Operating Systems

Concurrency: Mutual Exclusion and Synchronization

Chapter 5

Outline

- Principles of Concurrency
- Mutual Exclusion Software Approaches
- Mutual Exclusion: Hardware Support
- Mutual Exclusion: OS and programming language support
  - Semaphores

Concurrency

Concurrency affects a number of design issues

- Communication among processes
- Sharing resources
- Synchronization of multiple processes
- Allocation of processor time

Concurrency

Concurrency contexts

- Multiple applications
  - Multiprogramming
- Structured application
  - Application can be a set of concurrent processes
- Operating-system structure
  - Operating system is a set of processes or threads

Difficulties with Concurrency

The relative speed of execution of processes cannot be predicted

- Sharing global resources
  - Two processes access the same global variable?
- Management of allocation of resources
  - A process is allocated an I/O channel and then is suspended before using that channel?
- Programming errors difficult to locate

A Simple Example

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

chin and chout are shared global variables

A Simple Example

Process P1 **Process P2**

```
                    .
chin = getchar();
//Process P1 is     Run to completion
  interrupted


chout = chin;                    What will happen?
putchar(chout);
```

Single-processor multiprogramming system

Shared procedure

A Simple Example

Process P1	Process P2
`chin = getchar();`	.
`//Process P1 is interrupted`	`//P2 blocked from entering the procedure`
`//Process P1 resumed`	
`chout = chin;`	
`putchar(chout);`	

What will happen?

Single-processor multiprogramming system

Only one process at a time may be in that procedure

A Simple Example

```
  Process P1          Process P2

.                       .
chin = getchar();       .
.                       chin = getchar();
chout = chin;           chout = chin;
putchar(chout);         .
.                       putchar(chout);
.                       .
```

Multiprocessor system, P1 and P2 each on a separate processor invoking the echo proc.

Operating System Concerns

- Keep track of active processes
- Allocate and deallocate resources
  - Processor time
  - Memory
  - Files
  - I/O devices
- Protect data and resources
- Result of process must be independent of the speed of execution of other concurrent processes

Process Interaction

- Processes unaware of each other
  - Competition among processes for resources
- Processes indirectly aware of each other
  - Cooperation among processes by sharing (share access to the same object)
- Process directly aware of each other
  - Cooperation among processes by communication (communicate by process IDs)

Competition Among Processes for Resources 1/2

3 control problems

- The need for Mutual Exclusion
  - Critical sections
    - The portion of the program that uses a critical resource (a non-sharable resource)
    - Only one program at a time is allowed in its critical section
    - Example only one process at a time is allowed to send command to the printer
- Deadlock
- Starvation

Competition Among Processes for Resources 2/2

Mutual Exclusion mechanism in abstract terms

```
 /* program mutualexclusion */
const int n = /*number of processes*/
void P(int i)
{
        while (true)
        {
                entercritical (i);
                /*critical section */
                exitcritical(i);
                /* remainder */
        }
}
void main ()
{
        parbegin(P(R1), P(R2), …., P(Rn));
}
```

Cooperation Among Processes by Sharing

- Access to shared data
- Data items accessed in reading and writing modes
- Writing must be mutually exclusive
- Critical sections are used to provide data integrity → data coherence

P1:	P2:
a = a+1;	b = 2 * b;
b = b+1;	a = 2 * a;

Maintain a=b

Cooperation Among Processes by Communication

- Messages are passed
  - Mutual exclusion is not a control requirement
- Possible to have deadlock
  - Each process waiting for a message from the other process
- Possible to have starvation
  - Two processes sending message to each other while another process waits for a message

Requirements for Mutual Exclusion 1/3

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its non-critical section must do so without interfering with other processes
- No deadlock or starvation

Requirements for Mutual Exclusion 2/3

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Requirements for Mutual Exclusion 3/3

How can the requirements be satisfied?

