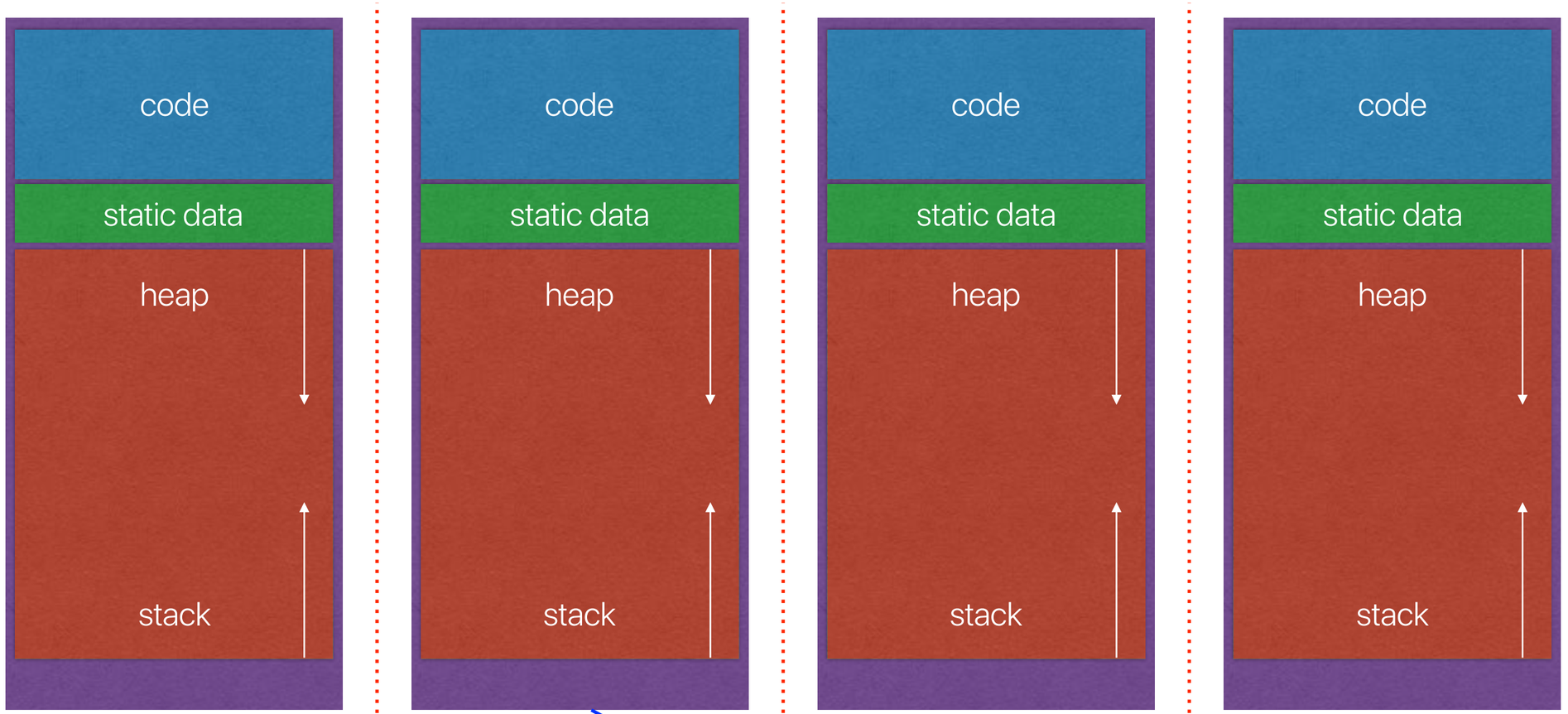
Process/Thread/Task Scheduling

Hung-Wei Tseng

Recap: Each process has a separate virtual memory space

