CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Spring 2022 L10
Scheduling, Synchronization

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

FAQ

- Shortest remaining time first (Preemptive SJF)
  - Need to track the remaining time for all processes
- Round Robin
  - Need to track the position of the processes in the Ready Queue
  - Also need to track the remaining time needed
  - Illustration on youtube
  - Animation CPU Scheduling Algorithm Visualization
- Time quantum- How to decide?
  - Rule of thumb: 80% of CPU bursts should be shorter than q

Disclaimer: I have not verified the accuracy of the on-line sources.

Schedulers

- Scheduling schemes have continued to evolve with continuing research. A comparison.
- Multilevel Feedback Queue Details at ARPACI-DUSSEAU
- Linux Completely fair scheduler (Con Kolivas, Anaesthetist):
  - Variable time-slice based on number and priority of the tasks in the queue.
    - Maximum execution time based on waiting processes (Q/n).
  - Processes kept in a red-black binary tree with scheduling complexity of O(log N)
  - Process with lowest weighted spent execution (virtual run time) time is picked next. Weighted by priority ("niceness").

Project

- See Document: Schedule/Proj Proposal or Canvas/Assignments
- Choices: Research (topics provided) or development (IoT). Some research/original thinking required for either.
- Deadlines: subject to revision.
  - D1. Team composition and idea proposal, Fri 2/25/2022
  - D2. Progress report, Thurs 4/7/2022
  - D3. Slides and final reports, Thurs 4/28/2022
  - D4. Presentations/demos 5/2-5/4 as arranged
  - D5: Peer Reviews due 5/7/2022 Sat

CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
ICQ

Shortest Job First Preemptive

Get a Gantt chart for the following processes for shortest job first *preemptive* scheduling. Do not click yet. Work on paper and wait for the next slide.

Process	Arrival Time	Burst time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Shortest Job First Preemptive

Q1.

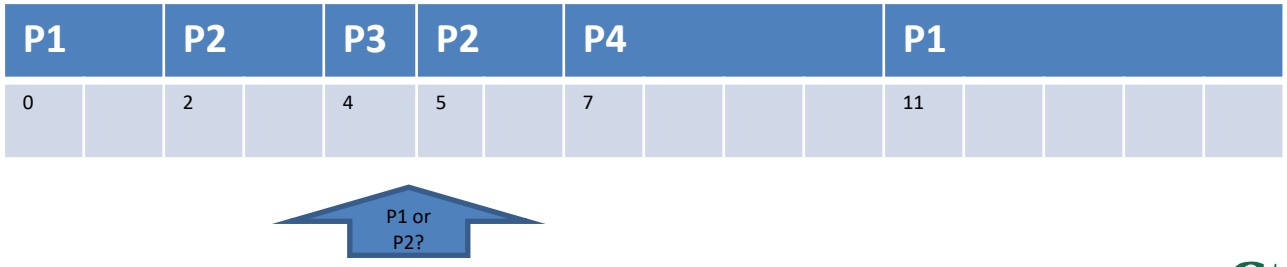
Does your Gantt chart look like this?

A. Yes

B. No.

C. I don't have a Gantt chart

Process	Arrival Time	Burst time
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Round Robin Scheduling

- The following processes are being scheduled using round-robin. **Obtain a Gantt chart for time quantum =2**
- (Get first 4 processes. More if you can).
- Assume that an incoming process enters the ready queue before a preempted process *at the same time*.

PID	Arrival Time	Burst Time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

Round Robin q =2

Q2.

My Gantt chart (first four) looks like this below.

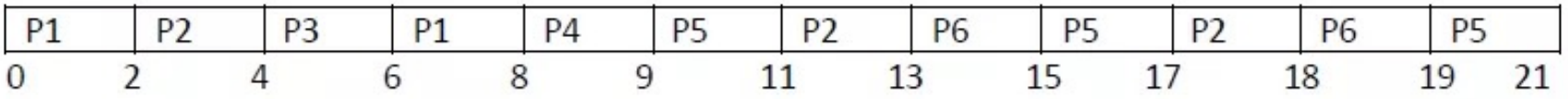
A. Yes.

