CSC 453 Operating Systems

Lecture 6 : Process Synchronization

Concurrent Processes : An Example

- Imagine that you have two ***concurrent*** processes that manage a checking account:
  - p_1 handles deposits and other credits.
  - p_2 handles checks and other debits.
- They would both share the variable balance:
  `shared double balance;`
- They both can reference balance:
  - P1 contains:
    `balance = balance + deposit;`
  - P2 contains
    `balance = balance – check;`

Race Conditions

- While it looks like these processes recalculate the balance in a single step, this is NOT how it looks on the machine language (or assembler) level:

Process P_1	*Process P_2*
`load R1, balance`	`load R1, balance`
`load R2, amount`	`load R2, amount`
`add R1, R2`	`sub R1, R2`
`store R1, balance`	`store R1, balance`

What you really have is a ***race condition***

What Is a Race Condition?

- A race condition is a situation in which the results of the operating system is determined by the results of a race between competing activities.
- A race condition is highly undesirable because the result will be unpredictable and the integrity of data may be compromised.

Critical Sections

- A critical section is a section of code where the process is performing operations that must be ***atomic***, i. e., where the operations must be performed as a unit without interruption.
- There are four required criteria in implementing critical sections:
  - No 2 processes can be inside a critical section at once
  - No assumptions can be made about speed or number of processors.
  - No processes outside critical section can block another process.
  - No process should wait forever to enter a critical section.

Using Locks

```
shared boolean lock = FALSE;
shared double balance;
```

Process 1

```
/* Acquire lock */
while (lock) ;
lock = TRUE;
/* Execute critical section */
balance
 = balance+deposit;
/* Release lock */
lock = FALSE;
```

Process 2

```
/* Acquire lock */
while (lock) ;
lock = TRUE;
/* Execute critical section */
balance
  = balance-check;
/* Release lock */
lock = FALSE;
```

Mutual Exclusion

- Mutual Exclusion means that when one process has access to a critical section, all other processes are barred from entering it.
- We saw earlier that that was one of the required criteria for critical sections.
- How will we implement this?

Achieving Mutual Exclusion – First Try

```
int  turn = 0;
```

```
… …
while (turn != 0)
     ; /* wait */
/* critical
     section */
turn = 1;
… …
```

```
… …
while (turn != 1)
     ; /* wait */
/* critical
     section */
turn = 0;
… …
```

Achieving Mutual Exclusion – 2nd Try

```
enum boolean   { false, true };
int  flag[2] = { false, false};
```

```
… …
while (flag[1])
     ; /* wait */
flag[0] = true;
/* critical
     section */
flag[0] = false;
… …
```

```
… …
while (flag[0])
     ; /* wait */
flag[1] = true;
/* critical
     section */
flag[1] = false;
… …
```

Achieving Mutual Exclusion – 3rd Try

```
enum boolean   { false, true };
int  flag[2] = { false, false};
```

```
… …
flag[0] = true;
while (flag[1])
     ; /* wait */
/* critical
     section */
flag[0] = true;
… …
```

```
… …
flag[1] = true;
while (flag[0])
     ; /* wait */
/* critical
     section */
flag[1] = true;
… …
```

Achieving Mutual Exclusion – 4th Try

```
… …
flag[0] = true;
while (flag[1]) {
  flag[0] = false;
  /* delay */
  flag[0] = true;
}
/* critical
     section */
flag[0] = true;
… …
```

```
… …
flag[1] = true;
while (flag[0])  {
  flag[1] = false;
  /* delay */
  flag[1] = true;
}
/* critical
     section */
flag[1] = true;
… …
```

Dekker's Algorithm – Common Declarations

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

Dekker’s Algorithm – Process 0

```
void p0(void)
{
      flag[0] = true;
      while (flag[1])
            if (turn == 1)    {
                  flag[0] = false;
                  while (turn == 1)
                        ;
                  flag[0] = true;
            }
      /* critical section */
      turn = 1;
      flag[0] = false;
      /* rest of process */
}
```

Dekker’s Algorithm – Process 1

```
void p1(void)
{
      flag[1] = true;
      while (flag[0])
            if (turn == 0)    {
                  flag[1] = false;
                  while (turn == 0)
                        ;
                  flag[1] = true;
            }
      /* critical section */
      turn = 0;
      flag[1] = false;
      /* rest of process */
}
```

Peterson’s Algorithm - Declarations

```
#include  “prototypes.h”
#define FALSE  0
#define TRUE   1
#define N 2    /* number of processes */

int  turn;     /* whose turn is it? */
int  interested[N]; /* initially all
                       false */
```

