BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

Process Synchronization

Notice: The slides for this lecture have been largely based on those accompanying an earlier edition of the course text *Operating Systems Concepts with Java*, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.

Race Condition

A **race** occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y.

Races usually occurs because programmers assume that threads will take some particular trajectory through the execution space, forgetting the golden rule that **threaded programs must work correctly for any feasible trajectory**.

Computer Systems
A Programmer's Perspective
Randal Bryant and David O'Hallaron

The Critical-Section Problem

Solution

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.)

Algorithm 3

```
public class Algorithm_3 implements MutualExclusion
{
    private volatile boolean flag[2];
    private volatile int turn;
    public Algorithm_3() {
      flag[0] = false;
      flag[1] = false;
      turn = TURN_0;
     }
    // Continued on Next Slide
```

Algorithm 3 – cont'd

```
public void enteringCriticalSection(int t) {
  int other = 1 - t;

  flag[t] = true;
  turn = other;
  while(flag[other] && turn == other)
        Thread.yield();
}


public void leavingCriticalSection(int t) {
      flag[t] = false;
 }
}
```

Synchronization Hardware

- Many systems provide hardware support for critical section code.
- 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:
  - **boolean getAndSet(boolean b)**
  - **void swap(boolean b)**

Semaphore as General Synchronization Tool

- **Counting semaphore** – integer value can range over an unrestricted domain.
- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement (also known as **mutex** locks).
- Note that one can implement a counting semaphore *S* as a binary semaphore.
- Provides **mutual exclusion**:

```
Semaphore S(1); // initialized to 1

acquire(S);
criticalSection();
release(S);
```

