CSCI 3753
Operating Systems

Interprocess Synchronization
(Test-and-Set, Semaphores)

Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 09/26/17

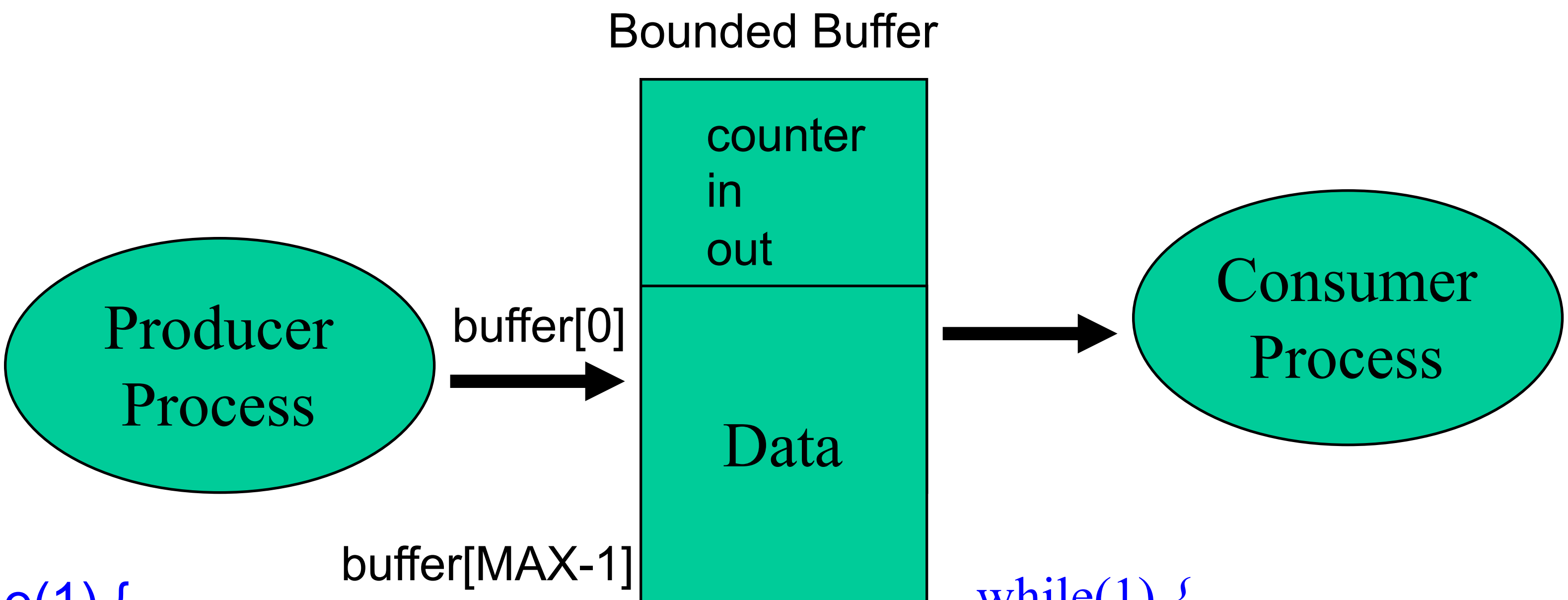
Concurrency

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

Producer writes new data into buffer and increments counter

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

Consumer reads new data from buffer and decrements 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 preempted, and the consumer process is scheduled to run, or vice versa, then the results of counter's value can be unpredictable

Synchronization

- Suppose we have the following sequence of interleaving. Let counter = 5 initially.

```
// counter++
(1) reg1 = counter;
(3) reg1 = reg1 + 1;
(5) counter = reg1;
```

```
// counter--;
(2) reg2 = counter;
(4) reg2 = reg2 - 1;
(6) counter = reg2;
```

		1	2	3	4	5	6
counter	5	5	5	5	5	6	4
reg1	?	5	5	6	6	6	6
reg2	?	?	5	5	4	4	4

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
- How can we implement mutual exclusion?

Regular code

Entry section

Critical section

Access shared resource

Exit section

Regular code

```
// Producer
while(1) {
    produce (nextdata);
    while (counter==MAX);
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    Entry section
    counter++;
    Exit section
}
```

```
// Consumer
while(1) {
    while (counter==0);
    getdata = buffer[out];
    out = (out+1) % MAX;
    Entry section
    counter--;
    Exit section
    consume (getdata)
}
```

Critical Section

Race Condition: Solution

- Solution must satisfy the following conditions:
  - **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 wish to enter their critical 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

Solution 1

- 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;
```

producer code

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

remaining producer code

consumer code

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

remaining consumer code

Solution1: Disabling interrupts

- 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

Software Only Solution

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

<u>Code for producer</u>

```
/* Acquire the lock */
  while(lock){ no_op;} (1)
  lock = TRUE; (3)
/* Execute critical
      section */
  counter++;
/* Release lock */
  lock = FALSE;
```

<u>Code for consumer</u>