B. No.

C. I don't have a Gantt chart.

PID	Arrival Time	Burst Time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

Grant Chart



P3 gets in before P1

Multilevel Feedback Queue

Q3.

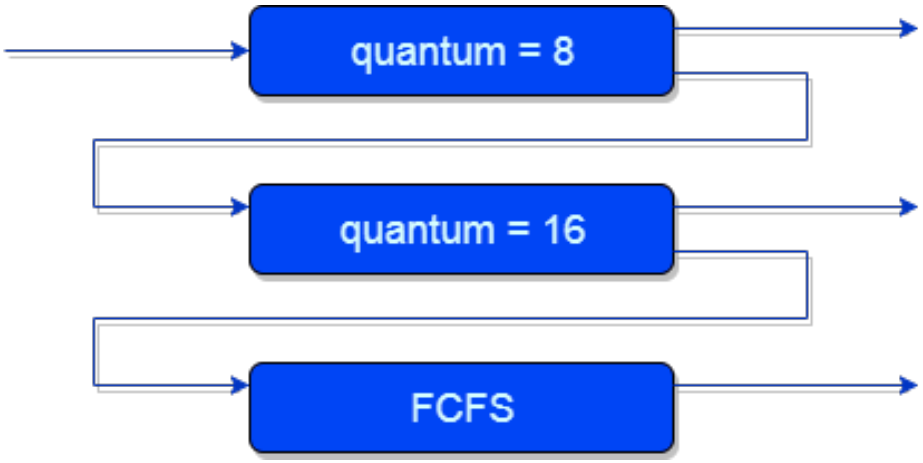
A process burst is 20 milliseconds. How long will it execute at the lowest level?

A. 4 ms

B. 0 ms

C. 20 ms

D. I don't know.



Answers

Shortest Job First Preemptive

Q1.

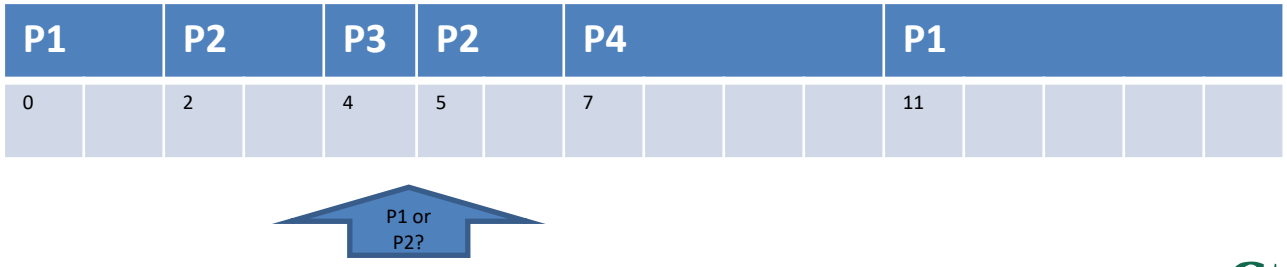
Does your Gantt chart look like this?

A. Yes

B. No.

C. I don't have a Gantt chart

Process	Arrival Time	Burst time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

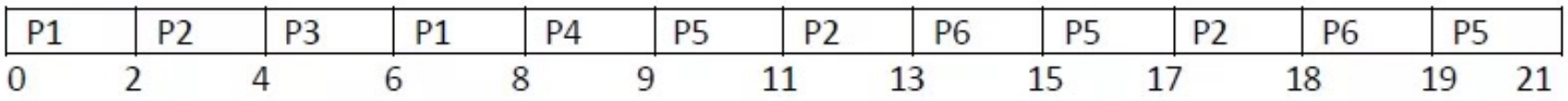


Round Robin Scheduling

Time 1: P2 arrives, gets in RQ.
Time 2: P2 starts.
P3 arrives, gets in RQ, P1 gets in RQ. RQ={P1, P3}
Time 3:P2 executing.
P4 arrives, gets in RQ, RQ={P4, P1, P3}
Time 4: P3 starts.
P5 arrives, gets in RQ, P2 gets in RQ. RQ={P2, P5, P4,P1}
Time 5: no change
Time 6: P1 starts.
P6 arrives, gets in RQ, P3 done. RQ={P6, P2, P5, P4}
Time 8: P4 starts
RQ={P6, P2, P5}
Time 9: P4 done, P5 starts
RQ={P6, P2}
T1me 11: P2 starts.
RQ={P5, P6} .....

