COSC 4600
Operating Systems Design

Spring 2019

Chapter 5: Synchronization Hardware

Inter-Process Communications: 2 ways

- Messages Passing – requires OS.
- Share Memory – doesn't take much OS work.
  - Need to synchronize concurrent accesses to shared data

Race Condition (1)

- Assuming A = 5, in shared memory of Process A and B.

Process A	Process B
(1) R1 = A	(1) R2 = A
(2) R1 = R1 + 1	(2) R2 = R2 -1
(3) A = R1	(3) A = R2

- What is the value of A?
  - 4, 5, or 6
- Inconsistency of shared data A

Race Condition (2)

- A race condition occurs when you have 2 or more processes sharing resources and what happens depends on the order that they run in.

- Why is it difficult to debug a race conditions?

Race Condition (3)

- Debugging Race Conditions is difficult
  - It depends on timing
- How to fix?
  - Prevent it
    - Using critical section, i.e., when a process is executing in its critical section, no other process is to be allowed to execute in its critical section.
  - Detect and rollback
    - Software transactional memory

Critical Section

- Ensure <u>no</u> two processes are <u>ever</u> in their shared critical section at the same time.
- No process should have to wait forever.
  - Avoid starvation/deadlock – 2 processes starve each other
- Potential Solutions?
  - hardware
  - software

Hardware Solution

Interrupt solution (could be hardware fix)

- Just before critical section, disable all interrupts
- Then run critical section of code
- Finally enable all interrupts
  - If you forget to re-enable interrupts, you are in big trouble.
  - This is a poor plan since it is not fair.
  - The OS can not turn interrupts back on, because it would interrupt the process.

Software Solution

- Attempt #1: Lock a shared variable in the shared resource.
- Set the lock to 0, means it is open and available
- Set the lock to 1, means it is locked and unavailable
- Problems:
    - The process could be interrupted before setting lock and after loop.

code:

Problems?

```
while (lock ==1) { };
lock = 1
    Critical code section
lock = 0
```

Hybrid (Hardware & Software) Solution TestAndndSet Instruction

- We need to have an “atomic” operation – one that can not be interrupted.

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

Solution using TestAndSet

- Solution:

lock is a shared boolean variable, initialized to false.

```
while (true) {
        while ( TestAndSet (&lock )) ;   // do nothing
                //    critical section
        lock = FALSE;
                //      remainder section
}
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
 {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
while (true)  {
        key = TRUE;
        while ( key == TRUE) Swap (&lock, &key );
                //    critical section
        lock = FALSE;
                //      remainder section
}
```

Any Performance Problem?

Spinlock

```
while (true) {
        while ( TestAndSet (&lock )) ;   // do nothing
            //    critical section
        lock = FALSE;
            //      remainder section
 }
```

```
while (true)  {
        key = TRUE;
        while ( key == TRUE) Swap (&lock, &key );
            //    critical section
        lock = FALSE;
            //      remainder section
 }
```

The Fix: Semaphore

- In 1965 Dijkstra invented semaphores to solve these problems.
- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
  - Provides a way to count the number of sleep/wakeups invoked
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Semantics of wait() and signal()

Semaphore as General Synchronization Tool

- Semaphore
  - 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
- Binary semaphore provides mutual exclusion

```
Semaphore S;    //  initialized to 1
wait (S);
  Critical Section
signal (S);
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting for each other.
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

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

Classical Problems

- Solutions using Semaphores
  - Producer-Consumer Problem
  - Dining-Philosophers Problem
  - Readers and Writers Problem

Producer-Consumer Problem

- A buffer can hold N items.
- When the buffer is not full, producer can put more items; otherwise, the producer will be blocked until the consumer removes some items.
- When the buffer is not empty, consumer can remove items; otherwise, the consumer will be blocked until producer puts some items.
- The buffer cannot be accessed by the producer and consumer simultaneously.

Bounded Buffer Problem (Cont.)

- The structure of the **producer** process

```
while (true)  {
    //   produce an item
    produce(item);
    wait (availableSlots);
    wait (mutex); //enter CR to add item.