Semaphore Implementation

```
acquire(S) {
    value--;
    if (value < 0) {
        add this process to list
        block;
    }
}
```

```
release(S) {
    value++;
    if (value <= 0) {
        remove some process P
          from list
        wakeup(P);
    }
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute acquire() and release() on the same semaphore at the same time.

- The implementation becomes the critical section problem:
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Applications may spend lots of time in critical section

Monitor

- Semaphores are low-level synchronization resources.
- A programmer's honest mistake can compromise the entire system (well, that is almost always true). We should want a solution that reduces risk.
- The solution can take the shape of high-level language constructs, as the **monitor** type:

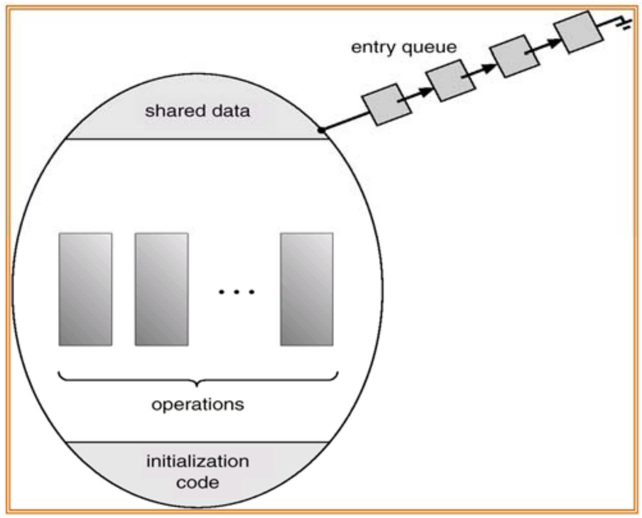
```
monitor monitor-name
{
   // variable declarations
   public entry p1(…) {
         …
   }
   public entry p2(…) {
         …
   }
}
```

A procedure within a monitor can access only local variables defined within the monitor.

There cannot be concurrent access to procedures within the monitor (only one thread can be ***active*** in the monitor at any given time).

Condition variables: queues are associated with variables. Primitives for synchronization are **wait** and **signal**.

Monitor



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

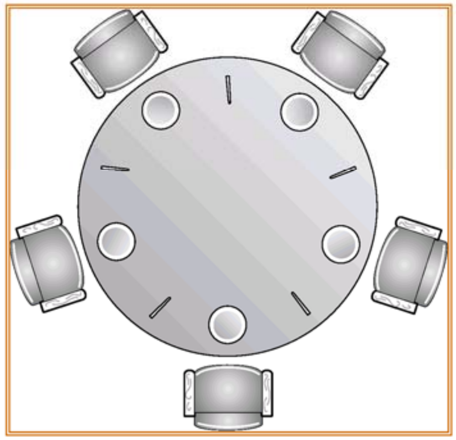
```
acquire(S);
acquire(Q);
 .
 .
 .
release(S);
release(Q);
```

P_1

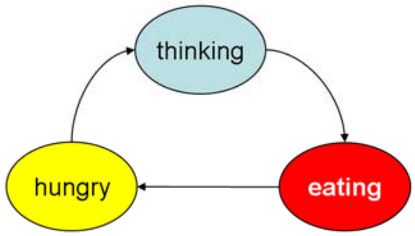
```
acquire(Q);
acquire(S);
 .
 .
 .
release(Q);
release(S);
```

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

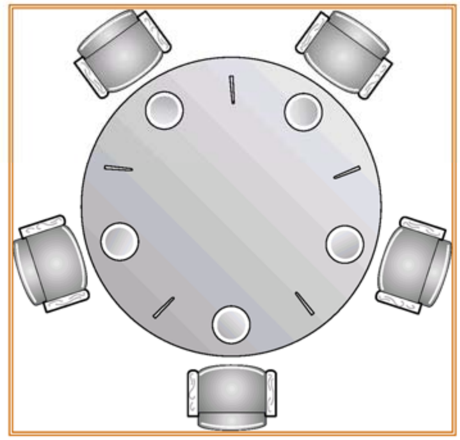
The *Dining-Philosophers* Problem



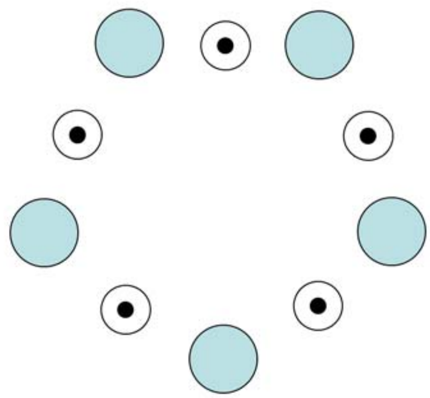
The *Dining-Philosophers* Problem



State diagram for a philosopher

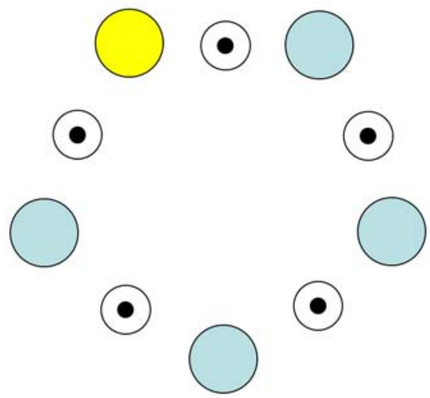


The *Dining-Philosophers* Problem

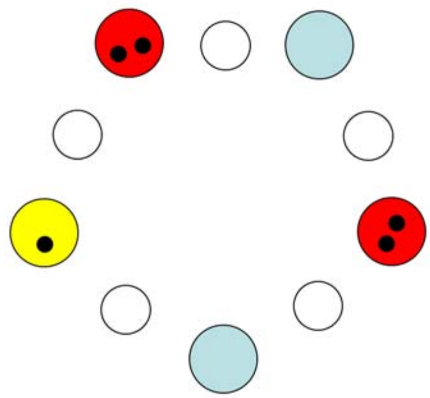


02/17/2010 CSCI 315 Operating Systems Design 15

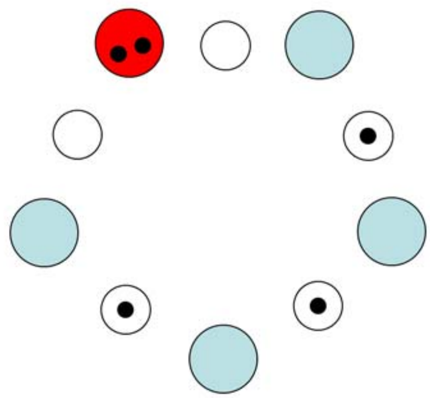
The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem

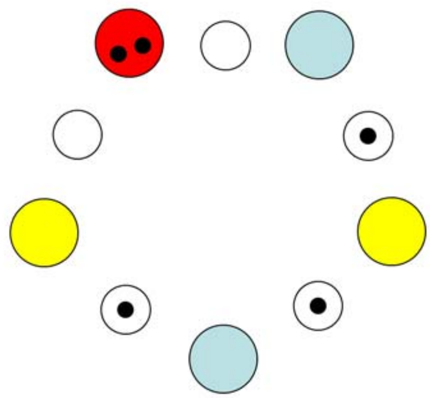


The *Dining-Philosophers* Problem



02/17/2010 CSCI 315 Operating Systems Design 18

The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem

<u>Question:</u> How many philosophers can eat at once? How can we generalize this answer for n philosophers and n chopsticks?

<u>Question:</u> What happens if the programmer initializes the semaphores incorrectly? (Say, two semaphores start out a zero instead of one.)

<u>Question:</u> How can we formulate a solution to the problem so that there is no deadlock or starvation?