PID	Arrival Time	Burst Time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

Grant Chart



Round Robin Scheduling

Q4.

My Gantt chart looks like this below.

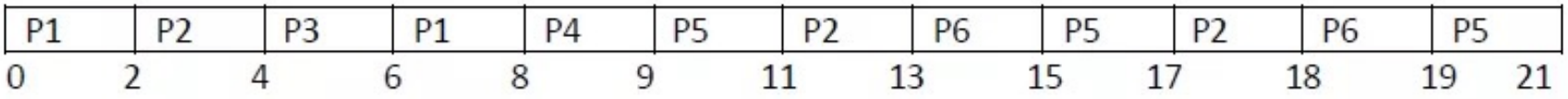
A. Yes.

B. No.

C. I don't have a Gantt chart.

PID	Arrival Time	Burst Time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

Grant Chart



Multilevel Feedback Queue

answer.

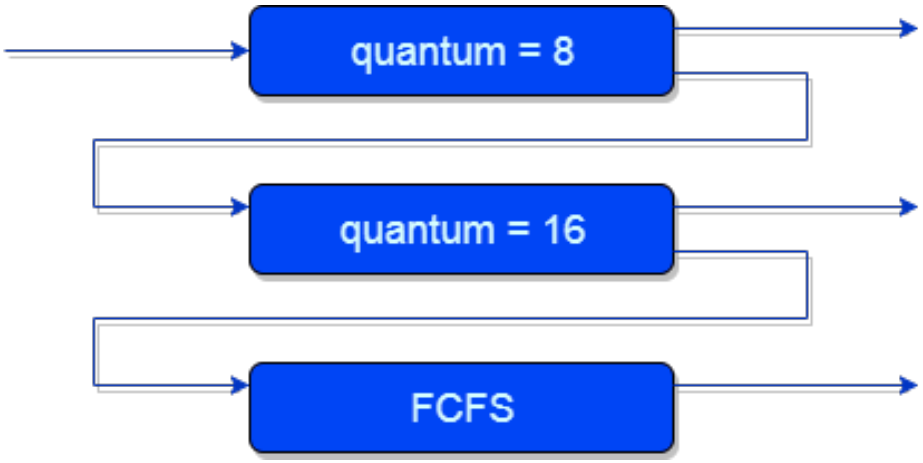
A process burst is 20 milliseconds. How long will it execute at the lowest level?

A. 4 ms

B. 0. ms

C. 20 ms

D. I don't know.



CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Back from IC Q.

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- **Assume Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – individual processors can be dedicated to specific tasks at design time
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, choices
  - all processes in common ready queue, **or**
  - each has its own private queue of ready processes
    - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running because of info in cache
  - **soft affinity: try but no guarantee**
  - **hard affinity can specify processor sets**

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor
  - Combination of push/pull may be used.

Real-Time CPU Scheduling

- Can present obvious challenges
  - **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
  - **Hard real-time systems** – task must be serviced by its deadline
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
  - **periodic** ones require CPU at constant intervals

RTOS: real-time OS. QNX in automotive, FreeRTOS etc.

Virtualization and Scheduling

- Virtualization software schedules multiple guests OSs onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn't own the CPUs
  - Can affect time-of-day clocks in guests
- Virtual Machine Monitor has its own scheduler
- Various approaches have been used
  - Workload aware, Guest OS cooperation, etc.

