Concurrency

critical section a section of code within a process that requires access to shared resources, and which may not be executed while another process is in a corresponding section of code

deadlock a situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something

livelock a situation in which two or more processes continuously change their state in response to changes in the other processes without doing any useful work

mutual exclusion the requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources

race condition a situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution

starvation a situation in which a runnable process is overlooked indefinitely by the scheduler; although it's able to proceed, it is never chosen

A Simple Example

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

```
        //Process P1              //Process P2
 t  |   .                         .
 i  |   chin = getchar();         .
 m  |   .                         chin = getchar();
 e  |   chout = chin;             chout = chin;
    |   putchar(chout);           .
    |   .                         putchar(chout);
    v   .                         .
```

Operating System Concerns

Keep track of various processes

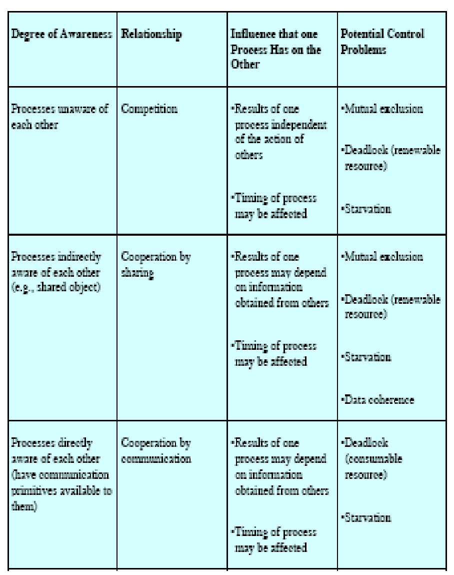
Allocate and deallocate resources

- processor time
- memory
- files
- I/O devices

Protect data and resources

Output of process must be independent of the speed of execution of other concurrent processes

Process Interaction



Degree of Awareness	Relationship	Influence that one Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Competition Among Processes for Resources

Mutual Exclusion

- critical sections
- 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

Requirements for Mutual Exclusion

Only one process at a time is allowed in the critical section for a resource

A process that halts in its noncritical section must do so without interfering with other processes

No deadlock or starvation

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

Mutual Exclusion: Hardware Support

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

Special Machine Instructions

- performed in a single instruction cycle
- access to the memory location is blocked for any other instructions

Mutual Exclusion: Hardware Support

Test and Set Instruction

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

Executed atomically, effectively as a single machine instruction.

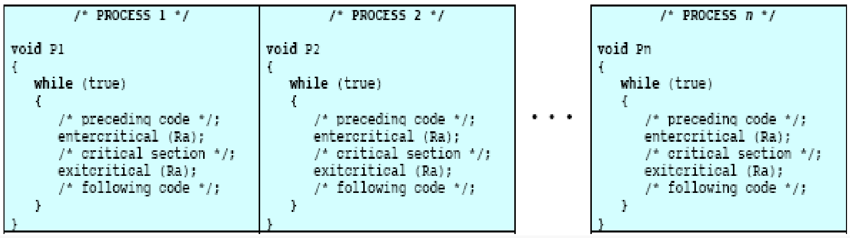
Mutual Exclusion: Hardware Support

Exchange Instruction

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

Executed atomically, effectively as a single machine instruction.

Mutual Exclusion



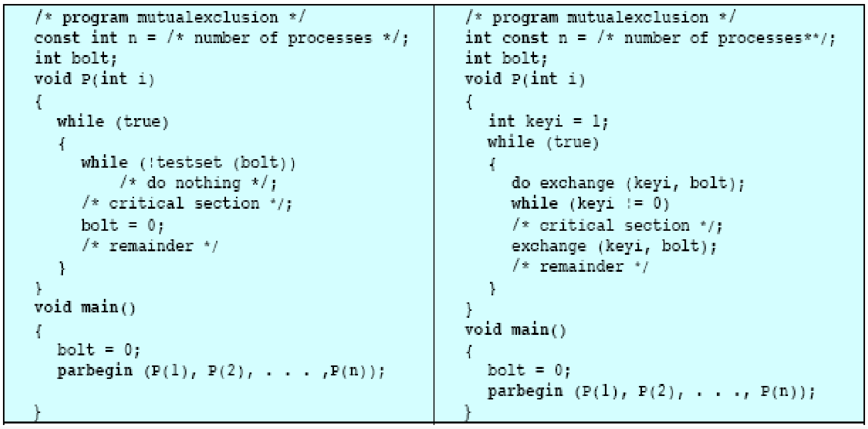
```
/* PROCESS 1 */

void P1
{
   while (true)
   {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
/* PROCESS 2 */

void P2
{
   while (true)
   {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

• • •

```
/* PROCESS n */

void Pn
{
   while (true)
   {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

Mutual Exclusion



```
/* 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));

}
```

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true)
   {
      do exchange (keyi, bolt);
      while (keyi != 0)
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

Using test-and-set

Using exchange

Mutual Exclusion via 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

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

Semaphores

Dijkstra

Special variable called a *semaphore* is used for signaling

If a process is waiting for a signal, it is suspended until that signal is sent

Semaphore is a variable that has an integer value

- may be initialized to a nonnegative number
- *wait* operation decrements the semaphore value
- *signal* operation increments semaphore value
- also has a process queue and a flag

Semaphore can be implemented using machine instruction test-and-set, or using interrupts.

Semaphore Primitives

```
struct semaphore {
   int Count;
   queue Q;
}

void semWait(semaphore s) {
   s.Count--;
   if ( s.Count < 0 ) {
      // place caller in Q
      // block caller
   }
}

void semSignal(semaphore s) {
   s.Count++;
   if ( s.Count <= 0 ) {
      // dequeue proc P from Q
      // place P on ready list
   }
}
```

Semaphore variable contains a counter and a process queue.