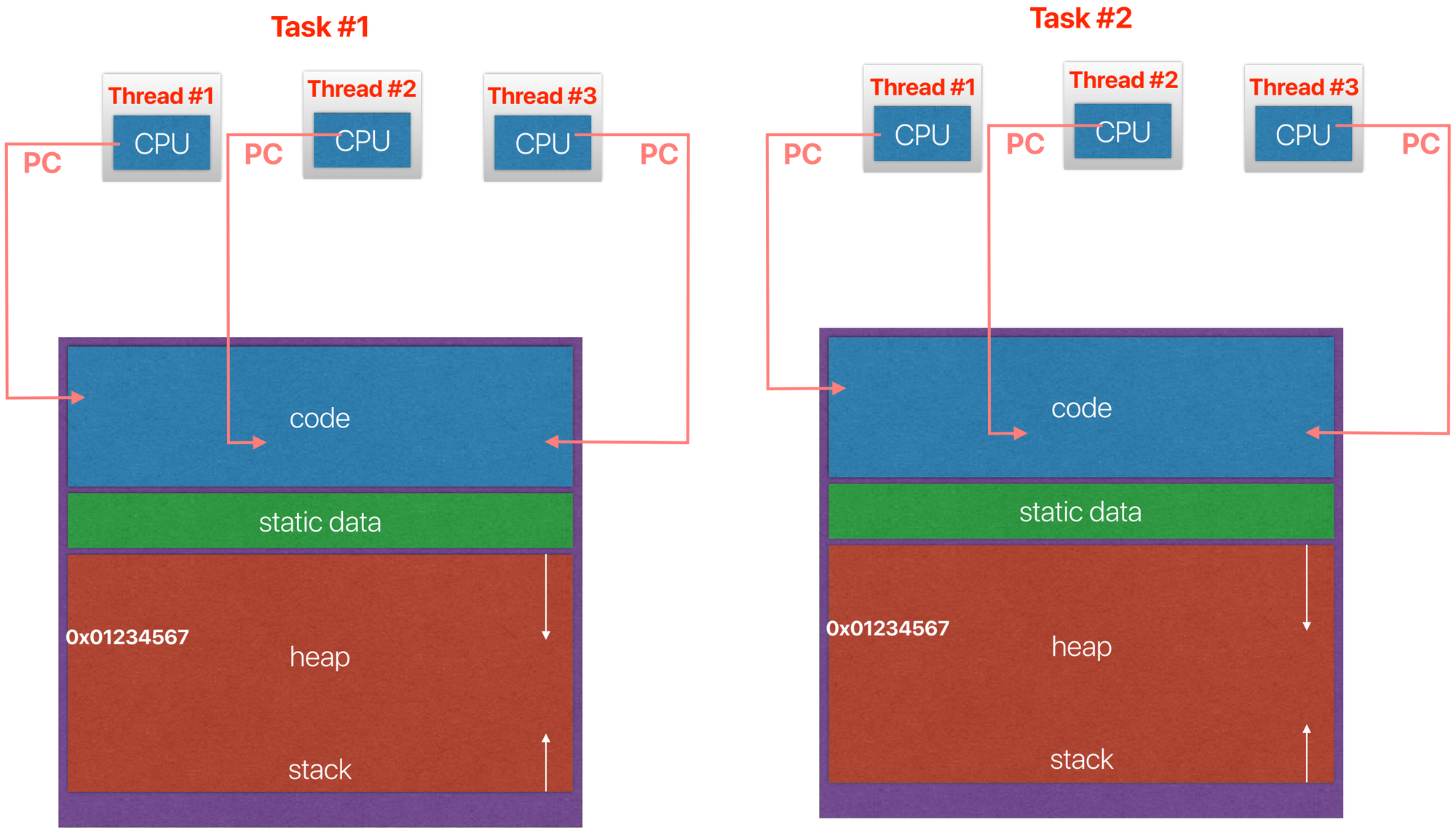


They are isolated from one another. Each of them is not supposed to know what happens to another one