Algorithm Evaluation

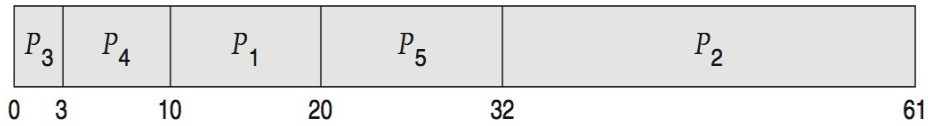
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

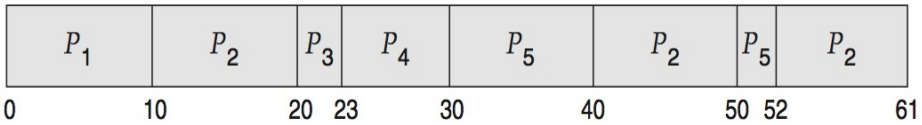
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:

  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Probabilitistic Models

- Assume that the arrival of processes, and CPU and I/O bursts are random
  - Repeat deterministic evaluation for many random cases and then average
- Approaches:
  - Analytical: Queuing models
  - Simulation: simulate using realistic assumptions

Queueing Models

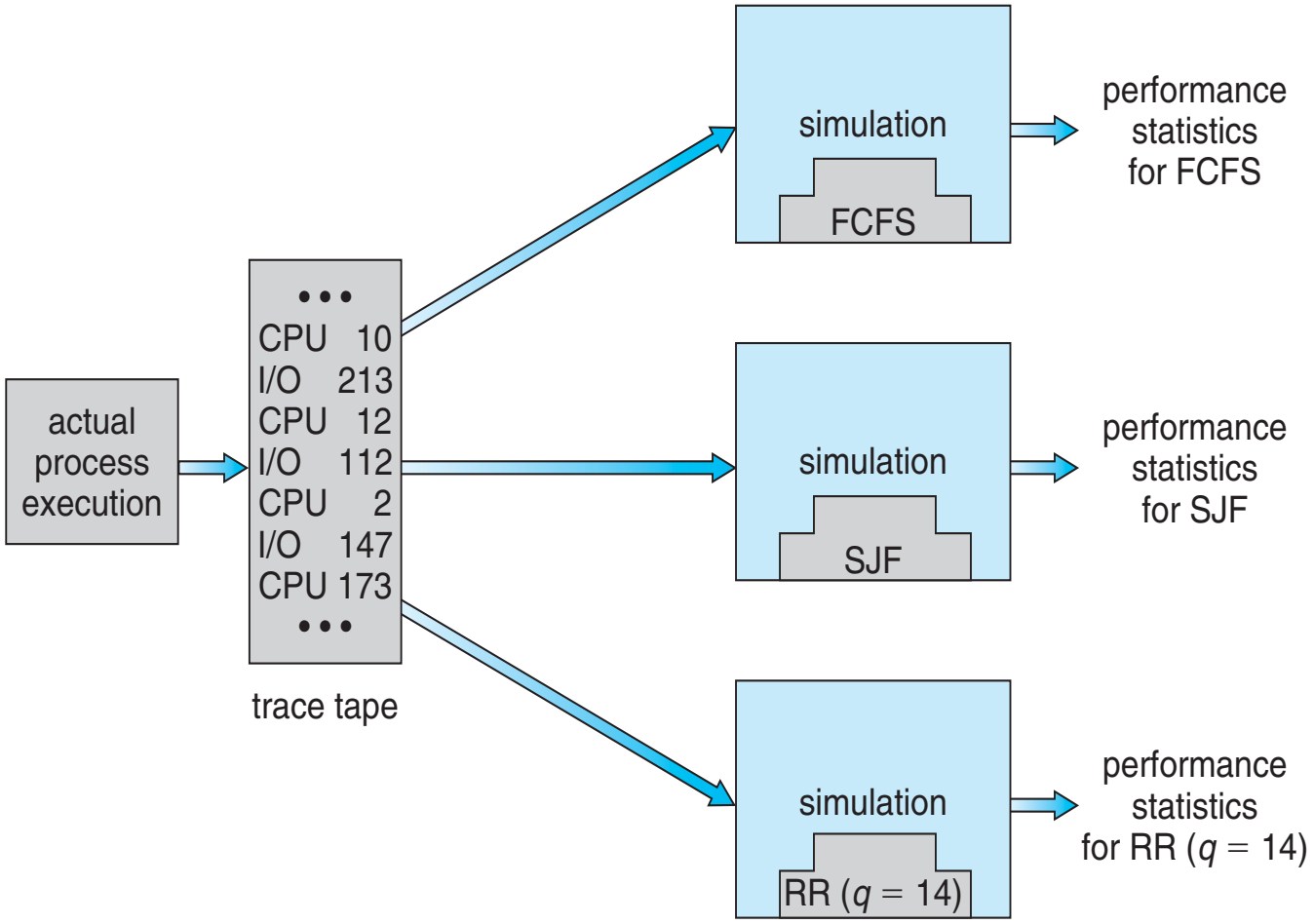
- Describes the arrival of processes, and CPU and I/O bursts probabilistically mathematically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Queueing Theory