When a process calls `semWait()`, the semaphore's counter is decremented.

If the counter is now negative, the caller is blocked.

The initial value given to the counter controls how many processes are allowed "past" the semaphore at once.

When a process calls `semSignal()`, the semaphore's counter is incremented.

If the counter is not positive, there must be at least one process blocked on the semaphore.

So, a blocked process is dequeued and allowed to "pass" the semaphore.

Binary Semaphore Primitives

```
struct binary_semaphore {
   enum {ZERO, ONE} Value;
   queue Q;
}

void semWaitB(binary_semaphore s) {
   if ( s.Value == ONE ) {
      s.Value = ZERO;
   else {
      // place caller in Q
      // block caller
   }
}
```

Binary semaphores can only be 0 or 1.

That means they would achieve one-at-a-time mutual exclusion.

```
void semSignalB(binary_semaphore s) {
   if ( s.Q.isempty() ) {
      s.Value = ONE;
   else {
      // dequeue proc P from Q
      // place P on ready list
   }
}
```

Implementing a Semaphore

The basic problem with implementing a semaphore is that only one process can be allowed to execute the body of `semWait()` or `semSignal()` at a time.

One way to do this is to use the hardware-level test-and-set instruction described earlier.

```
struct semaphore {
   int Count, Flag;
   queue Q;
}
```

```
void semWait(semaphore s) {
   while ( !testset(s.flag) );
   s.Count--;
   if ( s.Count < 0 ) {
      // place caller in Q
      // block caller
   }
   s.Flag = 0;
}
```

```
void semSignal(semaphore s) {
   while ( !testset(s.flag) );
   s.Count++;
   if ( s.Count <= 0 ) {
      // dequeue proc P from Q
      // place P on ready list
   }
   s.Flag = 0;
}
```

Mutual Exclusion Using Semaphores

```
const int n = ...; // set # of processes
semaphore s = 1;

void P( int i ) {
   while ( true ) {
      semWait(s);
      // critical section
      semSignal(s);
      // non-critical section
   }
}

int main() {
   parbegin(P(1), P(2), ..., P(n)); // spawn threads running P()
}
```

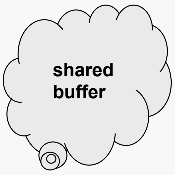
Producer/Consumer Problem

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



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

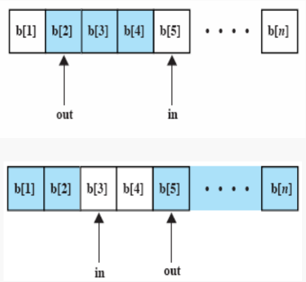
The buffer may be finite (bounded) or infinite.

Usually modeled as array-like; circular if finite.

PC Problem with Circular Buffer

```
producer:
while (true) {
   /* produce item v */
   while ((in + 1) % n == out);
   b[in] = v;
   in = (in + 1) % n
}
```

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



Infinite-buffer PC with Semaphores

```
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
```

```
void producer() {
   while (true) {
      produce();
      semWaitB(s);
      append();
      n++;
      if ( n == 1 )
         semSignalB(delay);
      semSignalB(s);
   }
}
```

```
void consumer() {
   semWaitB(delay);
   while (true) {
      semWaitB(s);
      take();
      n--;
      semSignalB(s);
      consume();
      if ( n == 0 )
         semWaitB(delay);
   }
}
```

```
int main() {
   n = 0;
   parbegin(producer, consumer);
}
```

Hypothetical Trace of Execution

	Producer	Consumer	s	n	delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0

4	if (n == 1)
	semSignalB(delay)
			0	1	1

5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0

12	if (n == 1)
	semSignalB(delay)
			0	1	1

13	semSignalB(s)		1	1	1

Hypothetical Trace Continued

	Producer	Consumer	s	n	delay
13	semSignalB(s)		1	1	1

14		if (n == 0)
		semWaitB(delay)
			1	1	1

15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1

18		if (n == 0)
		semWaitB(delay)
			1	0	0

19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

A flaw in the given “solution” is exposed by the potential interaction here.

The Consumer should have called semWaitB(delay) after it exhausted the buffer at line 8, which would block the Consumer.

But, due to a context switch, the Producer was able to increment n first, and that prevented the Consumer from correctly blocking itself.

Infinite-buffer PC with Semaphores

```
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
```

```
void producer() {
   while (true) {
      produce();
      semWaitB(s);
      append();
      n++;
      if ( n == 1 )
         semSignalB(delay);
      semSignalB(s);
   }
}
```

```
void consumer() {
   semWaitB(delay);
   while (true) {
      semWaitB(s);
      take();
      n--;
      if ( n == 0 )
         semWaitB(delay);
      semSignalB(s);
      consume();
   }
}
```

```
int main() {
   n = 0;
   parbegin(producer, consumer);
}
```

What about this… just move the conditional statement inside the consumer's critical section?

Infinite-buffer PC with Semaphores

```
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
```

```
void producer() {
   while (true) {
      produce();
      semWaitB(s);
      append();
      n++;
      if ( n == 1 )
         semSignalB(delay);
      semSignalB(s);
   }
}
```

```
void consumer() {
   int m;
   semWaitB(delay);
   while (true) {
      semWaitB(s);
      take();
      n--;
      m = n;
      semSignalB(s);
      consume();
      if ( m == 0 )
         semWaitB(delay);
   }
}
```

```
int main() {
   n = 0;
   parbegin(producer, consumer);
}
```