CSE 3221.3
Operating System Fundamentals

No.5

Process Synchronization(1)

Prof. Hui Jiang
Dept of Computer Science and Engineering
York University

Background: cooperating processes with shared memory

- Many processes or threads are cooperating:
  - One way is to use shared memory.
- But concurrent access to shared data may result in data inconsistency.
- To share data among processes (threads), we need some mechanisms to ensure the orderly execution of cooperating processes (threads) to maintain data consistency.

Process Synchronization

- How data inconsistency happens?
  - Example: producer-consumer problem using a bounded-buffer
- Pure software solution:
  - 2-process: Peterson's algorithm
  - N-process: Bakery algorithm
- Synchronization hardware
- Semaphores
- Three classic synchronization problems:
  - The bounded-buffer problem.
  - The reader-writer problem.
  - The dining-philosopher problem.

Producer-Consumer Problem: using shared memory

- Producer-Consumer problem:
  - Two parties: producer & consumer processes
  - A producer process produces information that is consumed by a consumer process.
  - Shared memory:
    - Bounded buffer: a fixed buffer size (producer blocks when the buffer is full)
  - Example:
    - Printer program → printer driver
    - Compiler → assembler

Bounded-Buffer Producer-consumer problem (1)

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer Producer-consumer problem (2)

- Producer process

```
item nextProduced;

while (1) {

     … /* generate a new item in nextProduced */

     while (counter == BUFFER_SIZE) ; /* do nothing */

     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Bounded-Buffer Producer-consumer problem(3)

- Consumer process

```
item nextConsumed;

while (1) {
      while (counter == 0)  ;    /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;

     … /* consume the item in nextConsumed */
}
```

Bounded-Buffer Producer-consumer problem (4)

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
  - which causes unexpected results
- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded-Buffer Producer-consumer problem (5)

- The statement "*counter++*" may be implemented in machine language as:

register1 = counter
register1 = register1 + 1
counter = register1

- The statement "*counter--*" may be implemented as:

register2 = counter
register2 = register2 – 1
counter = register2

Bounded-Buffer Producer-consumer problem(6)

- Assume *counter* is initially 5.
- First step: producer process adds a new item

producer: *register1 = counter* (***register1* = 5**)
producer: *register1 = register1 + 1* (***register1* = 6**)
producer: *counter = register1* (***counter* = 6**)

- Second Step: consumer process takes one item

consumer: *register2 = counter* (***register2* = 6**)
consumer: *register2 = register2 – 1* (***register2* = 5**)
consumer: *counter = register2* (***counter* = 5**)

Bounded-Buffer Producer-consumer problem(7)

- Initially *counter=5.*
- One interleaving of statements is:

producer: *register1 = counter* (***register1* = 5**)
producer: *register1 = register1 + 1* (***register1* = 6**)

Producer process is preempted → context switch → … → to consumer process

consumer: *register2 = counter* (***register2* = 5**)
consumer: *register2 = register2 – 1* (***register2* = 4**)

Consumer process is preempted → context switch → … → to producer process

producer: *counter = register1* (***counter* = 6**)

Producer is pre-empted again

consumer: *counter = register2* (***counter* = 4**)

- The value of *count* ends up with 4 (incorrect!!)

Race Condition

- Race condition: The situation where several processes (or threads) access and manipulate shared data concurrently. The final value of the shared data depends upon the particular order in which the access takes place.
  - General to all shared data in multiprogramming systems.
- Race condition happens in:
  - Multiple processes with shared memory
  - Multi-threaded program
  - Preemptive OS kernel

Race Conditions in OS

- Non-preemptive kernels
  - No race condition occurs in kernel.
- Preemptive kernels
  - Race condition could occur in kernel.
  - Protection techniques are needed for all shared data.
  - Examples:
    - Moving several PCB's from one waiting queue to ready queue; moving several PCB's to the same waiting queue.
    - Kernel counters; kernel flags, …

Bounded-Buffer Producer-consumer problem (8)

- The statements

  counter++; (in producer)

  counter--; (in consumer)

  must be performed ***atomically***.

- Atomic operation means an operation that completes in its entirety without interruption.

Process Synchronization

- Process synchronization: To prevent race conditions, concurrent processes must be synchronized to ensure an orderly execution sequence for all processes.
  - <u>To ensure only one process can manipulate the shared data at a time.</u>

    (the key idea of process synchronization)

The Critical-Section Problem(1)

- *n* processes all competing to use some shared data.
- Each process has code segments, called *critical section*, in which the shared data is accessed.
- Ensure that when one process is executing in its critical section, no other processes are allowed to execute in their critical sections.
  - The execution of critical sections by processes is *mutually exclusive* in time.

The Critical-Section Problem(2)

- General structure of each process P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

Solution to Critical-Section Problem

1. Mutual Exclusion
   - If a process is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress
   - If no process in its critical section and some processes wish to enter their critical sections, then only these processes wishing to enter the critical section can participate in the decision on which will enter the critical section next, and the decision selection of the processes cannot be postponed indefinitely.
3. Bounded Waiting
   - After a process has made a request to enter its critical section, there much be a bound on the number of times that other processes are allowed to enter their critical sections before that request is granted.

Software Solution to the critical section problem

- Assume each process is executing at a non-zero speed. No assumption on their relative speed.
- No assumption on special hardware instructions except each instruction is executed atomically.
- No assumption on the number of CPU's in the system.
- Starting from the case with only two processes:
  - Process P_0 and P_1
  - When presenting P_i, use P_j to indicate another (j=1-i)

Peterson's solution

- Combined shared variables of algorithms 1 and 2.
  boolean flag[2]; **initially** *flag [0] = flag [1] = false.*
  int turn; **initially** *turn = 0* **or** *1.*
- Process P_i