Simulations

- Queueing models limited
- **Simulations** more versatile
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems
  - Illustration

Evaluation of CPU Schedulers by Simulation



Actual Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Considerations
  - Most flexible schedulers can be modified per-site or per-system
  - Or APIs to modify priorities
  - Environments can vary

CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Synchronization

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

Process Synchronization: Objectives

- Concept of process synchronization.
- The critical-section problem, whose solutions can be used to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
- Classical process-synchronization problems
- Tools that are used to solve process synchronization problems

Process Synchronization

EW Dijkstra *Go To Statement Considered Harmful*

Process Synchronization

Overview

- We synchronization is needed
- Critical section: access controlled to permit just one process
  - How the critical section be implemented
  - Mutex locks and semaphores
- Classic synchronization problems
- Will a solution cause a deadlock?

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	Look in fridge. Out of milk.
12:40	Arrive at store.	Leave for store
12:45	Buy milk.	Arrive at store.
12:50	Arrive home, put milk away.	Buy milk
12:55		Arrive home, put milk away. Oh no!

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- **Illustration**: we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
  - have an integer **`counter`** that keeps track of the number of full buffers.
  - Initially, **`counter`** is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - decremented by the consumer after it consumes a buffer.

  Will it work without any problems?

Consumer-producer problem

Producer

```
while (true) {
        /* produce an item*/
  while (counter == BUFFER_SIZE) ;
                /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
  while (counter == 0);
        /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZ
   counter--;
        /* consume the item in
        next consumed */
}
```

They run "concurrently" (or in parallel), and are subject to context switches at unpredictable times.

In, out: indices of empty and filled items in the buffer.

36

Race Condition

`counter++` could be compiled as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be compiled as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

They run concurrently, and are subject to context switches at unpredictable times.

Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Overwrites!