Processor

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

Recap: Threads

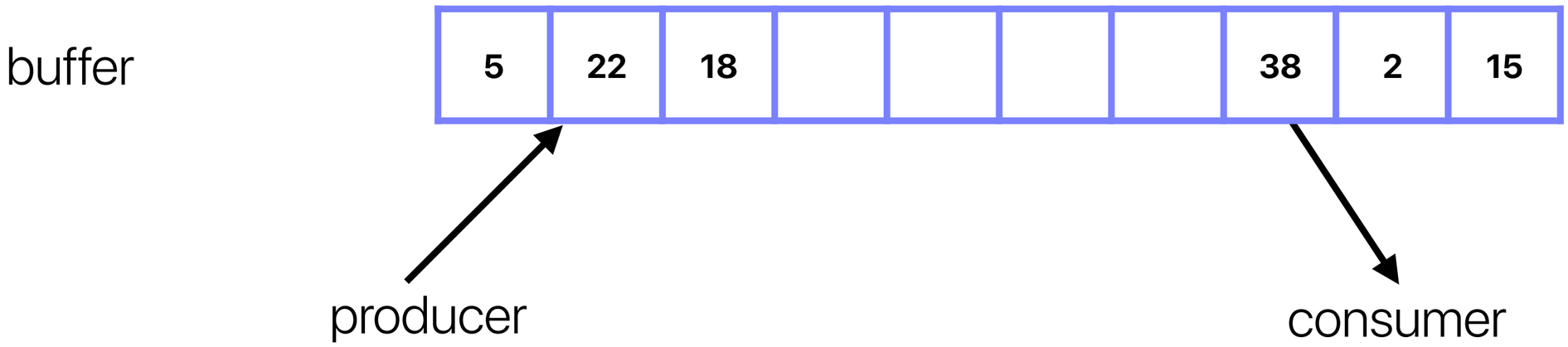


Recap: Solving the "Critical Section Problem"

1. Mutual exclusion — at most one process/thread in its critical section
2. Progress/Fairness — a thread outside of its critical section cannot block another thread from entering its critical section
3. Progress/Fairness — a thread cannot be postponed indefinitely from entering its critical section
4. Accommodate nondeterminism — the solution should work regardless the speed of executing threads and the number of processors

Bounded-Buffer Problem

- Also referred to as "producer-consumer" problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer



Without synchronization, you may write

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        int item = buffer[out];
        out = (out + 1) %
BUFF_SIZE;
        // do something w/ item
    }
    return NULL;
}
```

Use locks

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

You cannot produce and consume simultaneously!

Semaphores

Semaphores

- A synchronization variable
- Has an integer value — current value dictates if thread/process can proceed
- Access granted if val > 0, blocked if val == 0
- Maintain a list of waiting processes

Semaphore Operations

- `sem_wait(S)`
  - if S > 0, thread/process proceeds and decrement S
  - if S == 0, thread goes into “waiting” state and placed in a special queue
- `sem_post(S)`
  - if no one waiting for entry (i.e. waiting queue is empty), increment S
  - otherwise, allow one thread in queue to proceed

Semaphore Op Implementations

```
sem_init(sem_t *s, int initvalue) {
    s->value = initvalue;
}
```

```
sem_wait(sem_t *s) {
    while (s->value <= 0)
        put_self_to_sleep(); // put self to sleep
    s->value--;
}
```

```
sem_post(sem_t *s) {
    s->value++;
    wake_one_waiting_thread(); // if there is one
}
```

Atomicity in Semaphore Ops

- Semaphore operations must operate atomically
  - Requires lower-level synchronization methods requires (test-and-set, etc.)
  - Most implementations still require on busy waiting in spinlocks
- What did we gain by using semaphores?
  - Easier for programmers
  - Busy waiting time is limited

Using semaphores

- What variables to use for this problem?

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    Sem_init(&filled, 0);
    Sem_init(&empty, BUFF_SIZE);
    while(TRUE) {
        int item = …;
        Sem_wait(&W);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Sem_post(&X);
    }
    printf("parent: end\n");
    return 0;
}
```

```
int buffer[BUFF_SIZE]; // shared global
sem_t filled, empty;
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Sem_wait(&Y);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        // do something w/ item
        Sem_post(&Z);
    }
    return NULL;
}
```

	W	X	Y	Z
A	empty	empty	filled	filled
B	empty	filled	filled	empty
C	filled	empty	empty	filled

Outline

- Mechanisms of changing processes
- Basic scheduling policies
- An experimental time-sharing system — The Multi-Level Scheduling Algorithm
- Scheduler Activations

