent implementation of the scheduling policy
  - ③ Generic scheduling policy across different devices
  - ④ Starvation free

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Poll close in 1:30

What do ticket abstraction promote?

- How many of the following can the **ticket** abstraction in the lottery paper promote?
  - ① Proportional fairness
  - ② Machine-independent implementation of the scheduling policy
  - ③ Generic scheduling policy across different devices
  - ④ Starvation free

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

What lottery proposed?

- Each process hold a certain number of lottery tickets
- Randomize to generate a lottery
- If a process wants to have higher priority
  - Obtain more tickets!

What do ticket abstraction promote?

- How many of the following can the **ticket** abstraction in the lottery paper promote?
  - ① Proportional fairness
  - ② Machine-independent implementation of the scheduling policy
  - ③ Generic scheduling policy across different devices
  - ④ Starvation free
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Tickets represent the share of a process should receive from a resource

The ticket abstraction can be independent of machine speeds or details

You may use tickets on everything you would like to share

Eventually every process with a ticket gets to run
It's also state-free — reduce the overhead

Ticket economics

- Ticket transfers
- Ticket inflation
- Ticket currencies
- Compensation tickets

How good is lottery?

- The overhead is not too bad
  - 1000 instructions ~ less than 500 ns on a 2 GHz processor
- Fairness
  - Figure 5: average ratio in proportion to the ticket allocation
- Flexibility
  - Allows Monte-Carlo algorithm to dynamically inflate its tickets
- Ticket transfer
  - Client-server setup

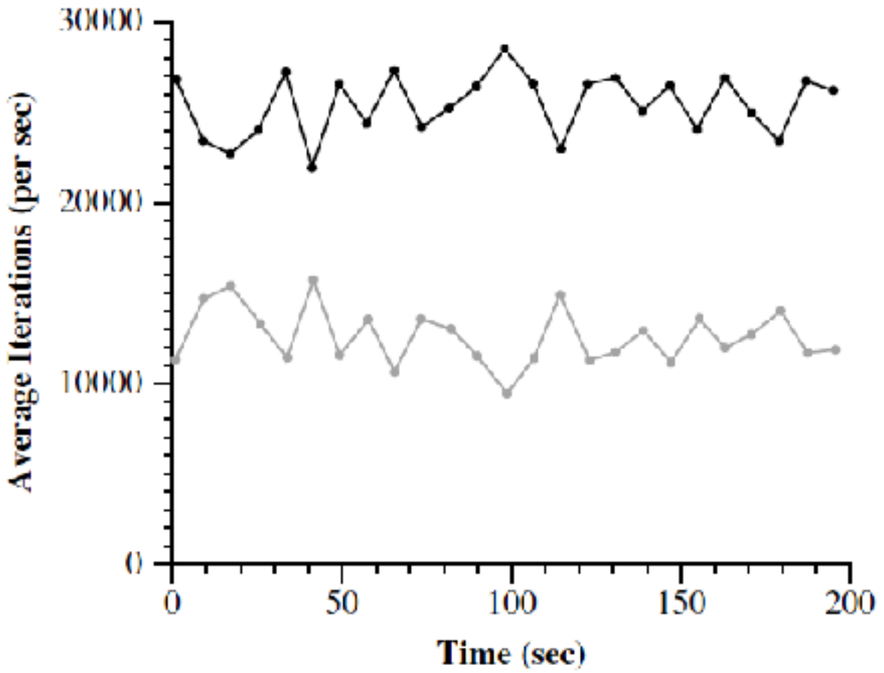


Figure 5: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.

The impact of “lottery”

- Data center scheduling
  - You buy “times”
  - Lottery scheduling of your virtual machine

Will you use lottery for your system?

- Will it be good for
  - Event-driven application
  - Real-time application
  - GUI-based system
- Is randomization a good idea?
  - The authors later developed a deterministic stride-scheduling

Announcement

- Reading quizzes due next Tuesday
- Project released
  - Groups in 2
  - Start as soon as you can — due in about a month
  - **Pull the latest version — had some changes for later kernel versions**
    https://github.com/hungweitseng/CS202-ResourceContainer
  - **Install an Ubuntu Linux 16.04.07 VM as soon as you can!**
  - **Please do not use a real machine — you may not be able to reboot again**
- Midterm
  - Will release on 2/10/2021 0:00am and due on 2/15/2021 11:59:00pm
  - You will have to find a consecutive, non-stop 80-minute slot with this period
  - One time, cannot reinitiate — please make sure you have a stable system and network
  - **No late submission is allowed**

Computer
Science &
Engineering

202

つづく