Critical Section Problem

We saw race condition between counter ++ and counter –

Solution to the "*race condition*" problem: critical section

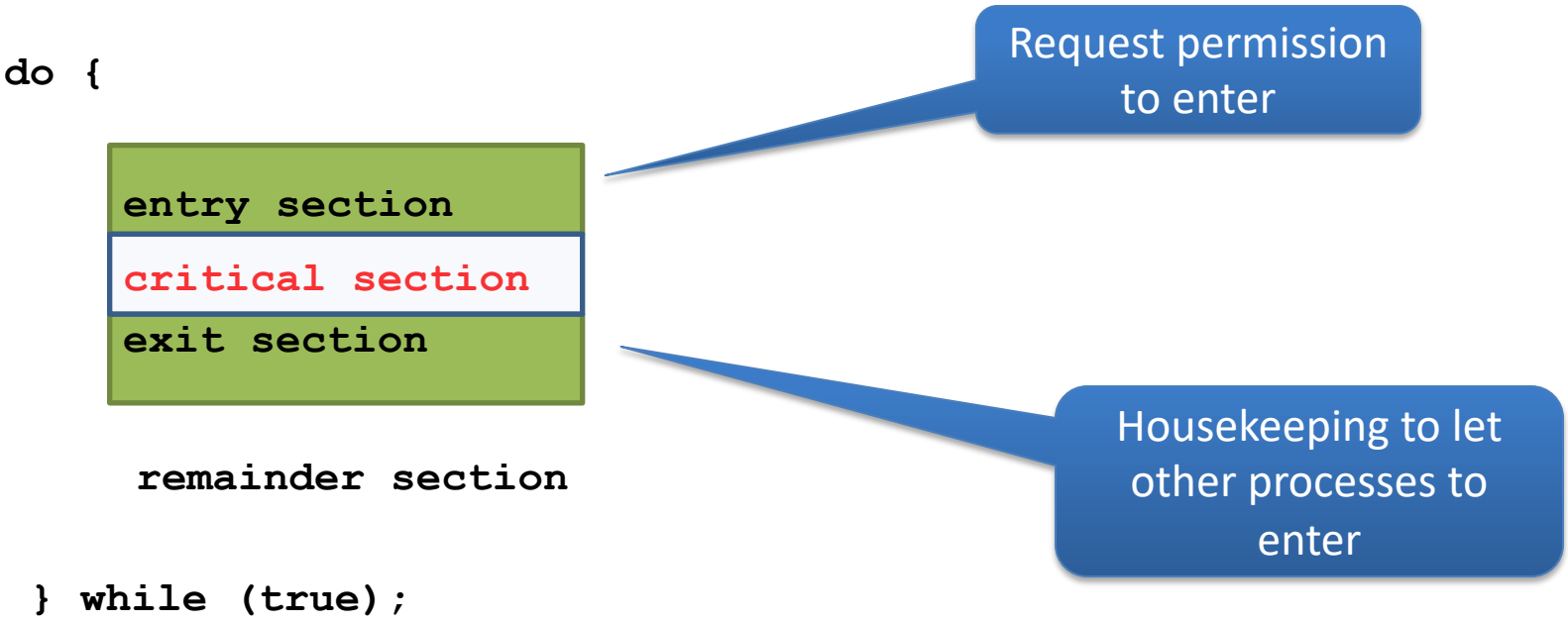
- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section follows.**

Race condition: when outcome depends on timing/order that is not predictable

Process Synchronization: Outline

- Critical-section problem to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
  - Peterson's solution
  - Atomic instructions
  - Mutex locks and semaphores
- Classical process-synchronization problems
  - Bounded buffer, Readers Writers, Dining Philosophers
- Another approach: Monitors

General structure: Critical section



A process is prohibited from entering the critical section while another process is in it.
Multiple processes are trying to enter the critical section concurrently by executing the same code.