- Processes responsibility, no support from OS or programming language
  - Software approaches
- Special-purpose machine instructions
- Some level of support within the OS or a programming language

Mutual Exclusion: Software Approaches

- Assume elementary mutual exclusion at memory access level
  - Simultaneous access to same location in memory are serialized by some sort of memory arbiter
- Dekker's Algorithm (reported by Dijkstra)
  - 4 attempts to reach the correct solution!
- Peterson's Algorithm

First Attempt 1/2

- Busy Waiting
  - Process is always checking to see if it can enter the critical section
  - Process can do nothing productive until it gets permission to enter its critical section

First Attempt 2/2

- Turn variable (figure 5.2a)

P0	P1
while (turn != 0) /**/;	while (turn != 1) /**/;
*/*critical section*/*	*/*critical section*/*
turn = 1;	turn = 0;

  - Shared global variable *turn* indicates who is allowed to enter next, can enter if *turn* = *me*
  - On exit, point variable to other process
  - Processes must strictly alternate → pace of execution dictated by slower process
  - If one process fails, other process is permanently blocked

Coroutine

- Designed to be able to pass execution control back and forth between themselves
- Inadequate to support concurrent processing

Second Attempt 1/2

- Each process can examine the other's status but cannot alter it
- When a process wants to enter the critical section is checks the other processes first
- If no other process is in the critical section, it sets its status for the critical section
- *This method does not guarantee mutual exclusion*
  - Each process can check the flags and then proceed to enter the critical section at the same time

Second Attempt 2/2

- "Busy" Flag → state information (figure 5.2b)

P0	P1
while (flag[1]) /**/;	while (flag[0]) /**/;
flag[0] = true;	flag[1] = true;
*/*critical section*/*	*/*critical section*/*
flag[0] = false;	flag[1] = false;

  - Each process has a *flag* to indicate it is in the critical section
  - Fails mutual exclusion if processes are in *lockstep*
  - What happens if a process fails inside its critical section?

Third Attempt 1/2

- Set flag to enter critical section before check other processes
- If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section
- Deadlock is possible when two process set their flags to enter the critical section. Now each process must wait for the other process to release the critical section

Third Attempt 2/2

•Busy Flag Modified (figure 5.2c)

P0	P1
flag[0] = true;	flag[1] = true;
while (flag[1]) /**/;	while (flag[0])/**/;
*/*critical section*/*	*/*critical section*/*
flag[0] = false;	flag[1] = false;

•Guarantees mutual exclusion

•Deadlock if processes are in lockstep (both processes set their flags to true before either has executed the while statement)

Fourth Attempt 1/3

- A process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag
- Other processes are checked. If they are in the critical region, the flag is reset and later set to indicate desire to enter the critical region. This is repeated until the process can enter the critical region.

Fourth Attempt 2/3

- It is possible for each process to set their flag, check other processes, and reset their flags. This scenario will not last very long so it is not deadlock. It is undesirable

Fourth Attempt 3/3

•Busy Flag Again (figure 5.2d)

P0	P1
flag[0] = true;	flag[1] = true;
while (flag[1])	while (flag[0])
{	{
flag[0] = false;	flag[1] = false;
/*delay*/	/*delay*/
flag[0] = true;	flag[1] = true;
}	}
/*critical section*/	/*critical section*/
flag[0] = false;	flag[1] = false;

•*Livelock* if processes are in lockstep

Correct Solution 1/2

- Each process gets a turn at the critical section
- If a process wants the critical section, it sets its flag and may have to wait for its turn

Correct Solution 2/2

```
//see Fig. 5.3
boolean flag [2];
int turn;

void main ()
{
  flag [0] = false;
  flag [1] = false;
  turn = 1;
  parbegin (P0, P1);
}
```

```
void P0()
{
        while (true)
        {
                flag [0] = true;
                while (flag [1])
                        if (turn == 1)
                        {
                          flag [0] = false;
                          while (turn == 1)
                          /* do nothing */;
                          flag [0] = true;
                        }
                /* critical section */;
                turn = 1;
                flag [0] = false;
                /* remainder */;

        }
}
```

Peterson's Algorithm

P0

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

P1

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

Can you show that mutual exclusion is preserved?

See page 212

Mutual Exclusion: Hardware Support 1/5

- Interrupt Disabling
  - A process runs until it invokes an operating-system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Processor is limited in its ability to interleave programs
  - Multiprocessing
    - disabling interrupts on one processor will not guarantee mutual exclusion

Mutual Exclusion: Hardware Support 2/5

- Special Machine Instructions
  - At a hardware level, access to a memory location excludes any other access to the same location
  - Performed in a single instruction cycle (Not subject to interference from other instructions)
    - Reading and writing
    - Reading and testing

Mutual Exclusion: Hardware Support 3/5

- Test and Set Instruction

```
boolean testset (int i) {
        if (i == 0) {
                i = 1;
                return true;
        }
        else {
                return false;
        }
}
```

