3F6 - Software Engineering and Design

Handout 13

Concurrent Systems II

With Markup

Edward Rosten

Contents

1. Critical Sections
2. Mutexes
3. Signals
4. Monitors
5. Semaphores
6. Pipes and messages

Copies of these notes plus additional materials relating to this course can be found at: `http://mi.eng.cam.ac.uk/~er258/teaching`.

Critical Sections

Concurrent access to shared writable resources causes *race conditions*.

```
int i;
// thread 1   |  // thread 2;
++i           | ++i            ← critical section
              |
              |

//In assembler
              |
LDAA i        | LDAA i
INCA          | INCA
STAA i        | STAA i
```

There is a race condition in the update of the date. Only one thread can sefely access this resource simultaneously. Sections of code using this resource are critical sections.

A lock is often called a *mutex* since it ensures *Mut*ual *ex*clusion of a region of code.

Critical sections can be protected with locks. Without care, acquisition of locks can cause deadlock. One way of preventing deadlock:

1. Number locks sequentially
2. Record last lock acquired
3. Acquiring a lock with a lower number is an error

A *critical section* is a part of the code which accesses a shared resource.

Any access to the critical section should ensure the following:

- **Mutual exclusion**. Only one thread at a time may enter the critical section;
- **Fairness**. Each thread trying to enter the critical section must eventually succeed;
- In the **Absence of contention** a single thread wishing to enter a critical section must succeed, ideally with minimal delay;

In addition, the system should be as **efficient** as possible.

- any thread which is blocked from entry to a critical section should not waste CPU

Solutions to this problem generally utilise low-level *system calls* provided by the operating system. Blocked threads are suspended on an event queue and resumed when it is their turn to enter the critical section.

Low-level code within the operating system will often waste CPU for a very short amount of time.

A Useless Access Control Mechanism?

```
bool avail;

void Lock(){
  while(avail == false);
  avail = false;
}

void Unlock()
{
  avail = true;
}
```

```
bool test_and_set(){
   bool old = avail;
   avail = false;
   return old;
}
void Lock(){
    while(test_and_set());
}
```

```
int i;
```

```
//Thread 1
Lock(avail);
i++;
Unlock(avail);
```

```
//Thread 2
Lock(avail);
i++;
Unlock(avail);
```

Now **`avail`** is the writable shared resource. With some modification, however, it does work!

1. Disable interrupts in **`Lock`** and **`Unlock`**. This is not sufficient for multiprocessor systems.
2. Use atomic operations.

Peterson's mutex algorithm

Pure software solutions exist to the mutex problem:

```
bool intent1 = 0, intent2 = 0
int turn;
```

```
//Thread 1                         //Thread 2

intent1 = 1;                       intent2 = 1;
turn = 2;                          turn = 1;
while(intent2 && turn == 2);       while(intent1 && turn == 1);

//Critical section                 //Critical section

intent1 = 0;                       intent2 = 0;
```

This algorithm does not work on modern multiprocessor systems because they are allowed to re-order instructions including writes to memory.

Signals - the user's perspective

Suppose that a thread needs to execute a single processing cycle every time that a user presses a key (e.g. to update a grammar checker in the background).

```
Signal keypress;
bool done = false;
// ---------------------------------------
void GrammarChecker(int i) {
   do {
      keypress.wait();
      // update grammar checking
      ...
   } until (done);
}
// ---------------------------------------
// Main thread
Thread gcheck = create(GrammarChecker,0,low);
while (inputting) {
   char ch = GetKey();
   result = Process(ch);
   if (result == error) {
      kill(gcheck);  reportError();
   }
   keypress.send();
}
done = true;
join(gcheck);
...
```

Running the grammar check as a low priority thread allows complex computation to be done in the background without spoiling user response times.

Signals - the OS perspective

Using a signal allows a thread to suspend itself (**wait**) until another thread sends it that signal (**send**). Like a semaphore, a signal is an operating system defined data type:

```
class Signal {
public:
   void send();
   void wait();
private:
   ThreadQueue q;
}
```

Draw q holding a list of process records. Each waiting to be signalled.

```
void Signal::wait()
{
   put caller's thread record on q;
   resume next thread in Ready queue;
}

void Signal::send()
{
   if (q is not empty) {
      remove next thread from q;
      place it in Ready queue;
   }
}
```

Note that `if (q is not empty)` could be `while (q is not empty)`
Need to check the exact semantics of actual implementation.

Semaphores - the user's perspective

Semaphores are a classic solution to the mutex problem. A semaphore counts available resources. Attempts to acquire resources when none remain blocks. A critical section has a single available resource: which thread (if any) is currently executing.

```
Semaphore s = 1;

//Thread 1                    //Thread 2

acquire(s);                   acquire(s);
//critical section            //critical section
release(s)                    release(s)
```

Operations:

- *Acquire* Waits while `s == 0`, then decrements s.
- *Release* Incerments s.