Solution to Critical-Section Problem

A good solution to the critical-section problem should have these attributes

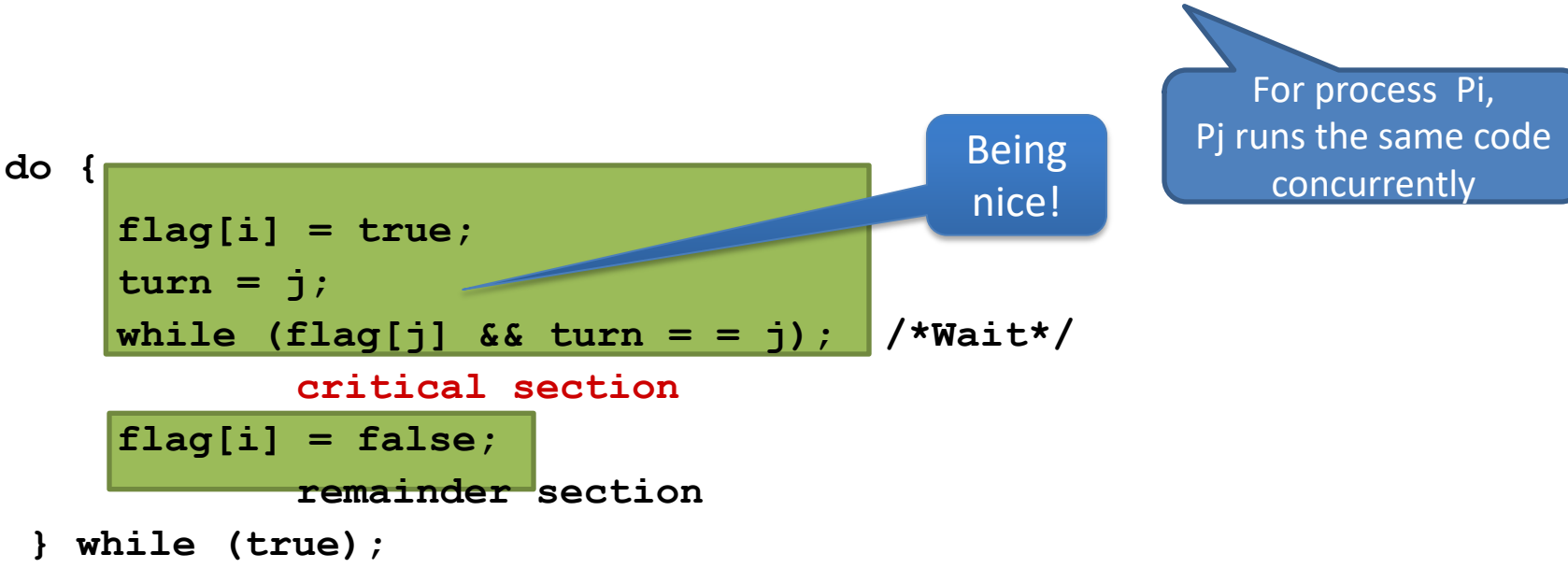
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - *If no process is executing in its critical section* and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the *number of times that other processes are allowed to enter their critical sections* after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution only
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
  - The variable **`turn`** indicates whose turn it is to enter the critical section
  - The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process $\mathbf{P_i}$ is ready to enter!

Algorithm for Process P_i



- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = *true*** implies that process P_i is ready!
- Note: Entry section- Critical section-Exist section
- These algorithms assume 2 or more processes are trying to get in the critical section.

Peterson's Solution (Cont.)

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **`flag[j] = false`** or **`turn = i`**

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met.

   A process waits only one turn.

Detailed proof in the text.

Note: there exists a generalization of Peterson's solution for more than 2 processes, but bounded waiting is not assured.

Synchronization: Hardware Support

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - test memory word and set value
  - swap contents of two memory words

Solution 1: using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
        while (test_and_set(&lock)) ; /* do nothing */

                /* critical section */

          …..
        lock = false;
                /* remainder section */
      …   ..

    } while (true);
```

To break out:
Return value of TestAndSet should be

FALSE

test_and_set(&lock) returns the lock value and then sets it to True .

Lock TRUE: locked, Lock FALSE: not locked.
If two TestAndSet() are attempted *simultaneously*, they will be executed *sequentially* in some arbitrary order

Solution 2: Swap: Hardware implementation

Remember this C code?

```
void Swap(boolean *a, boolean *b ) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Using Swap (concurrently executed by both)

```
do {
    key = TRUE;
    while (key == TRUE) {
       Swap(&lock, &key)
       }

    critical section

    lock = FALSE;

    remainder section
} while (TRUE);
```

