CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2024 L10

Scheduling, Synchronization

Slides based on

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

Tracking SRTF and RR

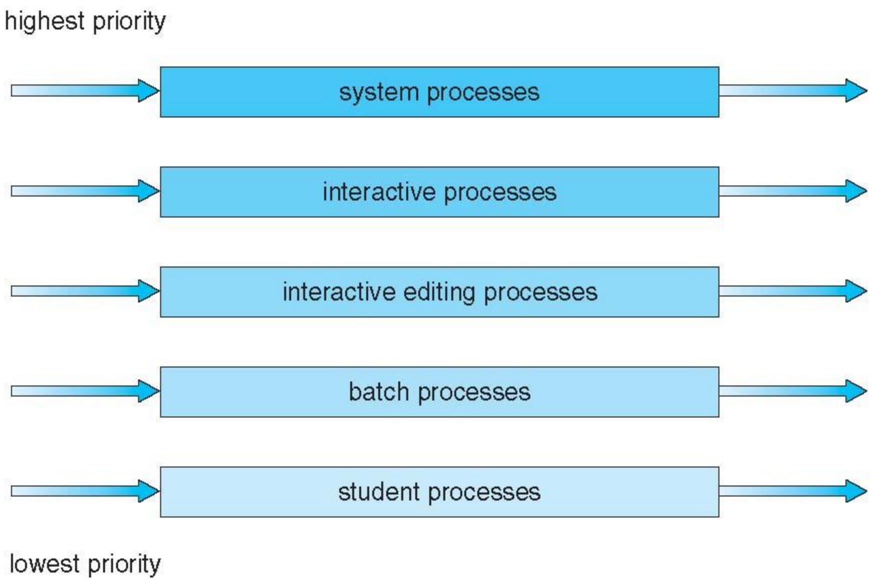
- Shortest remaining time first (Preemptive SJF)
  - Need to track the remaining time for all processes
- Round Robin
  - Need to track the position of the processes in the Ready Queue
  - Also need to track the remaining time needed
  - Illustration on youtube
  - Animation CPU Scheduling Algorithm 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.

Project

- See 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, 9/24/24
  - D2. Progress report, 10/31/24
  - D3. Slides and final reports, 11/20/24
  - D4. Presentations/demos 12/2-12/5 as arranged
  - D5: Peer Reviews due 12/6/24
- Teams: 2-3 students (see Teams channel “Project Teams”).

Multilevel Queue Scheduling



Real-time processes may have the highest priority.

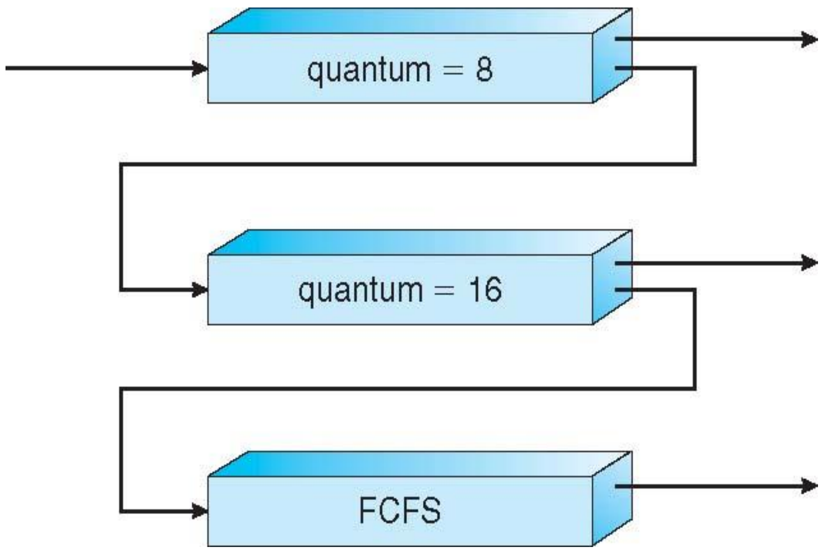
Multilevel *Feedback* Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service
  - Details at ARPACI-DUSSEAU

Inventor FJ Corbató won the Touring award!

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS (no time quantum limit)
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Upgrading may be based on aging. Periodically processes may be moved to the top level.

Variations of the scheme were used in earlier versions of Linux.