Alternative names:

Acquire/Release, Wait/Signal, Pend/Post, Enter/Leave, Procure/Vacate, P/V, Verhogen/Prolaag

Semaphores can be implemented using mutexes and busy waiting, but this is inefficient.

Semaphores - the OS perspective

```
class Semaphore {
public:
   void acquire();
   void release();
private:
   int remaining;    //Initialized to 1 for mutexes
   ThreadQueue q;
}

void Semaphore::acquire()
{
   if (remaining > 0) {
      remaining--;
   } else {
      put caller's thread record on q;
      Schedule next thread in Ready queue;
   }
}
void Semaphore::release ()
{
   if (q is empty) {
      avail++;
   } else {
       Move thread from q to Ready
   }
}
```

Access to `remaining`, `q` and `Ready` must be protectet on multi-processor systems.

Monitors

```
class Semaphore {
public:
  void acquire(){
    m.lock();
      if(remaining == 0)
        more.wait( m );
      remaining--;
    m.unlock();
  }

  void release(){
    m.lock();
    remaning++;
    more.signal();
    m.unlock();
  }
private:
  int remaining;
  Mutex m;
  Signal more;
}
```

This will deadlock unless signals temporarily release the mutex.

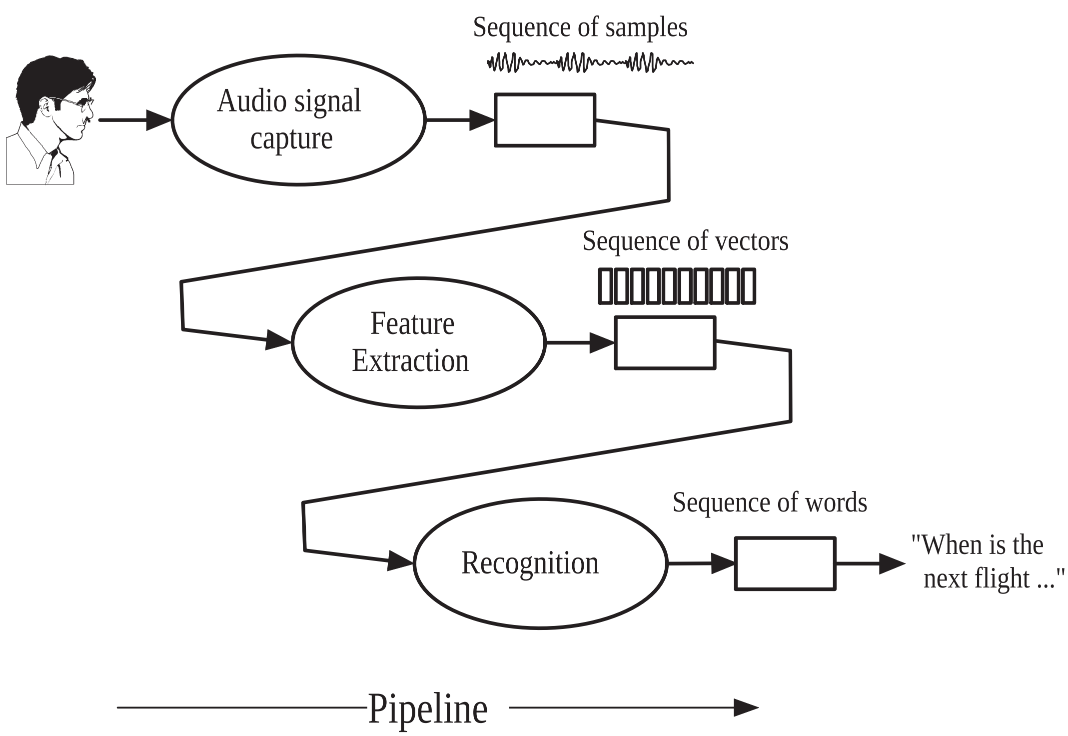
This is an example of a *monitor*

Monitors are classes which have every method protected by a lock. These are the synchronization primitive provided in Java.

Pipeline communication

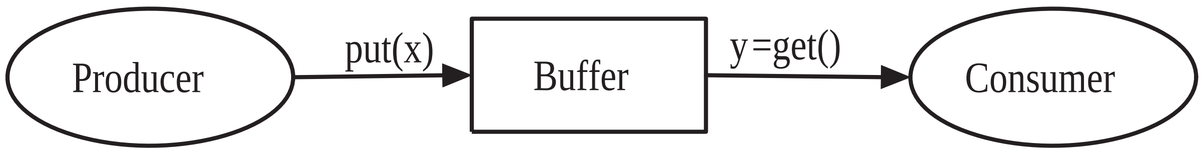
Thread and processes are often set-up as pipelines with the output of one is passed as input to the next. This can often simplify design since:

- Synchronization is performed only on the pipe.
- No deadlocks if the pipeline does not double back.