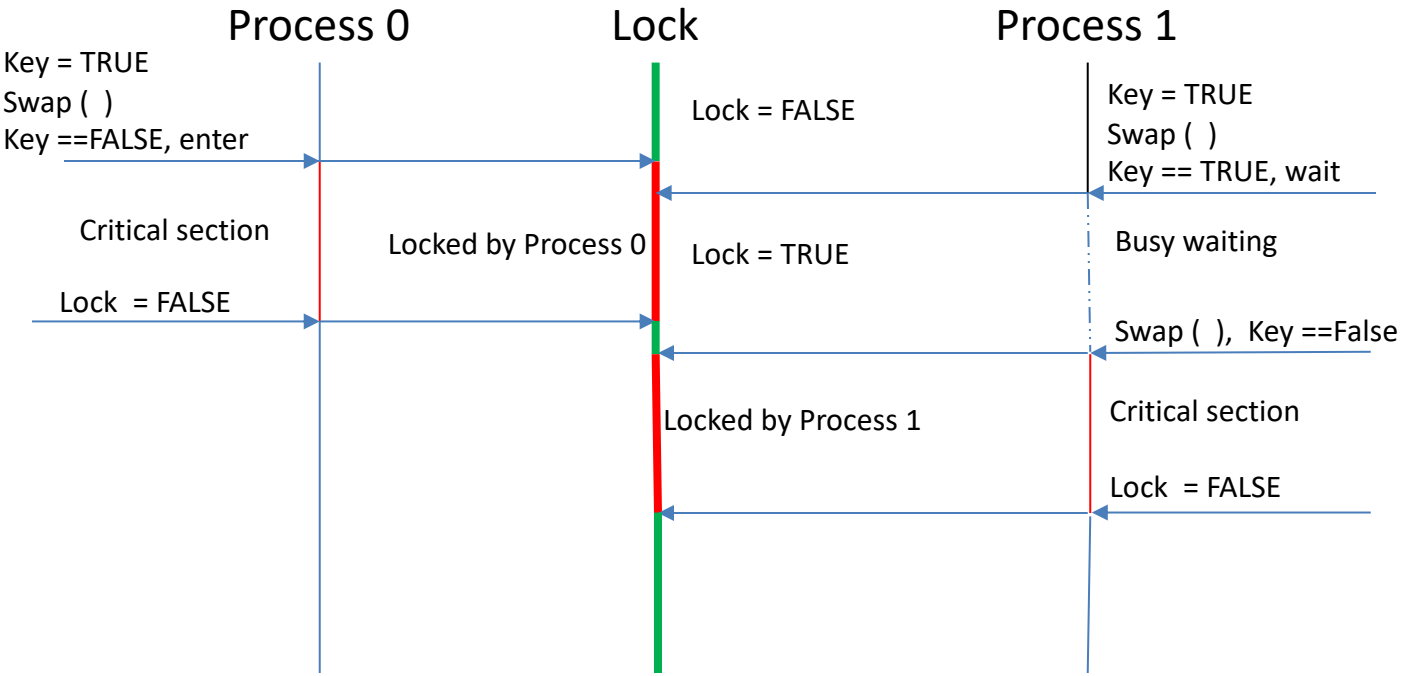
Lock is a SHARED variable.
Key is a variable local to the process.

Lock == false when no process is in critical section.

Cannot enter critical section UNLESS lock == FALSE *by other process or initially*

If two Swap() are executed simultaneously, they will be executed sequentially in some arbitrary order

Swap()



Note: I created this to visualize the mechanism. It is not in the book. - Yashwant

Bounded-waiting Mutual Exclusion with test_and_set

```
For process i:
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

Shared Data structures initialized to FALSE

- `boolean waiting[n];` Pr n wants to enter
- `boolean lock;`

The entry section for process i :

- First process to execute TestAndSet will find key == false ; ENTER critical section,
- EVERYONE else must wait

The exit section for process i:

Attempts to finding a suitable waiting process j (while loop) and enable it,

or if there is no suitable process, make lock FALSE.

Bounded-waiting Mutual Exclusion with test_and_set

The previous algorithm satisfies the three requirements

- **Mutual Exclusion**: The first process to execute TestAndSet(lock) when lock is false, will set lock to true so no other process can enter the CS.
- **Progress**: When a process i exits the CS, it either sets lock to false, or waiting[i] to false (allowing j to get in) , allowing the next process to proceed.
- **Bounded Waiting**: When a process exits the CS, it examines all the other processes in the waiting array in a circular order. Any process waiting for CS will have to wait at most n-1 turns