Carried out atomically, not subject to interruption

Mutual Exclusion: Hardware Support 4/5

A process that finds *bolt* equal to zero can enter its critical section

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
        while (true)
        {
                while (!testset (bolt)) /* do nothing */;
                /* critical section */;
                bolt = 0;
                /* remainder */
        }
}
void main()
{
        bolt = 0;
        parbegin (P(1), P(2), . . . ,P(n));
}
```

Mutual Exclusion: Hardware Support 5/5

- Exchange Instruction

```
void exchange(int register,
                    int memory) {
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

See Fig. 5.5b for a mutual exclusion protocol based on this instruction

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Mutual Exclusion: OS and programming support

- Semaphores
- Monitors
- Message Passing

Semaphores 1/5

- Two or more processes can cooperate by means of simple signals
- A process is forced to stop at a specified place until it has received a specific signal
- Special variable called a semaphore is used for signaling
  - To transmit a signal via a semaphore *s*, a process executes primitive *signal(s) (V → increment in dutch)*
  - To receive a signal via semaphore *s*, a process executes primitive *wait(s) (P → test in dutch)*

Semaphores 2/5

- *If a process is waiting for a signal, it is suspended until that signal is sent*
- Wait and signal operations cannot be interrupted
- Queue is used to hold processes waiting on the semaphore

Semaphores 3/5

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - *wait* operation decrements the semaphore value
    - If value becomes negative, process executing the *wait* is blocked
  - *signal* operation increments semaphore value
    - If value is not positive, a process blocked by a *wait* operation is unblocked

Semaphores 4/5

```
struct semaphore {
        int count;
        queueType queue;
}

void wait(semaphore s)
{
        s.count--;
        if (s.count < 0)
        {
            place this process in s.queue;
            block this process
        }
}
void signal(semaphore s)
{
        s.count++;
        if (s.count <= 0)
        {
            remove a process P from s.queue;
            place process P on ready list;
        }
}
```

Definition of semaphore primitives

See Fig. 5.6

Strong semaphore versus weak semaphore (specification of order of processes to be removed from queue)

Semaphores 5/5

```
struct binary_semaphore {
        enum (zero, one) value;
        queueType queue;
};

void waitB(binary_semaphore s)
{
     if (s.value == 1)
          s.value = 0;
     else
        {
          place this process in s.queue;
          block this process;
        }
}
void signalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Definition of binary semaphore primitives

See Fig. 5.7

Mutual Exclusion using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true)
    {
        wait(s);
        /* critical section */;
        signal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . ., P(n));
}
```

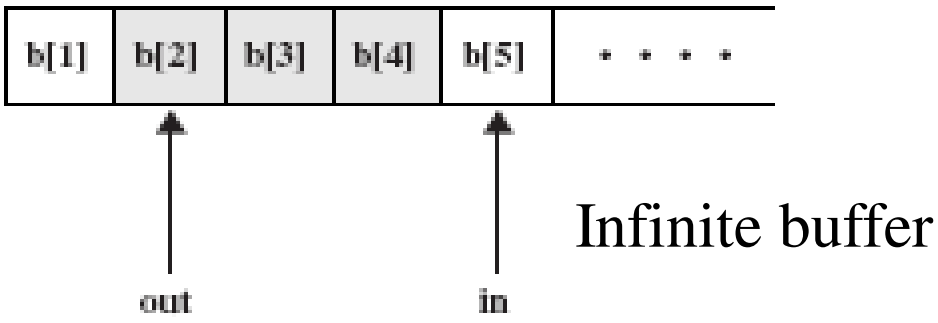
One process is allowed in its critical section at a time

Producer/Consumer Problem

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time

Producer

```
producer:
while (true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```



Note: shaded area indicates portion of buffer that is occupied

Consumer

```
consumer:
while (true) {
  while (in <= out)
     /*do  nothing */;
  w = b[out];
  out++;
  /* consume item w */
}
```

Must not attempt to read from an empty buffer (in > out)

Producer/Consumer using Binary Semaphores

Producer

```
do forever
        produce item
        waitB(s)
        append
        n++
        if (n == 1)
            signalB(delay)
        signalB(s)
```

Consumer

