InterProcess Communication

Race Condition

Shared Data:

```
tail A[i]
```

Enqueue:

- `A[tail] = 20;`
- `tail++;`
- `A[tail] = 9;`
- `tail++;`

Process A      Process B

Critical Regions

- Process A enters critical region
- Process B tries to enter critical region
- Process B blocked
- Time
- Process A leaves critical region
- Process B enters critical region
- Process B leaves critical region

Goals

- No two processes (threads) can be in their critical region at the same time
- No assumptions about # of CPUs or their speed
- No process outside of its critical region may block another process
- No process should have to wait forever to enter its critical region

Strict Alternation

```
while (TRUE) {
    while (turn != 0) {
        // loop
    }
    critical_region();
    turn = 1;
    noncritical_region();
}
```

```
while (TRUE) {
    while (turn != 1) {
        // loop
    }
    critical_region();
    turn = 0;
    noncritical_region();
}
```

Busy Waiting

```
#define FALSE 0
#define TRUE 1
#define N 2 // # of processes
int interested[N]; // Set to 1 if process j is interested
int last_request; // Who requested entry last?

void enter_region(int process) {
    int other = 1-process; // # of the other process
    interested[process] = TRUE; // show interest
    while (interested[other] == TRUE && last_request == process)
        ; // wait while other process runs
}

void leave_region(int process) {
    interested[process] = FALSE; // I’m no longer interested
}
```
Hardware Support

Code for process Pi:
```c
while (1) {
    while (!TestAndSet(lock))
        // critical section
    lock = 0;
    // remainder of code
}
```

Code for process Pj:
```c
while (1) {
    while (Swap(lock, 1) == 1)
        // critical section
    lock = 0;
    // remainder of code
}
```

Producer/Consumer Problem

Shared variables:
```c
const int n;
typeinfo = Item;
int buffer[n], in = 0, out = 0,
counter = 0;
```

Atomic statements:
```c
counter ++;
counter --;
```

Producer:
```c
Item pitm;
while (1) {
    produce an item into pitm
    if (counter == n)
        sleep();
    buffer[in] = pitm;
in = (in + 1) % n;
    if (counter == 1)
        wakeup(consumer);
}
```

Consumer:
```c
Item citm;
while (1) {
    full.down();
    mutex.down();
    citm = buffer[out];
    if (counter == 0)
        wakeup(producer);
    consume item in citm
    mutex.up();
    empty.up();
    // consume item from
    // citm
}
```

Semaphore with Blocking

```c
class Semaphore {
    int value;
    ProcessList pl;
    void down () {
        value --;
        if (value < 0) {
            // add this process to pl
            pl.enqueue(currentProcess);
            Sleep();
        }
    }
    void up () {
        Process P;
        value ++;
        if (value > 0) {
            // remove a process P from pl
            P = pl.dequeue();
            Wakeup(P);
        }
    }
}
```

Producer/Consumer with Semaphores

```c
const int n;
Semaphore empty(n), full(n), mutex(1);
Item buffer[n];
```

Producer:
```c
int in = 0;
Item pitm;
while (1) {
    // produce an item
    // into pitm
    mutex.down();
    empty.down();
    mutex.down();
    buffer[in] = pitm;
in = (in + 1) % n;
    mutex.up();
    full.up();
}
```

Consumer:
```c
int out = 0;
Item citm;
while (1) {
    full.down();
    mutex.down();
    citm = buffer[out];
    if (counter == 0)
        wakeup(producer);
    consume item from
    citm
    mutex.up();
    empty.up();
    // consume item from
    // citm
}
```

Binary Semaphore

*Semaphore that only takes on the values 0 or 1*

Counting Semaphore
Mutex
A simplified version of a Semaphore that can only be locked or unlocked

Monitors

```
class ProducerConsumer {
    private static final int n;
    Item buffer[] = new Item[n];

    public synchronized Item consumer() {
        while (count == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.err.println("interrupted");
            }
        }
        cItm = buffer[out];
        out = (out + 1) % n;
        count--;
        if (count == 0) {
            // wake up the consumer
            notify();
        }
        return cItm;
    }

    public synchronized void producer() {
        while (count == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.err.println("interrupted");
            }
        }
        buffer[in] = pItm;
        in = (in + 1) % n;
        count++;
        if (count == n) {
            // wake up the producer
            notify();
        }
    }
}
```

Message Passing

Barriers

```
Processes approaching barrier
0 and D at barrier
All at barrier
Barrier releases all processes
```
Dining Philosophers