Bucknell UNIVERSITY

Synchronization

CSCI 315 Operating Systems Design
Department of Computer Science

Notice: The slides for this lecture have been largely based on those from an earlier edition of the course text *Operating Systems Concepts, 8th ed.*, 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 Synchronization Problem

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the "orderly" execution of cooperating processes.

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.)

Typical Process P_i

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


Peterson's Solution

```
int turn;
boolean flag[2];
```

```
do {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j);
      critical section
   flag[i] = FALSE;
      remainder section
} while (TRUE);
```

Using Locks

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


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:
  - Test memory word and set value.
  - Swap the contents of two memory words.

TestAndSet

```
boolean TestAndSet(boolean *target)
{
    boolean ret_val = *target;
    *target = TRUE;
    return ret_val;
}
```


Lock with TestAndSet

```
boolean lock = FALSE;

do {
  while (TestAndSet(&lock));
     critical section
  lock = FALSE;
     remainder section
} while (TRUE);
```


CompareAndSwap

```
int CompareAndSwap (int *value,
  int expected, int new_value){

  int temp = *value;
  if (*value == expected)
    *value = new_value;

  return temp;
}
```


Lock with CompareAndSwap

```
boolean lock = 0;

do {
  while(CompareAndSwap(&lock,0,1) != 0);
     critical section
  lock = 0;
     remainder section
} while (TRUE);
```

How are we meeting requirements?

Do the solutions above provide:

- Mutual exclusion?
- Progress?
- Bounded waiting?

Semaphores

- **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

wait(S); // or acquire(S) or P(S)
criticalSection();
signal(S); // or release(S) or V(P)
```


Semaphore Implementation

```
typedef struct {
  int value;
  struct process
  *list;
} semaphore;
```


Semaphore Implementation

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
        add process to S->list
        block();
  }
}
```

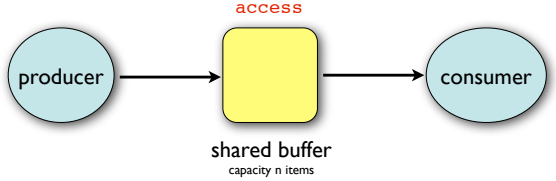
```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
        remove a process P
              from S->list
        wakeup(P);
  }
```


Semaphore Implementation

- Must guarantee that no two processes can execute **signal()** and **wait()** 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

The Bounded-Buffer Problem

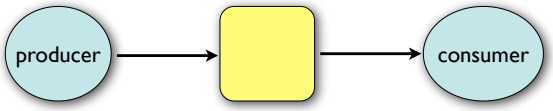
```
int n;
mutex access;       init(&access,1);
semaphore empty;    init(&empty,n);
semaphore full;     init(&full,0);
```



The Bounded-Buffer Problem

```
do {// produce item and save
     wait(&empty);
     wait(&access);
     // add item and save
     signal(&access);
     signal(&full);
} while (true);
```

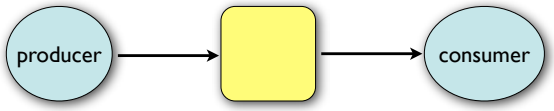
Producer



The Bounded-Buffer Problem

```
do {// produce item and save
     wait(&empty);
     wait(&access);
     // add item and save
     signal(&access);
     signal(&full);
} while (true);
```

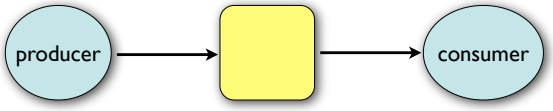
critical section



The Bounded-Buffer Problem

Consumer

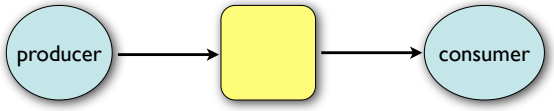
```
do {  wait(&full);
      wait(&access);
      // remove item and save
      signal(&access);
      signal(&empty);
      // consume save item
} while (true);
```



The Bounded-Buffer Problem

critical section

```
do {  wait(&full);
      wait(&access);
      // remove item and save
      signal(&access);
      signal(&empty);
      // consume save item
} while (true);
```



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:

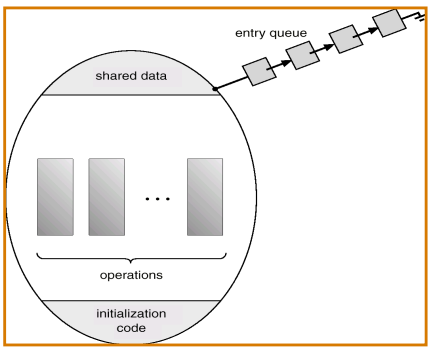
```
monitor mName {
  // shared variables
declaration
  procedure P1 (…) {
     …
  }
  procedure Pn (…) {
     …
  }
  init code (…) {
    ….
  }
}
```

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

There cannot be concurrent access to procedures within the monitor (only one process/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



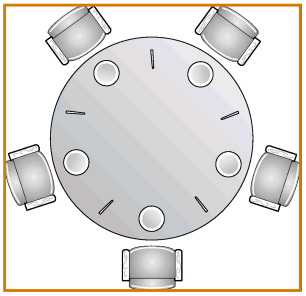
Deadlock and Starvation

- **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

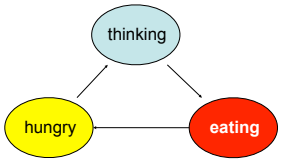
P_0	P_1
acquire(S);	acquire(Q);
acquire(Q);	acquire(S);
.	.
.	.
.	.
release(S);	release(Q);
release(Q);	release(S);

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

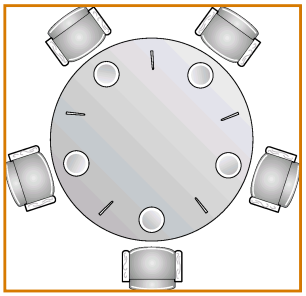
The *Dining-Philosophers* Problem



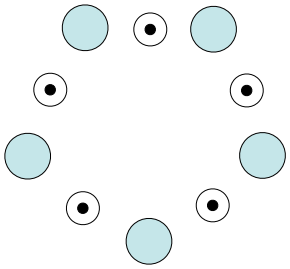
The *Dining-Philosophers* Problem



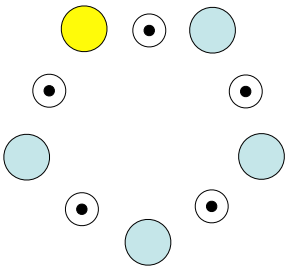
State diagram for a philosopher



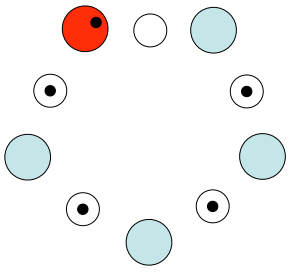
The *Dining-Philosophers* Problem



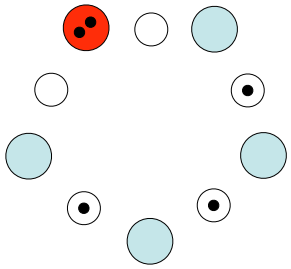
The *Dining-Philosophers* Problem



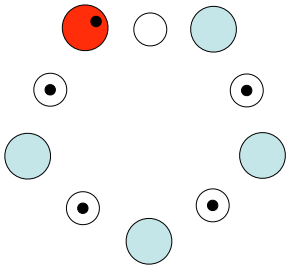
The *Dining-Philosophers* Problem



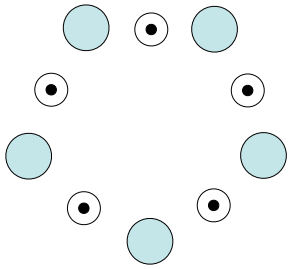
The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem



Limit to Concurrency

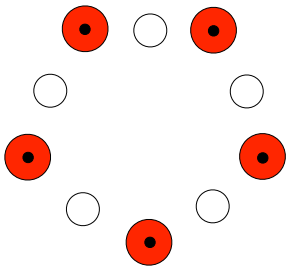
What is the maximum number of philosophers that can be eating at any point in time?

Philosopher's Behavior

- Grab chopstick on left
- Grab chopstick on right
- Eat
- Put down chopstick on right
- Put down chopstick on left

How well does this work?

The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem

Question: How many philosophers can eat at once? How can we generalize this answer for *n* philosophers and *n* chopsticks?

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

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