The mechanisms of changing processes

The mechanisms of changing the running processes

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

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

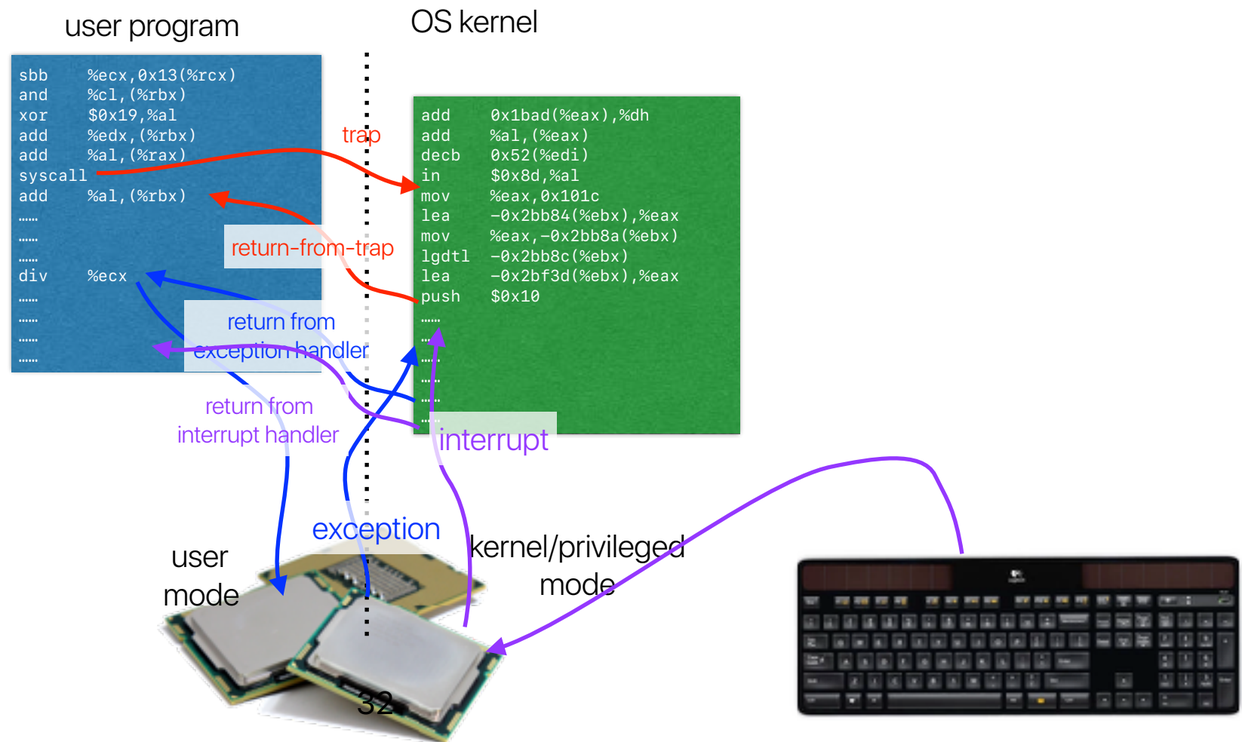
- The OS controls the scheduling — can change the running process even though the process does not give up the resource
- But how?

How to achieve preemptive multitasking

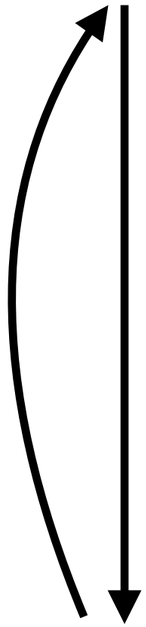
- Which of the following mechanism are used to support preemptive multitasking?
  A. Exception
  B. Interrupt
  C. System calls

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

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

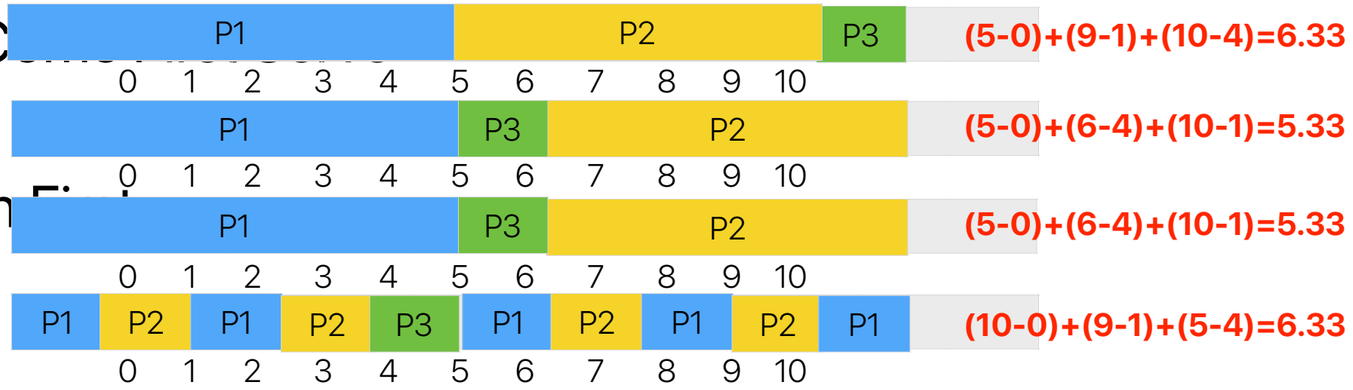
Best for turn-around time

- 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



41

Parameters for policies

- How many of the following scheduling policies require knowledge of process run times?
  - 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