Completely fair scheduler Linux 2.6.23

Goal: fairness in dividing processor time to tasks (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).
  - Fewer processes waiting, they get more time each
- Queue ordered in terms of "virtual run time"
    - execution time on CPU added to value
  - smallest value picked for using CPU
  - small values: tasks have received less time on CPU
  - I/O bound tasks (shorter CPU bursts) will have smaller values
- *Balanced (red-black) tree* to implement a ready queue;
  - Efficient. O(log n) insert or delete time
- Priorities (*niceness*) cause different decays of values: higher priority processes get to run for longer time
  - virtual run time is the weighted run-time

Scheduling schemes have continued to evolve with continuing research. A comparison.

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.

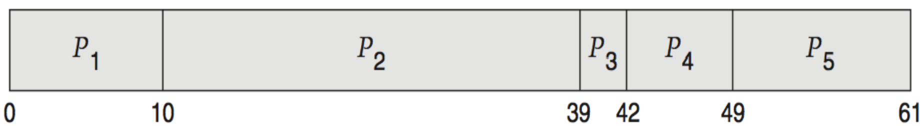
Algorithm Evaluation

- 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:

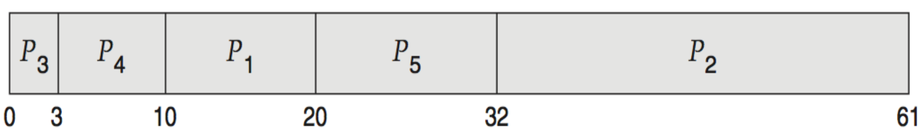
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

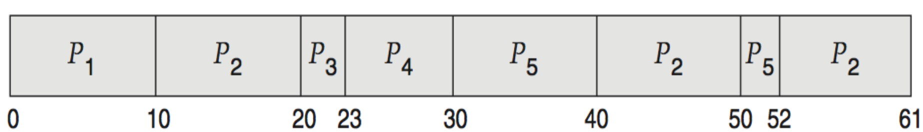
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - 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

Colorado State University

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

Little's Formula for av Queue Length

- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:
  - n = average queue length
  - W = average waiting time in queue
  - λ = average arrival rate into queue

$$n = \lambda \times W$$

Each process takes $1/\lambda$ time to move one position.
Beginning to end delay $W = n \times (1/\lambda)$

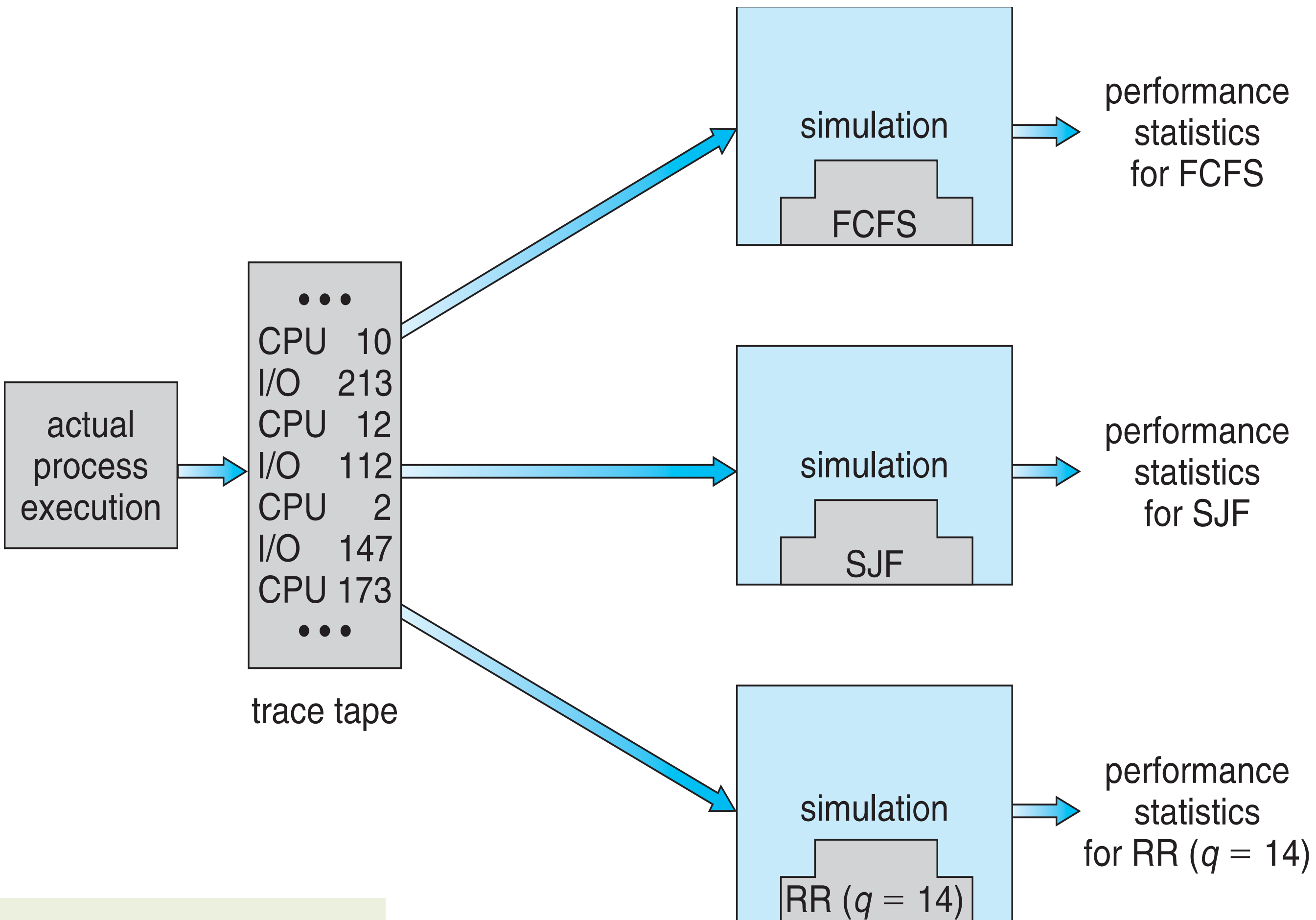
  - Valid for any scheduling algorithm and arrival distribution
- Example: average 7 processes arrive per sec, and 14 processes in queue,
  - then average wait time per process $W = n/\lambda = 14/7 = 2$ sec