Communication between threads uses *bounded buffers*.

Bounded Buffers



Assume x and y are of type Datum:

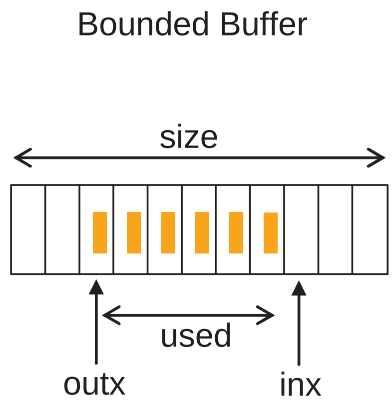
- Buffer is a bounded first in, first out queue. It can hold at most N items of type Datum.
- Buffer has two principal operations:
  1. `put(x)` store item x in buffer
  2. `get()` return next item from buffer
- Buffer allows consumer and producer to proceed asynchronously
- Producer only has to stop when buffer is full
- Consumer only has to stop when buffer is empty

To implement such a buffer, the calls to `put` and `get` must be mutually exclusive since they access a shared memory buffer.

When the buffer is full or empty, the caller must wait for an appropriate *notfull* or *notempty* signal.

Implementation of the Buffer class using a Monitor:

```
class Buffer {
public:
        void put(Datum x);
        Datum get();
private:
        const int size = N;
        Datum buf [size];
        int inx,outx,used;
        Signal notfull, notempty;
        Semaphore lock;
}
```




```
void Buffer::put(Datum x) {

        lock.enter();

        while (used == size)
           notfull.wait ();
        buf [inx] = x;
        inx = (inx+1) % size;
        ++used;
        notempty.send ();

        lock.leave();

}
```

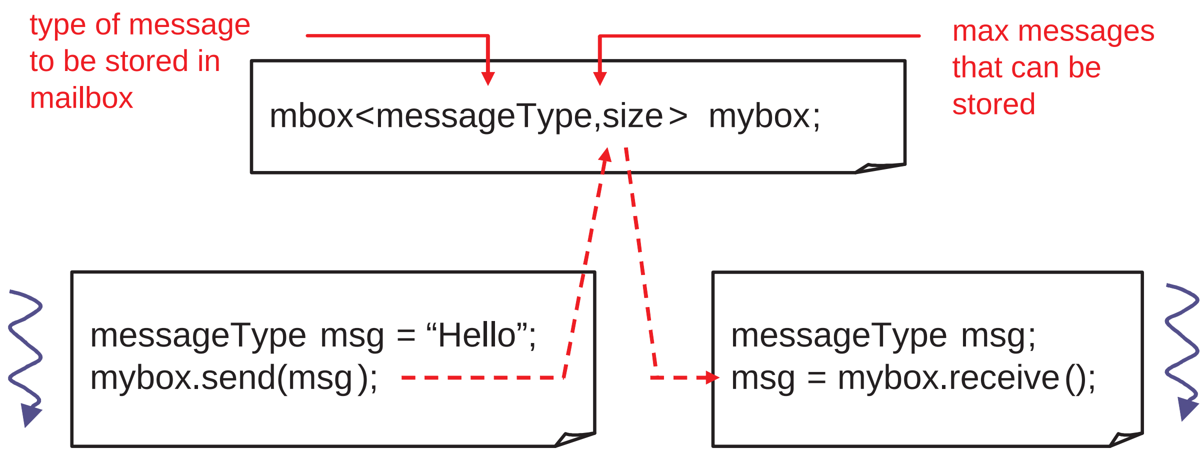
```
Datum Buffer::get() {
        Datum x;
        lock.enter();

        while (used == 0)
            notempty.wait ();
        x = buf [outx];
        outx = (outx+1) % size;
        --used;
        notfull.send ();

        lock.leave();
        return x;
}
```