Peterson’s Algorithm - Entering

```
void enter_region(int process)
/* process: who is entering 0 or 1) */
{
  int  other; /* # of other process */
  other = 1 – process;
  interested[process] = TRUE;
  turn = other;
  while (turn == other &&
          interested[other])
     ;
}
```

Peterson's Algorithm - Leaving

```
void leave_region(int process)
/* process: who is entering 0 or 1) */
{
  /* Indicate departure from critical
     section */
  interested[process] = FALSE;
}
```

Disabling Interrupts

- Since the sequence of instructions in either process can be interrupted, let's disable interrupts so the instructions of the critical section will proceed without interruption:
- `shared double balance;`

<u>Process 1</u>

```
disableInterrupts();
balance
     = balance + deposit;
enableInterrupts();
```

<u>Process 2</u>

```
disableInterrupts();
balance
     = balance - check;
enableInterrupts();
```

Test and Set Lock

- Test and Set is a single machine instruction introduced by IBM in its Series 360 computers.
  - A single bit stored the value of the lock as 0 (free) or 1 (busy).
  - If Process 0 tested the condition and found the lock free, it would set it and continue, clearing the lock when it leaves the critical section.
  - If it finds the lock set, it is placed in a loop where it continually tests the lock until it is cleared.
- This works well for a some number of processes but can lead to ***starvation***.

Implementing Test and Set Lock

```
enter_region:
        tsl     register, flag    ; copy flag to register
                                  ; and set flag to 1
        cmp     register, #0      ; was flag zero?
        jnz     enter_region      ; if nonzero, lock is set so loop
        ret                       ; return to caller, enter  critical
                                  ; section
leave_region:
        mov     flag, #0          ; store a 0 in flag
        ret                       ; return to caller
```

Semaphores

- Edsger Dijkstra introduced the concept of the semaphore in his landmark paper “Co-operating Sequential Processes” as a mechanism for coordinating processing that share resources (including critical sections).
- A semaphore ***s*** is a non-negative integer variable which is changed or tested exclusively by the primitives P and V.
  – V(s) : [s = s + 1]
  – P(s): [while s == 0 { wait }; s = s – 1]

Semaphores and Critical Sections

- Because the P and V operations are indivisible, they can be used to implement critical sections:

```
semaphore mutex = 1;
```

<u>Process 0</u>

```
… …
P(mutex)
/* critical
     section */
V(mutex)
… …
```

<u>Process 1</u>

```
… …
P(mutex)
/* critical
     section */
V(mutex)
… …
```

Implementing Semaphores

```
class     semaphore {
public:
     semaphore (int  v);
     void P();
     void V();
private:
     int  value;
}
```

Implementing Semaphore Constructor

```
semaphore::semaphore (int  v) {
     //allocate space for the semaphore
          object in the OS
     value = v;
}
```

Implementing Semaphore P Operation

```
void semaphore::P()
{
     disableInterrupts();
     //Loop until value is positive
     while (values == 0) {
          enableInterrupts();
          disableInterrupts();
     }
     --value;
     enableInterrupts();
}
```

Implementing Semaphore V Operation

```
void semaphore::V()
{    disableInterrupts();
     value++;
     enableInterrupts();
}
```

Consumer-Producer Problem – Semaphore Solution

```
semaphore mutex = 1, full = 0,
          empty = N;
buftype   buffer[N];
```

Consumer-Producer: Producer Process

```
producer()
{
     buftype   *next, *here;
     while (TRUE)   {
          produceItem(next);
          //Claim an empty buffer
          P(empty);
          // Manipulate the pool
          P(mutex);
            here = obtain(empty);
          V(mutex);
```

Producer Process (continued)

```
            copyBuffer(next, here);
            //Manipulate the pool
            P(Mutex);
              release(here, fullPool);
            V(mutex);
            // Signal a full buffer
            V(full);
        }
    }
```

Consumer-Producer: Consumer Process

```
consumer()
{
     buftype   *next, *here;
     while (TRUE)   {
          // Claim a full buffer
          P(full);
          // Manipulate the pool
          P(mutex);
            here = obtain(full);
          V(mutex);
          copyBuffer(here, next);
```

Consumer Process (continued)

```
        // Manipulate the pool
        P(mutex);
          release(here, emptyPool);
        V(mutex);
        // Signal an empty buffer
        V(empty);
        consumeItem(next);
    }
}
```

Reader-Writer Problem

- A data object, such as a file, is to be shared by several concurrent processes.
  - If one of these processes are reading, then any of the others can read, but they cannot write.
  - If one of these processes are writing, then no others can write ***OR*** read.
- There are two really problems:
  - No reader should wait for other readers to finish simply because a writer is waiting. (***This may starve writers.***)
  - If a writer is waiting, no new readers should start reading. (***This may starve readers.***)