```
/* Acquire the lock */
while(lock){ no_op;}(2)
lock = TRUE; (4)
/* Execute critical
      section */
counter--;
/* Release lock */
lock = FALSE;
```

<u>A flawed lock implementation:</u>

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

Software Only Solution

- Implementing mutual exclusion in software is extremely difficult
- *Read Section 5.3 for a software only solution*

- Need help from hardware
- Modern processors provide such support
    - Atomic test and set instruction
    - Atomic compare and swap instruction

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 sequence is a single instruction (atomic), implemented in 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 TS () 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
- Don’t have to disable and reenable interrupts - time-consuming
- Do you see any problems?
  - busy waiting

sleep( ) and **wakeup( )** primitives

- *sleep()*: causes a running 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

```
// producer
while(1) {
    if (counter==MAX) sleep();
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
    if (counter == 1) wakeup (p2);
}

// consumer
while(1) {
    if (counter==0) sleep();
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
    if (counter == MAX – 1) wakeup (p1);
}
```

- Problem with counter++ and counter-- still exist
  - Can be solved using TS (exercise?)
- Consumer reads counter and counter = 0
- Scheduler schedules the producer
- Producer puts an item in the buffer and signals the consumer to wake up
  - Since consumer has not yet invoked sleep(), the wakeup() 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 sleep() and wakeup() 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 two standard atomic operations
  - wait( ) (also called P(), short for Dutch word *proberen* "to test")
    - somewhat equivalent to a test-and-set, but also involves *decrementing* the value of S
  - signal( ) (also called V(), 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;
    PID *list[ ];
} semaphore;
```

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

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

Both wait() and signal() 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:

```
wait(S);
   // execute critical section
   counter++;
signal(S);
```

Process P2:

```
wait(S);
   // execute critical section
   counter--;
signal(S);
```

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

Problems with semaphores

shared R1, R2;

semaphore Q = 1;   // binary semaphore as a mutex lock for R1

semaphore S = 1;   // binary semaphore as a mutex lock for R2

Process P1:

wait(S); (1)

wait(Q); (3)

modify R1 and R2;

signal(S);

signal(Q);

Process P2:

wait(Q); (2)

wait(S); (4)

modify R1 and R2;

signal(Q);

signal(S);

- Potential for deadlock

Deadlock

- In the previous example,
  - Each process will block on a semaphore
  - The signal() 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 wait( ) and signal( ) 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 wait() with a second wait() instead of a signal()
- A programmer forgets and omits the wait(mutex) or signal(mutex)
- A programmer reverses the order of wait() and signal()

Producer consumer problem

- We have already seen this problem with one producer and one consumer
- General problem: multiple producers and multiple consumers
- Producers puts new information in the buffer
- Consumers takes out information from the buffer

Semaphore empty = 0, full = MAX, m = 1;

```
producer()                          consumer ()
    produce_info(item);                 wait(empty);
    wait(full);                         wait(m);
    wait(m);                                remove_info(item);
        enter_info(item);               signal(m);
    signal(m);                          signal(full);
    signal(empty);                      consume_info(item);
```

Semaphores empty and full are used for maintaining counter values and signaling between producer and consumer processes.

Semaphore m is used for mutual exclusion among producer processes and among consumer processes.

Pthread Synchronization

- Mutex locks
  - Used to protect critical sections
- Some implementations provide semaphores through POSIX SEM extension
  - Not part of Pthread standard

```
#include <pthread.h>
pthread_mutex_t m; //declare a mutex object
Pthread_mutex_init (&m, NULL); // initialize mutex object
```

```
//thread 1
pthread_mutex_lock (&m);
   //critical section code for th1
pthread_mutex_unlock (&m);
```

```
//thread 2
pthread_mutex_lock (&m);
   //critical section code for th2
pthread_mutex_unlock (&m);
```

Pthread mutex

- pthread mutexes can have only one of two states: lock or unlock
- Important restriction
  - Mutex ownership: Only the thread that locks a mutex can unlock that mutex
  - So, mutexes are strictly used for mutual exclusion while binary semaphores can also be used for synchronization between two threads

POSIX semaphores

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
//pshared: 0 (among threads); 1 (among processes)

int sem_wait(sem_t *sem); //same as wait( )

int sem_post(sem_t *sem); //same as signal( )

sem_getvalue( ), sem_close( )
```

Kernel Synchronization

- At any time, many kernel mode processes may be active
  - Share kernel data structures
  - Notice that even though user processes have their own address spaces, race conditions can still arise when they execute in kernel mode, e.g. executing a system call
- Preemptive and non-preemptive kernels
  - Preemptive kernel: allows a process to be preempted while running in kernel mode
    - Race conditions can occur
  - Non-preemptive kernel: does not allow a process to be preempted while running in kernel mode
    - Race conditions cannot occur

Windows Synchronization

- Kernel level
  - Single processor system: temporarily mask interrupts for all interrupt handlers that may also access a shared resource
  - Multiprocessor system: use spin lock (busy waiting)
- User level
  - Dispatcher objects: mutex locks, semaphores, …

Linux Synchronization

- Kernel level
  - Prior to version 2.6, non-preemptive kernel, but later versions are fully preemptive
  - Atomic integers: all math operations on atomic integers are performed without interruptions

```
atomic_t counter;
atomic_set(&counter, 5);
atomic_add(10, &counter);
…
```

  - Mutex locks, spin locks and semaphores, enabling/disabling interrupts on single processor systems
- User level
  - Futex, semop( ): system call