Interprocess communcation

- in Operating systems we find there are a number of mechanisms used for interprocess communication (IPC)
- the IPC mechanisms can be divided into two groups, those which work well using shared memory and those which work with non shared memory
- some common methods of IPC are: sockets, semaphores and mailboxes
- sockets and mailboxes are normally used by non shared memory programs
  - ie client and server on different machines

Interprocess communication in shared memory systems

- semaphores are more appropriate for multiple processes sharing some common memory
- we will be covering a semaphores and message passing after networking with sockets
- message passing
  - can be used in shared memory systems
- this week we will look at Semaphores

Semaphores: shared memory interprocess communication

- processes within an operating system do not act in isolation
  - on the one hand they co-operate to implement an application
  - on the other hand they compete for resources, processor time, device access etc
- these two elements of co-operation and competition imply some form of communication between the processes

Semaphores: shared memory interprocess communication

- in effect there are two categories for interprocess communication
  - **mutual exclusion**
  - **synchronisation**
- **mutual exclusion**
  - some resources in an operating system are non sharable, maybe access to the sound card or access to the GPU
  - access needs to be granted to one process at a time
- **synchronisation**
  - processes run asynchroneously relative to each other
  - sometimes there will be points beyond which a process cannot proceed until another process has completed some activity

Mutual exclusion

- require a mechanism to ensure that only one process can manipulate data at any one time
  - *mutual exclusion*
- the concepts we discuss today are *very* important for operating systems
  - a fundamental building block

How do we implement mutual exclusion?

- simplest mechanism
  - mask processor interrupts off
  - processor cannot respond to any interrupt and therefore will execute code in sequence until it masks interrupt back on again
  - sometimes these critical sections of code are called *atomic*
  - what are this disadvantages with this approach?
  - what are this advantages with this approach?

How do we implement mutual exclusion?

- another mechanism is *semaphores*
  - essentially a binary *semaphore* is a token which can be grabbed by *only one* process at a time
  - a token is taken at the entry to the critical section and given back at the end of the critical section
  - a process can only enter once it has the token

Semaphores

- the most important single contribution towards interprocess communication was the introduction of **semaphores** by E.W. Dijkstra in 1965
  - a semaphore is a data type and the primitive operators are `wait` and `signal`
- these are the classic operators translated from Dutch words

Semaphores

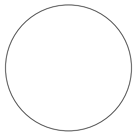
- consider the following two processes:
-

```
           /* Shared semaphore */
Semaphore token;  /* initial value 1 */

void ProcessA ()          void ProcessB ()
{                         {
   while (TRUE) {            while (TRUE) {
      ...                       ...
      Wait(Token)               Wait(Token)
      /* critical               /* critical
         region */                 region */

      Signal(Token)             Signal(Token)
      ...                       ...
   }                         }
}                         }
```


Semaphores

-

SEMAPHORE token



  - `Wait` gets the token
  - `Signal` returns the token

Semaphores

- note that `Wait` and `Signal` are both *atomic*
- they are implemented in software with processor interrupts masked off
- this allows us to build critical regions which can execute with processor interrupts on
- this is overall efficient as we only have to mask processor interrupts off during the execution of `Wait` and `Signal`
  - this time should be short compared with the time to execute the critical region

Semaphores

- we can express `Wait` and `Signal` in pseudo code:
-

```
void Wait (s)
{
   when s>0
      s--;
}

void Signal (s)
{
   s++;
}
```

Semaphores

- in our previous example the initial value of s would be 1
  - note that this is pseudo code
  - note the use of **when**

Semaphores

- we have now seen how a critical section can be achieved by using semaphore primitives `Wait` and `Signal`
- for example access to the shared buffer will be a critical section

Starting to implement a shared buffer using semaphores

```
void put (char ch)        char get (void)
{                         {

   Wait(Mutex)               Wait(Mutex)
   /* safe to alter */       /* safe to alter */
   /* buffer        */       /* buffer        */
   place ch into buf         remove ch from buf

   Signal(Mutex)             Signal(Mutex)

                             return ch;
}                         }

      char buffer[Max];   /* global data */
      SEMAPHORE Mutex;    /* global data */
```

- we will return to this code next week

Implementing synchonisation with a Semaphore

```
void ProcessA ()                    void ProcessB ()
{                                   {
   while (TRUE) {                      while (TRUE) {
      ...                                 ...
      Wait(sync)                          /* point B */
      /* process B reached                Signal(sync)
         point B.  */                     ...
      ...                              }
   }                                }
}
```

Python Semaphores and Threads

- in python you can create threads and create semaphores
  - there are a number of Python primatives which operate on semaphores but we will concentrate on those which map onto `Wait` and `Signal`

Python Semaphores and Threads

- semaphores can be created and used by:

```
from thread import start_new
from threading import Semaphore

Mutex = Semaphore(value=1)

Mutex.acquire() # Wait

Mutex.release() # Signal
```

- a thread can be created by using `start_new`

Example in Python of two threads synchronising

- **simplesync.py**

```
#!/usr/bin/env python

import sys, time
from thread import start_new
from threading import Semaphore

sync = Semaphore(value=0)

def processA (p, count):
    global sync
    print "processA", p, "comes to life"
    while True:
        time.sleep (5)  # do some work
        sync.release()  # indicate we have finished our w
```


Example in Python of two threads synchronising

- **simplesync.py**

```
def processB (p, count):
    global sync
    print "processB", p, "comes to life"
    while True:
        print "waiting for process A to complete its work
        start_time = time.time()
        sync.acquire()
        end_time = time.time()
        print "processB", p, "spent", end_time - start_ti

def main ():
    start_new(processA, (1, 0))
    processB (2, 0)

main ()
```

Example in Python of two threads implementing mutual exclusion

- **simplemutex.py**

```
#!/usr/bin/env python

import sys, time
from thread import start_new
from threading import Semaphore

mutex = Semaphore(value=1)
n = 0  # global variable which will be incremented and
       # decremented inside the critical region
```


Example in Python of two threads implementing mutual exclusion

- **simplemutex.py**

```
def process (p, count):
    global mutex, n
    print "process", p, "comes to life"
    while True:
        start_time = time.time()
        print "process", p, "waiting to enter"
        mutex.acquire()
        end_time = time.time()
        print "process", p, "spent", end_time - start_time, "seconc
        # critical region
        n += 1
        if n != 1:
            print "something has gone very wrong!"
            sys.exit (1)
        time.sleep (5)
        n -= 1
        mutex.release()
        print "process", p, "finished critical region"
```


Example in Python of two threads implementing mutual exclusion

- **simplemutex.py**

```
def main ():
    for i in range (3):
        start_new(process, (i, 0))
    process (4, 0)

main ()
```