     //  add the item to the  buffer

    signal (mutex); //leave CR system.
    signal (availableItems);
}
```

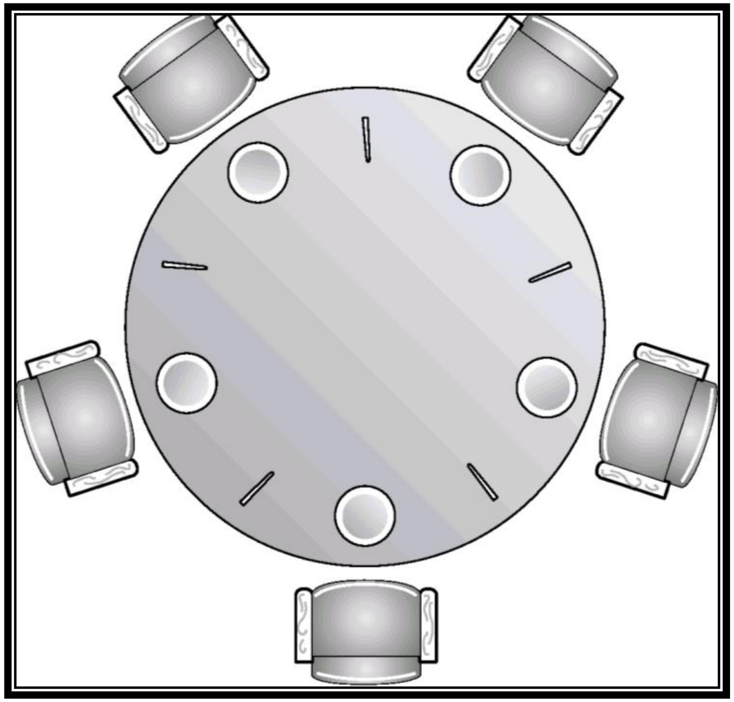
Bounded Buffer Problem (2)

- The structure of the consumer process

```
while (true) {
    wait (availableItems);
     wait (mutex);  //Enter CR
           //  remove an item from  buffer
    signal (mutex); //leave CR
    signal (availableSlots);
    //  consume the removed item
}
```

Dining-Philosophers Problem

- 5 philosophers
- 5 forks
- A philosopher needs two forks to eat



With Semaphores

```
semaphore S[5] = {1,1,1,1,1}   // init to 1
    while (1) {
            Wait(S[left]);
            Wait(S[right]);
            eat ()
            Signal(s[left]);
            Signal(s[right]);
    }
```

Problem:
If all philosophers pick up left fork 1st, then no right forks are available.

We need to be able to pick up both forks or none, so we need a semaphore to do this: make picking up forks atomic.

```
Philosphers(i)  {

    while(1) {

        take_forks(i);
        eat();
        put_forks(i);
    }  }
```

```
take_forks(i) {
        Wait(mutex);
        Wait(S[left]);
        Wait(S[right]);
        Signal(mutex);
}
```

```
put_forks(i) {
        Wait(mutex);
        Signal(S[left]);
        Signal(S(right]);
        Signal(mutex);
}
```

Problem:
If philosophers is blocked on a right or left fork, he will be deadlocked, because no other process can put_forks, since it is in it's critical section!

A Correct Solution

```
#define N 5 /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore s[N]; /* one per philosopher, all 0 */

void test(int i) {
    if ( state[i] == HUNGRY && state[LEFT(i)] != EATING &&
         state[RIGHT(i)] != EATING )
         {state[i] = EATING;
          V(s[i]);}
}
```

```
void get_forks(int i) {
   P(mutex);
   state[i] = HUNGRY;
   test(i);
   V(mutex);
   P(s[i]);
}
void put_forks(int i) {
   P(mutex);
   state[i]= THINKING;
   test(LEFT(i));
   test(RIGHT(i));
   V(mutex);
}
void philosopher(int process) {
   while(1) {
       think();
       get_forks(process);
       eat();
       put_forks(process);
    }
}
```

Readers/writers problem

Two types of processes

1. readers – read information from the database
2. writers – write to the database

- And we want concurrent access to the database.
  - Constraint-1: At most **one writer** should access the database at any time
  - Constraint-2: A writer and reader cannot access the database at the same time.

try #1

```
semaphore db = 1;
writer () {
    P(db);   // protect db from other writers and readers
    modify database
    V(db);
}
readers (){
    P(db);
    read database
    V(db);
}
```

no problem with constraints
but it excludes other readers

try #2

- we want to allow multiple readers, but no readers w/ writers

```
v = 1;
db =1;

writer()
    P(db)
    v = 0;
    modify(database)
    v = 1;
    V(db)

reader()
    if (v==1)
        read (database)
```

Problems?

Try #3

```
semaphore mutex
int rc =0;   //reader count
reader()
     P(mutex) // Protect rc
     if (rc ==0) {  P(db); }
     rc ++;
     V(mutex);

     read database;

     P(mutex)
     rc –
     if (rc ==0)  {  V(db); }
     V(mutex);
```

```
writer()
     P(db);
     modify database
     V(db);
```

Problems?

Problems

- This solution allows multi readers
  - no writer/reader combo, but writer will starve
    - may never get a chance if readers keep arriving. Writer unblocked only when all readers done.

- Can be done with higher level primitives to make it easier.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
            …

   procedure Pn (…) {……}

   Initialization code ( ….) { … }
            …
   }
}
```

Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Solution to Dining Phil

Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

dp.pickup (i)

EAT

dp.putdown (i)

```
monitor DP
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait();
    }

    void putdown (int i) {
          state[i] = THINKING;
                // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
     }

void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
              self[i].signal () ;
         }
 }

 initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
 }
}
```