CSCI 3753
Operating Systems

Interprocess Synchronization
(Test-and-Set, Semaphores)

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 02/11/13

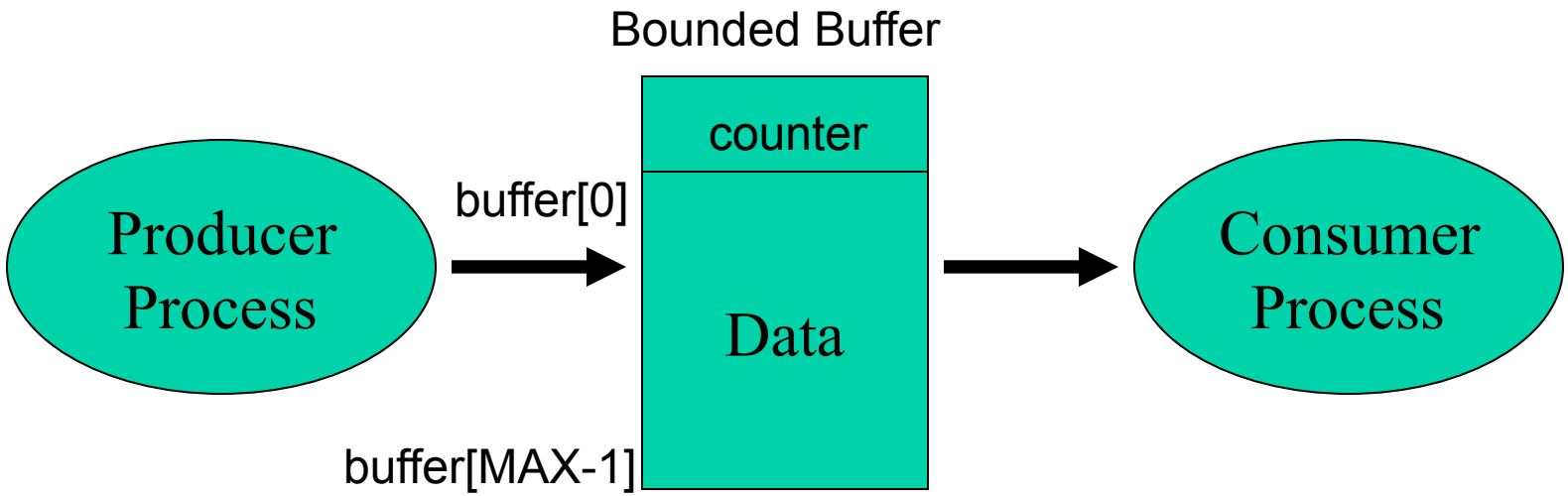
Concurrency

- Multiple processes/threads executing at the same time accessing a shared resource
  - Reading a file
- Value of concurrency – speed & economics
- But few widely-accepted concurrent programming languages (Java is an exception)
- OS tools to support concurrency tend to be "low level"

Producer consumer problem

- Also known as *bounded buffer problem*.
- Two processes (producer and consumer) share a fixed size buffer.
- Producer puts new information in the buffer.
- Consumer takes out information from the buffer.

Synchronization




```
while(1) {
    while(counter==MAX);
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
}
```

```
while(1) {
    while(counter==0);
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
}
```

Producer writes new data into buffer and increments counter

counter updates can conflict!

Consumer reads new data from buffer and decrements counter

Synchronization

- Suppose we have the following sequence of interleaving, where the brackets [value] denote the local value of counter in either the producer or consumer's process. Let counter=5 initially.

// counter++		// counter--;	
(1) reg1 = counter;	[5]	(2) reg2 = counter;	[5]
(3) reg1 = reg1 + 1;	[6]	(4) reg2 = reg2 - 1;	[4]
(5) counter = reg1;	[6]	(6) counter = reg2;	[4]

- At the end, desired value of counter should be 5 with one producer and one consumer, but counter = 4! Plus if steps (5) and (6) were reversed, then counter=6 !!! - undesirable and unpredictable *race condition*
- Basic Problem: unprotected access to a shared variable (counter)

Synchronization

counter++; can compile into several machine language instructions, e.g.

```
reg1 = counter;
reg1 = reg1 + 1;
counter = reg1;
```

counter--; can compile into several machine language instructions, e.g.

```
reg2 = counter;
reg2 = reg2 - 1;
counter = reg2;
```

If these low-level instructions are *interleaved*, e.g. the Producer process is context-switched out, and the Consumer process is context-switched in, and vice versa, then the results of counter's value can be unpredictable

Race Condition

- Situations when two are more processes (or threads) are accessing a shared resource, and the final result depends on which process runs precisely when are called race conditions.
- Race conditions can occur if two are more processes are accessing a shared resource.
- The part of the program where a shared resource is accessed is called *critical section*.
- We need a mechanism to prohibit multiple processes from accessing a shared resource at the same time.

Mutual Exclusion

- No more than one process can execute in a critical section at any time
  - Two or more processes may not execute in a critical section (access to the same shared resource) at the same time.
- How can we implement mutual exclusion?

```
entry section

    critical section (manipulate common var's)

exit section
    remainder section code
```

```
//Producer
    entry section

        counter++;

    exit section

        remainder section code
```

```
//Consumer
    entry section

        counter--;

    exit section

        remainder section code
```

Critical Section

- Critical section access should satisfy multiple properties
  - **mutual exclusion**
    - if process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  - **progress**
    - if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next
    - this selection cannot be postponed indefinitely (OS must make a decision eventually, hence "progress")
  - **bounded waiting**
    - there exists a bound, or limit, on the number of times other processes can enter their critical sections after a process X has made a request to enter its critical section and before that request is granted (no starvation)
- For most of the following slides, we will primarily be concerned with how to achieve mutual exclusion

Disabling interrupts

- Ensure that when a process is executing in its critical section, it cannot be preempted.
- Disable all interrupts before entering a CS.
- Enable all interrupts upon exiting the CS.

```
shared int counter;
```

Code for p_1

```
disableInterrupts();
counter++;
enableInterrupts();
```

Code for p_2

```
disableInterrupts();
counter--;
enableInterrupts();
```

- Problems:
  1. If a user forgets to enable interrupts???
  2. Two or more CPUs???