```
do {
    flag [ i ] = true;
    turn = j;
    while (flag [ j ] && turn == j) ;
        critical section
    flag [ i ] = false;
        remainder section
} while (1);
```

Signaling P_i wish to enter the critical section

Asserting that if the other process wish to enter the critical section, it can do

Checking which will enter the critical section

- Meets all three requirements; solves the critical-section problem for two processes perfectly.

Proving Peterson's Algorithm

Process P_0:

```
do {
    flag [ 0 ] = true;
    turn = 1;
    while (flag [ 1 ] && turn == 1) ;
    critical section of P0
    flag [ 0 ] = false;
  remainder section of P0
} while (1);
```

Process P_1:

```
do {
    flag [ 1 ] =  true;
    turn = 0;
    while (flag [ 0 ] && turn == 0) ;
    critical section of P1
    flag [ 1 ] = false;
  remainder section of P1
} while (1);
```

Modification 1

- Shared variables:
  - *int turn;*
    initially *turn = 0*
  - *turn == i* ➔ P_i can enter its critical section.
- Process P_i

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress.

Modification 1 works?

Shared variables:
int turn; **initially** *turn = 0*

Process P_0:

```
do {
    while (turn != 0) ;
    critical section of P0
    turn = 1;
  remainder section of P0
} while (1);
```

Process P_1:

```
do {
    while ( turn != 1) ;
    critical section of P1
    turn = 0;
  remainder section of P1
} while (1);
```

Modification 2

- Shared variables
  - *boolean flag[2];*
    initially *flag [0] = flag [1] = false.*
  - *flag [i] == true* ➔ P_i requests to enter its critical section.
- Process P_i

```
do {
    flag[ i ] = true;
    while ( flag[ j ] ) ;
        critical section
    flag[ i ] = false;
        remainder section
} while (1);
```

Signal that P_i is ready to enter its critical section

- Satisfies mutual exclusion, but not progress requirement.

Modification 2 works?

Shared variables
boolean flag[2];
initially *flag [0] = flag [1] = false.*
flag [i] == true ➔ P_i **requests to enter its critical section.**

```
Process P0:

do {
   flag [ 0 ] = true;
  while (flag [ 1 ]) ;
   critical section of P0
   flag [ 0 ] = false;
  remainder section of P0
} while (1);
```

```
Process P1:

do {
   flag [ 1 ] =  true;
   while (flag [ 0 ]) ;
   critical section of P1
   flag [ 1 ] = false;
  remainder section of P1
} while (1);
```

Bakery Algorithm(1)

- **To solve critical section problem with n processes.**
- **Before entering its critical section, each process receives a number. Holder of the smallest number enters the critical section first.**
- **The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...**
- **If processes P_i and P_j receive the same number, then comparing their process number, if $i < j$, then P_i enters first; else P_j enters first.**

Bakery Algorithm(2)

- **Notation: (ticket #, process id #)**
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0, a_1, \ldots, a_{n-1})$ is a number, k, such that $k >= a_i$ for $i = 0, \ldots, n-1$
- **Shared data**
  - *boolean choosing[n];* (**Initially** *false*)
  - *int number[n];* (**Initially** *0*)

Bakery Algorithm(3)

The Structure of process Pi

```
do {
    choosing[ i ] = true;
    number[ i ] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[ i ] = false;
    for (j = 0; j < n; j++) {
            while (choosing[ j ]) ;
            while ((number[ j ] != 0) && (number[ j ],j) < (number[ i ],i)) ;
    }
      critical section
    number[i] = 0;
      remainder section
} while (1);
```

Bakery Algorithm(3)

The Structure of process Pi

```
do {

    number[ i ] = max(number[0], number[1], …, number [n – 1])+1;

    for (j = 0; j < n; j++) {

            while ((number[ j ] != 0) && (number[ j ],j) < (number[ i ],i)) ;
    }
      critical section
    number[i] = 0;
      remainder section
} while (1);
```

Synchronization Hardware

- **In a uni-processor case, disable interrupt while a shared variable is being modified.**
  - Not feasible in a multiprocessor case
  - Not possible in user-space
- **Many popular CPUs support a limited number of atomic operations:**
  - *Atomic integer instructions*
  - *Atomic bit operation instructions*
  - *Atomic TestAndSet instruction*
  - *Atomic Swap instruciton*

TestAndSet Instruction

- TestAndSet: test and modify the content of a word atomically.
- Atomic even in multi-processor case: if two TestAndSet instructions are executed simultaneously on two different CPUs, they will be executed sequentially in some arbitrary order.

```
boolean TestAndSet (boolean &target) {
        boolean rv = target;
        target = true;

        return rv;
}
```

Mutual Exclusion with TestAndSet (for multiple processes)

- **Shared data:**
  boolean lock = false;
- **Process P_i**

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
}
```

Swap instruction

- Swap the contents of two words atomically as one uninterruptible unit.

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

Mutual Exclusion with Swap (for multiple processes)

- **Shared data:**
  boolean lock; **(initialized to *false*)**
- **Process P_i (with a local variable *key*)**

```
do {
    key = true;
    while (key == true)
        Swap(lock,key);
      critical section
    lock = false;
      remainder section
}
```