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

Colorado State University

Evaluation of CPU Schedulers by Simulation



Simulation using real data

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]*

Colorado State University

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.

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 {$p_0, p_1, \ldots p_{n-1}$}
- 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

```
do {

    entry section
    critical section
    exit section

     remainder section

 } while (true);
```

Request permission to enter

Housekeeping to let other processes to enter

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 P_i is ready to enter!

Algorithm for Process P_i

For process Pi,
Pj runs the same code concurrently

Being nice!

```
do {
      flag[i] = true;
      turn = j;
      while (flag[j] && turn = = j);  /*Wait*/
              critical section
      flag[i] = false;
              remainder section
  } while (true);
```

- 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!
- 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

   If a process wants to enter, it only has to wait until the other finishes.

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. May not work in multiple processor systems, turn may be modified by by both processors.

Synchronization: Hardware Support

- Modern 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
  - Other

Solution 1: using test_and_set()

Lock TRUE: locked, Lock FALSE: not locked. Lock is a shared variable.
test_and_set(&lock) returns the lock value and then sets it to True .

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

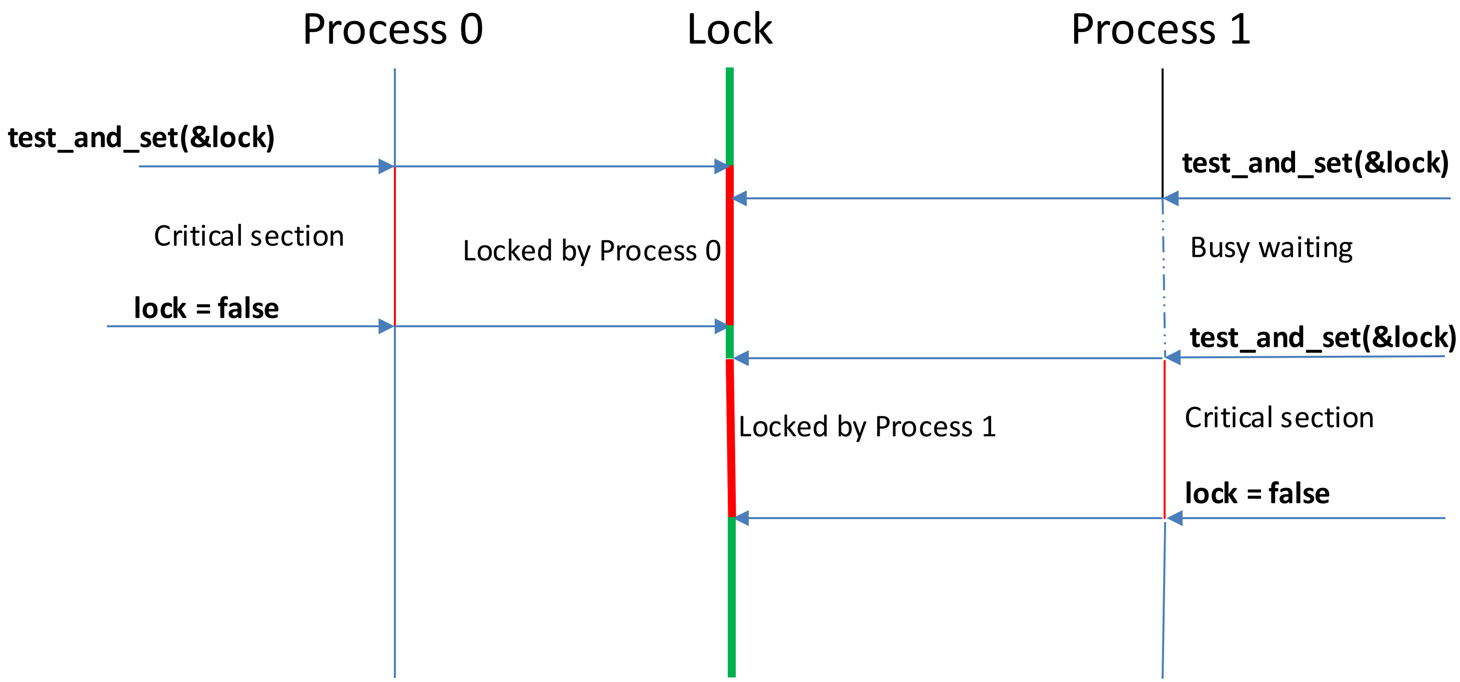
```
do {
     while (test_and_set(&lock)) ; /* do nothing */

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

  } while (true);
```