- Interrupts could be disabled arbitrarily long
- Really only want to prevent p_1 and p_2 from interfering with one another; this blocks all processes
- Blocks overlapping I/O

A Flawed Lock Implementation

```
shared boolean lock = FALSE;
shared int counter;
```

Code for p_1

```
/* Acquire the lock */
  while(lock){ no_op;}
  lock = TRUE;
/* Execute critical
       section */
  counter++;
/* Release lock */
  lock = FALSE;
```

Acquire(lock)

Code for p_2

```
/* Acquire the lock */
while(lock){ no_op;}
lock = TRUE;
/* Execute critical
       section */
counter--;
/* Release lock */
lock = FALSE;
```

Both processes may enter their critical section if there is a context switch just before the <lock = TRUE> statement

Need a way to test and set the value of a variable atomically

Atomic Test-and-Set

- Need to be able to look at a variable and set it up to some value without being interrupted

  y = read (x); x = value;

- Modern computing systems provide such an instruction called *test-and-set (TS);*

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

- The entire instruction (sequence) is atomic, enforced by hardware

Mutual exclusion using TS

```
shared boolean lock = FALSE;
shared int counter;
```

Code for p_1

```
/* Acquire the lock */
  while(TS(&lock)) ;

/* Execute critical section */
  counter++;
/* Release lock */
  lock = FALSE;
```

Code for p_2

```
/* Acquire the lock */
   while(TS(&lock)) ;

/* Execute critical section */
        counter--;
/* Release lock */
   lock = FALSE;
```

- The boolean TestandSet() instruction is essentially a swap of values
  - The x86 CPU instruction set contains atomic instructions such as XCHG that are essentially swap statements
  - Can use atomic XCHG to implement spinlocks
- Mutual exclusion is achieved - no race conditions
  - If one process X tries to obtain the lock while another process Y already has it, X will wait in the loop
  - If a process is testing and/or setting the lock, no other process can interrupt it
- The system is exclusively occupied for only a short time - the time to test and set the lock, and not for entire critical section
  - typically only about 10 instructions
- Don't have to disable and reenable interrupts - time-consuming
- Do you see any problems? → busy waiting

wait( ) and **signal( )** primitives

- Also called *sleep()* and *wakeupl()* primitives
- *wait()*: causes a process to block.
- *wakeup(pid)*: causes the process whose id is *pid* to move to ready state.
  - No effect if process *pid* is not blocked.

```
while(1) {
    if (counter==MAX) wait();
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
    if (counter == 1) signal (p2);
}
```

```
while(1) {
    if (counter==0) wait();
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
    if (counter == MAX – 1) signal (p1);
}
```

- Consumer reads counter and counter = 0.
- Scheduler schedules the producer.
- Producer puts an item in the buffer and signals the consumer
  - Since consumer has not yet invoked wait(), the signal() invocation by the producer has no effect.
- Consumer is scheduled, and it blocks.
- Eventually, producer fills up the buffer and blocks.
- How can we solve this problem?
  - Need a mechanism to count the number of wait() and signal() invocations.

Semaphores

- More general solution to mutual exclusion proposed by Dijkstra
- Semaphore S is an abstract data type that, apart from initialization, is accessed only through 2 standard atomic operations
  - P(), also called wait(), short for Dutch word *proberen* "to test"
    - somewhat equivalent to a test-and-set, but also involves *decrementing* the value of S
  - V(), also called signal(), short for Dutch word *verhogen* "to increment"
    - *increments* the value of S
  - OS provides ways to create and manipulate semaphores atomically

Semaphores

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

```
P(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

```
V(semaphore *s) {
    s→value++;
    if (s→value <= 0) {
        remove a process P from s→list;
        wakeup (P);
    }
}
```

Both P and V operations are atomic

Mutual Exclusion with Semaphores

```
semaphore S = 1;  // initial value of semaphore is 1
int counter;      // assume counter is set correctly somewhere in
                     code
```

Process P1:

```
P(S);
   // execute critical section
   counter++;
V(S);
```

Process P2:

```
P(S);
   // execute critical section
   counter--;
V(S);
```

- Both processes atomically P() and V() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable counter

Problems with semaphores

- Potential for deadlock

```
Semaphore Q= 1;    // binary semaphore as a mutex lock
Semaphore S = 1;   // binary semaphore as a mutex lock
variable R1, R2;
```

Process P1:

```
P(S);          (1)
P(Q);          (3)

modify R1 and R2;

V(S);
V(Q);
```

Process P2:

```
P(Q);          (2)
P(S);          (4)

modify R1 and R2;

V(Q);
V(S);
```

Deadlock

- In the previous example,
  - Each process will block on a semaphore
  - The V() statements will never get executed, so there is no way to wake up the two processes
  - There is no rule wrt the order in which P( ) and V( ) operations may be invoked
  - In general, with N processes sharing N semaphores, the potential for deadlock grows

Other problematic scenarios

- A programmer mistakenly follows a P() with a second P() instead of a V()
- A programmer forgets and omits the P(mutex) or V(mutex)
- A programmer reverses the order of P() and V()