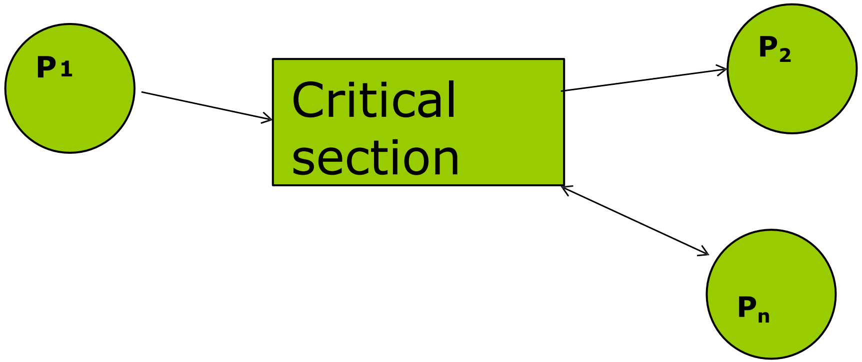
Operating Systems

Lecture 5: Process Synchronization

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



Critical Section Problem

- 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**

Critical Section

- General structure of process p_i is

```
do {
    entry section
        critical section
    exit section
        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

- 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

Synchronization Hardware

- Many systems provide hardware support for 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

Semaphore

- OS designers build software tools to solve critical section problem
- Simplest is **mutex** lock
- Semaphore ***S*** – integer variable
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while (S <= 0)
       ; // busy wait S--;     }

                signal (S) {
                    S++;   }
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
- **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
  - Consider P_1 and P_2 that require S_1 to happen before S_2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0

```
wait(S);
wait(Q);
.
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
.
signal(Q);
signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended

Priority Inversion

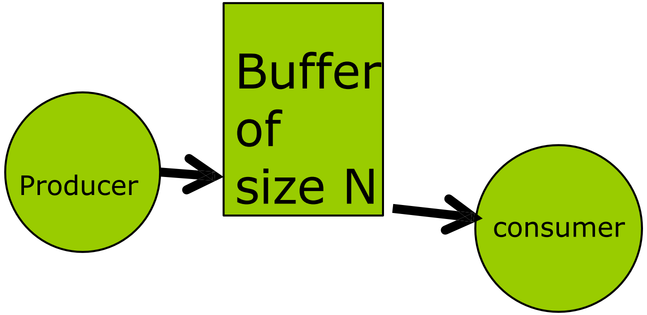
- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- **As an example**, assume we have three processes—*L*, *M*, and *H*—whose priorities follow the order $L < M < H$.
- Assume that process *H* requires resource *R*, which is currently being accessed by process *L*.
- Now suppose that process *M* becomes runnable, thereby preempting process *L*.
- Indirectly, a process with a lower priority—process *M*—has affected how long process *H* must wait for *L* to relinquish resource *R*.

Solved via **priority-inheritance protocol**

- All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values.

Bounded-Buffer Problem

- ***N*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value N



The producer/consumer process

The producer

```
do {
   ...
/* produce an item in
next_produced */
    ...
wait(empty);
wait(mutex);
    ...
/* add next produced to
the buffer */
    ...
signal(mutex);
signal(full);
} while (true);
```

The consumer

```
do {
   wait(full);
   wait(mutex);
      ...
/* remove an item from
buffer to next_consumed */
      ...
   signal(mutex);
   signal(empty);
      ...
/* consume the item in
next consumed */
      ...
} while (true);
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

End of Chapter 5