To break out:
Return value of TestAndSet should be FALSE

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

test_and_set(&lock)

Shared variable lock is initially FALSE




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

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

Solution 2: Swap: Hardware implementation

Another way of sensing/setting the lock (next slide).

Background: 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);
```

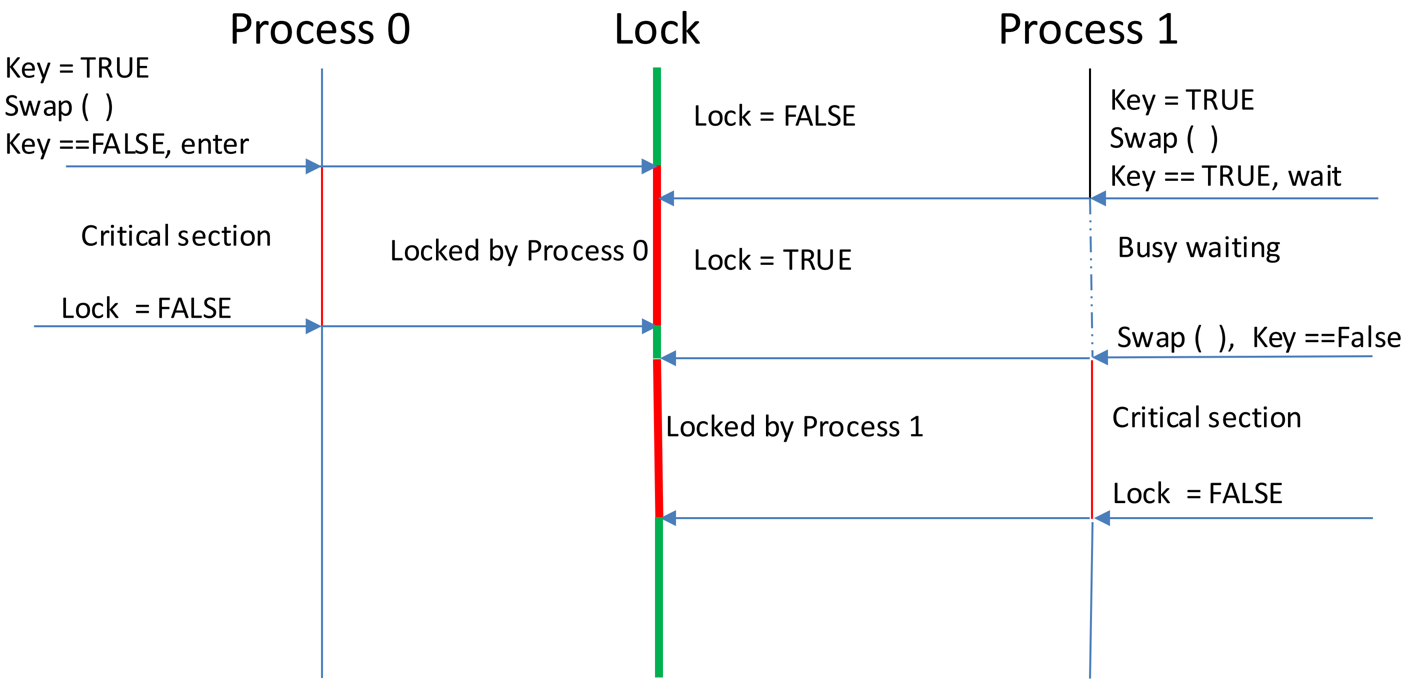
Lock is a SHARED variable.
Key is a variable local to the process.

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

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Colorado State University