These policies are not realistic — forget about them in real implementation

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

An experimental time-sharing system

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

Why Multi-level scheduling algorithm

- Why MIT's experimental time-sharing system proposes Multi-level schedule algorithm? How many of the following
  - 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

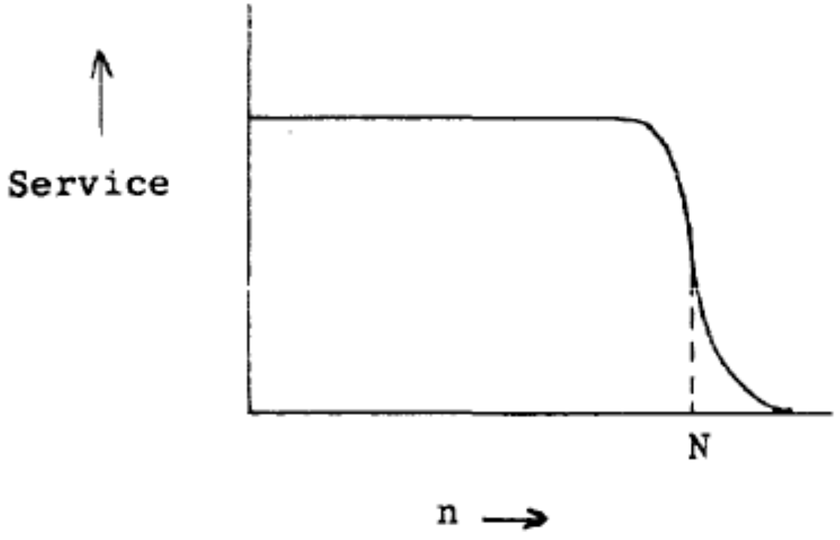
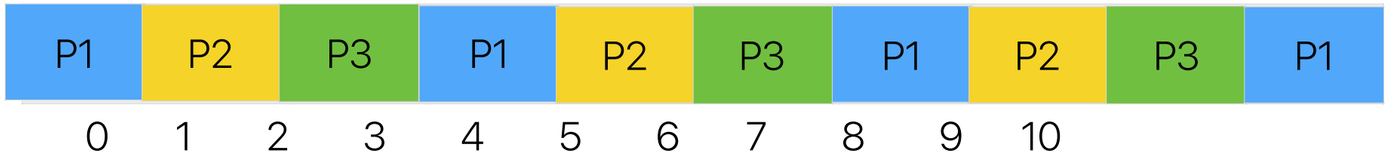


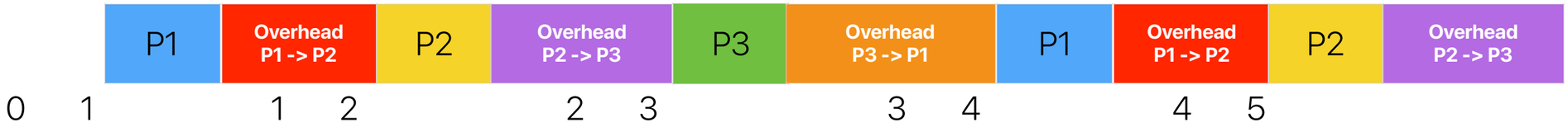
Figure 1. Service vs. Number of Active Users

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

Correction: disable interrupts?

- Which of the following can disabling interrupts guarantee on multicore processors?

What if we have multiple processors?

A. At most one process/thread in its critical section
B. A thread outside of its critical section cannot block another thread from entering its critical section
C. A thread cannot be postponed indefinitely from entering its critical section

What if the thread hold the critical section crashes?

D. The solution should work regardless the speed of executing threads and the number of processors
E. None of the above

What if we have multiple processors?

Announcement

- Reading quiz due next Tuesday
- Project due date 3/3
  - In about a month
  - Please come to our office hours if you need help for your projects