The Writer Process

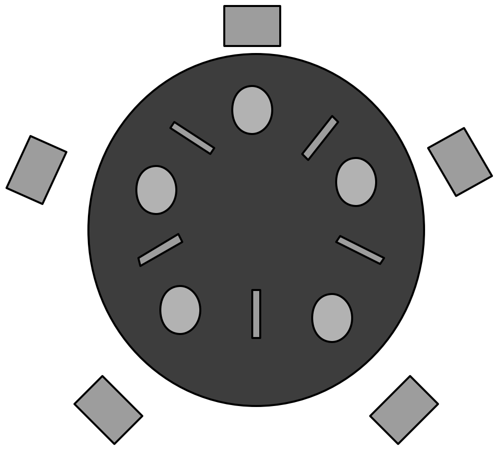
```
semaphore mutex, wrt;
int       readcount;

void      writer()
{
     P(wrt)
     … …
     Do the writing
     … …
     V(wrt)
}
```

The Reader Process

```
void reader(void)
{
     P(mutex)
     readcount++;
     if (readcount == 1) P(wrt);
     V(mutex);
     … …
     Perform the reading
     … …
     P(mutex)
     --readcount;
     if (readcount == 0) V(wrt);
     V(mutex);
}
```

}

Dining Philosopher Problem



A Potential Solution For the Dining Philosophers

```
semaphore chopstick[5];
void philosopher(int i)
{
     do {
          P(chopstick[i]);
          P(chopstick[(i+1}%5];
          eat
          V(chopstick[i]);
          V(chopstick[(i+1}%5];
          think
     } while (TRUE);
}
```

Solutions to the Dining Philosopher's Deadlock

- Allow no more than 4 philosophers at the table.
- Philosophers must pick up both chopsticks at once.
- Odd-numbered philosophers pick up left chopstick first; even-numbered philosophers pick up right chopstick first.

Monitors

- A monitor is a high-level synchronization mechanism proposed by C. A. R. Hoare and P. Brinch Hansen.
- Monitors rely on condition variables and the signal and wait operators.
- Mutual exclusion is automatic; by definition, only one process can be active in a monitor at any time.

Monitor Solution to the Consumer-Producer Problem

```
MONITOR ProducerConsumer;
  TYPE Condition =(NotFull, NotEmpty);
  VAR Count : Integer;
  PROCEDURE Enter;
    BEGIN
      IF Count = N  THEN Wait(NotFull)
      Enter_Item;
      COUNT := Count + 1;
      IF Count = 1 THEN Signal(NotEmpty)
    END;  { Enter }
```

Consuming an Item : The Monitor Solution

```
  PROCEDURE Remove;
    BEGIN
      IF Count = 0  THEN Wait(NotEmpty)
      Remove_Item;
      COUNT := Count - 1;
      IF Count = N-1 THEN Signal(NotFull)
    END;  { Enter }
BEGIN
  Count := 0
END MONITOR;
```

The Producer Process : The Monitor Solution

```
PROCEDURE Producer;
  BEGIN
    WHILE True DO
      BEGIN
        Produce_Item;
        ProducerConsumer.Enter
  END;  { Producer }
```

The Consumer Process : The Monitor Solution

```
PROCEDURE Consumer;
  BEGIN
    WHILE True DO
      BEGIN
        ProducerConsumer.Remove
        Consume_Item;
 END;  { Consumer }
```

Monitor Solution to the Dining Philosopher Problem

```
MONITOR DiningPhilosophers;
  TYPE
  Condition=(Thinking, Hungry, Eating);
  Range = 0..4;
  VAR  State:ARRAY[Range] OF Condition;
       Self:ARRAY[Range] OF Condition;
```

Picking It Up: the Dining Philosophers

```
  PROCEDURE PickUp(i : Range);
    BEGIN
      State[i] := Hungry;
      Test(i);
      IF State[i] <> Eating
        THEN Self[i].Wait
    END;  { PickUp }
```

Putting It Down : the Dining Philosophers

```
PROCEDURE PutDown(i : Range);
  BEGIN
    State[i] := Thinking;
    Test((i+4)MOD 5);
    Test((i+1)MOD 5);
  END;  { PutDown }
```

Testing : the Dining Philosophers

```
PROCEDURE Test(k : Range);
  BEGIN
    IF (State[(k+4) MOD 5) <> Eating)
        AND (State[k]= Hungry) AND
          State[(k+1)MOD 5] <> Eating
      THEN BEGIN
             State[k] := Eating;
             Self[k].Signal;
           END;  { then }
  END;  { Test }
```

The Philosophers Process

```
BEGIN
  FOR i := 0 TO 4
    DO State[i] := Thinking
END;  { DiningPhilosopher }
```