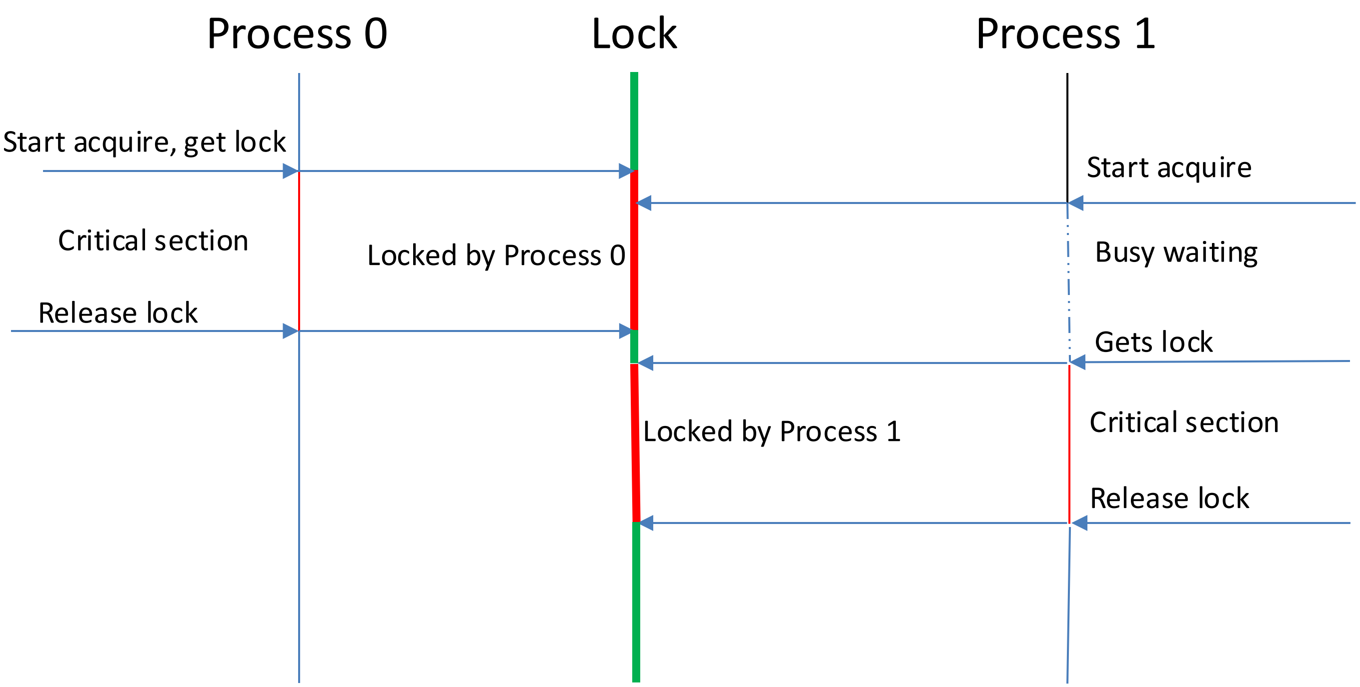
acquire() and release()

`acquire() {`
`    while (!available)`
`      ; /* busy wait */`
	`release() {`
	`    available = true;`
	`  }`

- **Usage**

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

acquire() and release()



How are locks supported by hardware?

- Atomic read-modify-write
- Atomic instructions in x86
  - LOCK instruction prefix, which applies to an instruction does a read-modify-write on memory (INC, XCHG, CMPXCHG etc)
  - Ex: lock cmpxchg <dest>, <source>
- In RISK processors? Instruction-pairs
  - LL (Load Linked Word), SC (Store Conditional Word) instructions in MIPS
  - LDREX, STREX in ARM
  - Creates an atomic sequence