Message Passing

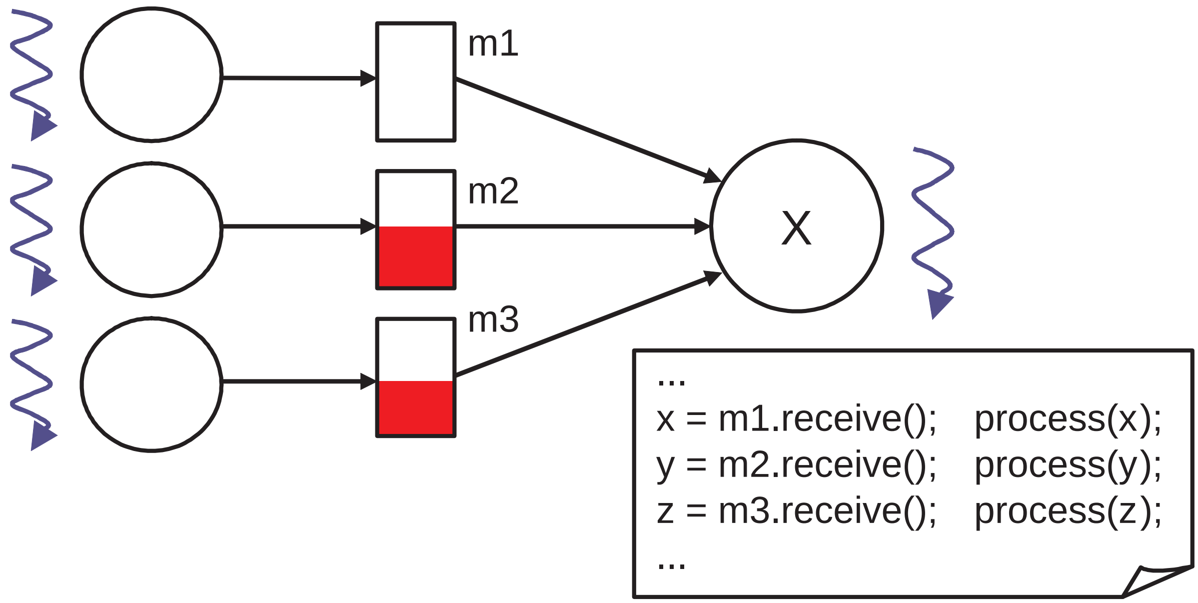
The use of bounded buffers to connect asynchronous threads is so common that some systems provide a bounded buffer as the basic primitive for communication then the buffer is called a mailbox.



- when mailbox is full - sender blocks
- when mailbox is empty - receiver blocks

Sometimes this can be a problem ...

Consider a thread that is processing messages from several sources:



How can thread X avoid blocking on an empty mailbox whilst other boxes have data ready for processing?

We could check how many messages a mailbox holds before calling `receive()`, but this results in inefficient polling.

The Select Statement

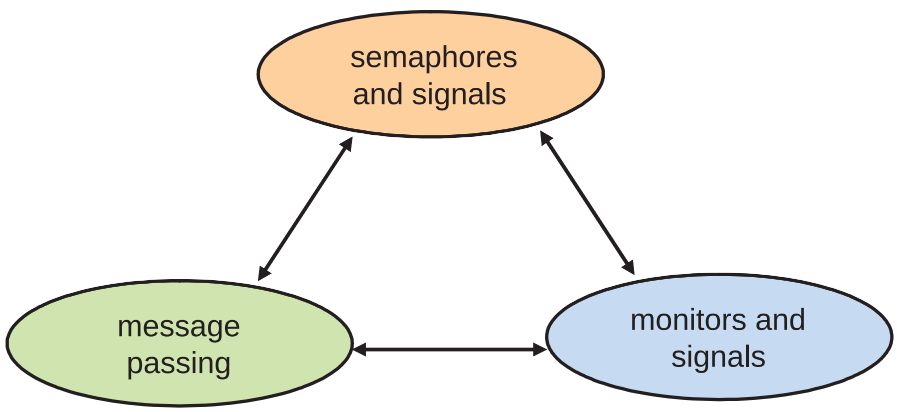
Systems which have message passing as a built-in feature solve the problem by providing a `select` statement:

```
select(m1, m2, m3) {
   m1 =>
      x = m1.receive();  process(x);
      break;
   m2 =>
      y = m2.receive();  process(y);
      break;
   m3 =>
      z = m3.receive();  process(z);
      break;
}
```

If one or more mailboxes are nonempty, then one of the branches is selected. Otherwise, the caller waits and selects the first message to arrive.

Which concurrency mechanism is best?

All of the three approaches to handling mutual exclusion and event notification are orthogonal. Given one you can implement the others:



Semaphores and signals used to be maligned, but the expressive power of languages such as C++ allows any desired mechanism to built on top of any basic primitive.

Low-level synchronization is required where there is shared memory. Examples are inside multi-threaded code or inside an operating system. The largest shared memory machine has 1024 CPUs.

Messages easily generalize to large distributed systems where messages are sent across a network. Most supercomputer software uses message passing.

Summary

- Concurrency is essential for providing real-time interaction with asynchronous external processes (eg humans, control system, etc).
- Concurrency is essential for high performance computing.
- Unlike processes, threads share the same memory space and thereby allow very efficient real-time operation.
- Safe communication between processes/threads requires explicit support:
  - semaphores and signals
  - monitors and signals
  - message passing
- Traditionally semaphores have been criticised as being too low level and error prone, however, object-oriented languages such as C++ allow critical sections to be safely encapsulated.
- Message passing often simplifies concurrent code because it reduces the number of shared writable resources.