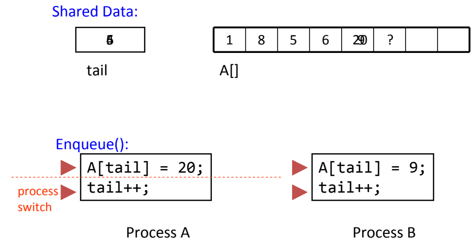
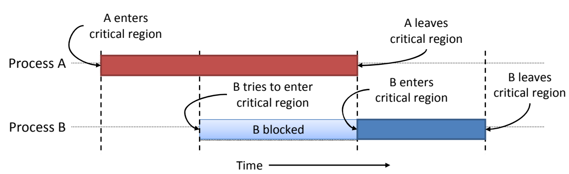
InterProcess Communication

Race Condition



Critical Regions



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

Process A

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

Process B

```
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
     last_request = process;              // Set it to my turn
     while (interested[other]==TRUE && last_request == process )
       ; // Wait while the other process runs
}

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

Hardware Support

```
int lock = 0;
```

Code for process P_i

```
while (1) {
  while (TestAndSet(lock))
    ;
  // critical section
  lock = 0;
  // remainder of code
}
```

Code for process P_i

```
while (1) {
  while (Swap(lock,1) == 1)
    ;
  // critical section
  lock = 0;
  // remainder of code
}
```

Producer/Consumer Problem

Shared variables

```
const int n;
typedef … Item;
Item buffer[n];
int in = 0, out = 0,
    counter = 0;
```

Atomic statements:

```
counter += 1;
counter -= 1;
```

Producer

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

Consumer

```
Item citm;
while (1) {
  if (counter == 0)
    sleep();
  citm = buffer[out];
  out = (out+1) % n;
  counter -= 1;
  if (count == n-1)
    wakeup(producer);
  consume the item in citm
  …
}
```

Semaphore with Blocking

```
class Semaphore {
    int value;
    ProcessList pl;

    void down () {
        value -= 1;
        if (value < 0) {
            // add this process to pl
            pl.enqueue(currentProcess);
            Sleep();
        }
    }

    void up () {
        Process P;
        value += 1;
        if (value <= 0) {
            // remove a process P from pl
            P = pl.dequeue();
            Wakeup(P);
        }
    }
}
```

Producer/Consumer with Semaphores

```
const int n;
Semaphore empty(n),full(0),mutex(1);
Item buffer[n];
```

Producer

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

Consumer

```
int out = 0;
Item citem;
while (1) {
  full.down();
  mutex.down();
  citem = buffer[out];
  out = (out+1) % n;
  mutex.up();
  empty.up();
  // consume item from
  // citem
}
```

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

Binary Semaphores

Shared variables

```
Semaphore mutex;
```

Code for process P_i

```
while (1) {
  down(mutex);
  // critical section
  up(mutex);
  // remainder of code
}
```

Monitors

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

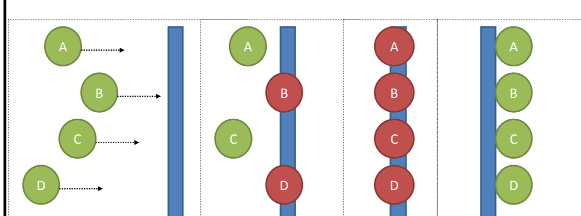
```
public synchronized Item consumer() {
  while (count == 0) {
    try {
      wait();
    }
    catch (InterruptedException e) {
      System.err.println("interrupted");
    }
  }
  cItm = buffer[out];
  out = (out + 1) % n;
  count-=1;
  if (count == n-1) {
    // wake up the producer
    notify();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

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

Locks and Condition Variables

Message Passing

Barriers



Processes approaching barrier | B and D at barrier | All at barrier | Barrier releases all processes

Dining Philosophers