CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2016 Lecture 15

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

We are here

- Major Scheduling algorithms seen.
- Continue a little more discussion of scheduling, then on to Synchronization
- Note: Quiz and Written Homework

- Information on Grading
  - Midterm
  - Final

Questions from Last Time

- Waiting time? initial plus other chunks
- Thread scheduling vs process scheduling: same or different?
  - PCB vs TCB, process-contention scope vs. system- contention scope (SCS)
- Role of time quantum q in Round Robin?
- Round robin for multi-level queue- why?
- Multi-level: what after low priority processes are done?
- Scheduling algorithm in OSX? Multilevel feedback queue

Virtualization and Scheduling

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

Operating System Examples

- Solaris scheduling: 6 classes, Inverse relationship between priorities and time quantum
- Windows XP scheduling: 32 priority levels (real-time, not real-time levels)
- Linux scheduling: newer - Completely fair scheduler (CFS):
  - 140 priority levels
  - variable timeslice, number and priority of the tasks in the queue

- Approaches evolve.

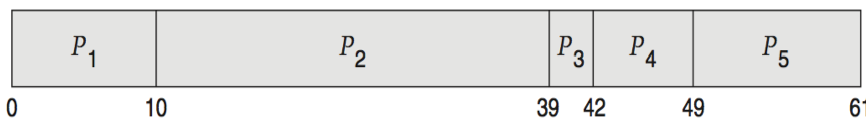
Algorithm Evaluation

- 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:

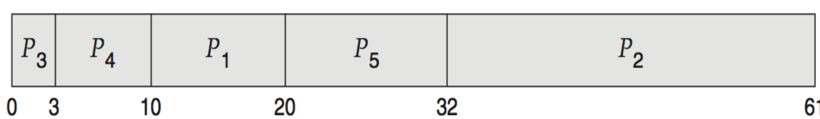
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

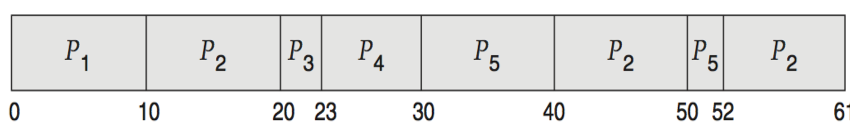
- For each algorithm, calculate minimum average waiting time
- Simple, *but requires exact numbers for input, applies only to those inputs*
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Colorado State University

Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - 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

Little's Formula

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

  $$n = \lambda \times W$$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more flexible
  - 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

Implementation

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

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

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.

15

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!

rado State University

Critical Section Problem

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**

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

Critical Section

- General structure of process P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Critical Section

```
do {

     entry section

          critical section

     exit section

          remainder section

 } while (true);
```

Algorithm for Process P_i

```
do {

     while (turn == j);
               critical section
     turn = j;
               remainder section
 } while (true);
```

Solution to Critical-Section Problem

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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

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!

Algorithm for Process P_i

For process Pi

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

Being nice!

- 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!

Peterson's Solution (Cont.)

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   $\mathbf{P_i}$ enters CS only if:

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

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

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

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words