```
    wait(delay)
    do forever
        waitB(s)
        remove from queue
        n--
        m = n
        signalB(s)
        if (m = 0)  waitB(delay)
```

Fig. 5.13

n: number of items in the buffer, init to 0

s: semaphore used to enforce mutual exclusion, init to 1

delay: semaphore used to force consumer to wait if buffer is empty, init to 0

Producer/Consumer using Counting Semaphores

Producer
do forever

produce()
wait(s)
append()
signal(s)
signal(n)

Consumer
do forever

wait(n)
wait(s)
take()
signal(s)
consume()

Fig. 5.14

What happens if these 2 statements are interchanged?

n: semaphore, number of items in the buffer, init to 0

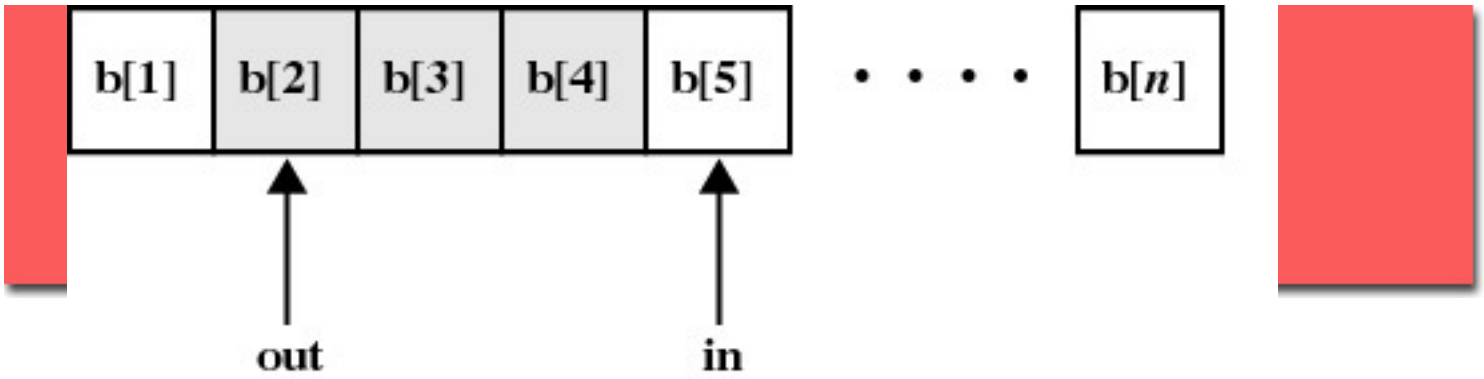
s: semaphore used to enforce mutual exclusion, init to 1

Producer with Circular Buffer

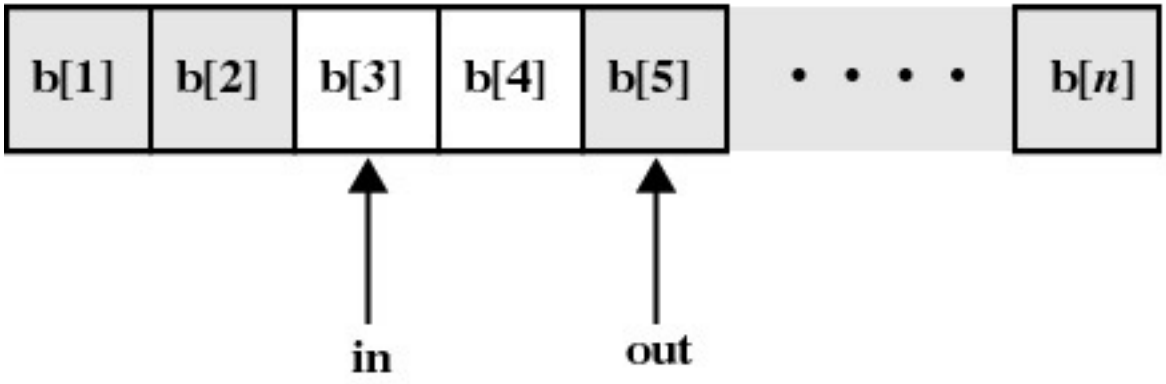
```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)
     /* do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

Consumer with Circular Buffer

```
consumer:
while (true) {
  while (in == out)
     /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```



(a)



(b)

Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

Producer/Consumer using circular buffer

Producer

```
do forever
        produce()
        wait(e)
        wait(s)
        append()
        signal(s)
        signal(n)
```

Consumer

```
do forever
        wait(n)
        wait(s)
        take()
        signal(s)
        signal(e)
        consume()
```

Fig. 5.16

n: semaphore, number of items in the buffer, init to 0

s: semaphore used to enforce mutual exclusion, init to 1

e: semaphore, number of empty spaces in buffer, init to size of buffer