## Concurrency

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical section</td>
<td>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</td>
</tr>
<tr>
<td>deadlock</td>
<td>A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something</td>
</tr>
<tr>
<td>livelock</td>
<td>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</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>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</td>
</tr>
<tr>
<td>race condition</td>
<td>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</td>
</tr>
<tr>
<td>starvation</td>
<td>A situation in which a runnable process is overlooked indefinitely by the scheduler; although it’s able to proceed, it is never chosen</td>
</tr>
</tbody>
</table>

### A Simple Example

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

```
//Process P1
.. chin = getchar();
.. chout = chin;
.. putchar(chout);
```

```
//Process P2
.. chin = getchar();
.. chout = chin;
.. putchar(chout);
.. putchar(chout);
```
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

<table>
<thead>
<tr>
<th>Degree of Awareness</th>
<th>Relationship</th>
<th>Influence that one Process Has on the Other</th>
<th>Potential Control Problems</th>
</tr>
</thead>
</table>
| Processes unaware of each other | Cooperation | Status of one process independent of the status of others | - Mutual exclusion
- Deadlock (incompatible resources)
- Contention |
| Processes unaware of each other (e.g., shared object) | Cooperation by sharing | Status of one process may depend on information obtained from others | - Mutual exclusion
- Deadlock (incompatible resources) |
| Processes do not share common work resources available to them | Cooperation by communication | Status of one process may depend on information obtained from other processes | - Deadlock (incompatible resources) |
| Processes do not share common work resources available to them | Cooperation by communication | Status of one process may depend on information obtained from other processes | - Deadlock (incompatible resources) |
### Competition Among Processes for Resources

<table>
<thead>
<tr>
<th>Concurrency 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mutual Exclusion</strong></td>
</tr>
<tr>
<td>- critical sections</td>
</tr>
<tr>
<td>- only one program at a time is allowed in its critical section</td>
</tr>
<tr>
<td>- example: only one process at a time is allowed to send command to the printer</td>
</tr>
</tbody>
</table>

| Deadlock |

| Starvation |

### Requirements for Mutual Exclusion

<table>
<thead>
<tr>
<th>Concurrency 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only one process at a time is allowed in the critical section for a resource</td>
</tr>
</tbody>
</table>

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
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

Test and Set Instruction
```c
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

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

Executed atomically, effectively as a single machine instruction.

---

Mutual Exclusion

```c
/* PROCESS 1 */
void p1 {
    while (true) {
        /* preceding code */
        enterritical (Ra);
        /* critical section */
        exitritical (Ra);
        /* following code */
    }
}

/* PROCESS 2 */
void p2 {
    while (true) {
        /* preceding code */
        enteritical (Rb);
        /* critical section */
        exitritical (Rb);
        /* following code */
    }
}

/* PROCESS N */
void pn {
    while (true) {
        /* preceding code */
        enteritical (Rn);
        /* critical section */
        exitritical (Rn);
        /* following code */
    }
}
```
Concurrency 11

Mutual Exclusion

Using test-and-set

```c
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void su(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]);
    }
```

Using exchange

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

Special variable called a \textit{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
- \textit{wait} operation decrements the semaphore value
- \textit{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

```c
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 \textit{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 \textit{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

Binary semaphores can only be 0 or 1.
That means they would achieve one-at-a-time mutual exclusion.

```c
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
    }
}

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.

```c
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

```c
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:**

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

**consumer:**

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

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

**Infinite-buffer PC with Semaphores**

```c
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

<table>
<thead>
<tr>
<th>Line</th>
<th>Action</th>
<th>s</th>
<th>n</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>n++</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>if ( n == 1 ) semSignalB(delay)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>semWaitB(delay)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>n--</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>n++</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>if ( n == 1 ) semSignalB(delay)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

## Hypothetical Trace Continued

<table>
<thead>
<tr>
<th>Line</th>
<th>Action</th>
<th>s</th>
<th>n</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>if ( n == 0 ) semWaitB(delay)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>n--</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>if ( n == 0 ) semWaitB(delay)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>n--</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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();
        if ( m == 0 )
            semWaitB(delay);
    }
}

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

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

What about this... just move the conditional statement inside the consumer’s critical section?