Handout 14
Concurrent Systems II
With Markup

Steve Young
Contents

1. Critical Sections
2. Semaphores and Signals
3. Monitors
4. Message Passing

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

Consider the use of shared memory.

The program fails if the consumer thread reads \texttt{today} whilst it is being updated by the producer thread.

Sections of code which are involved in accessing shared resources are called \textit{critical sections}. 
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;
- **No deadlock.** If several threads are trying to enter the critical section, one must succeed;
- **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;
- **Immunity to failure.** The failure of one thread must not jeopardise the operation of the other threads.

In addition, the system should be as **efficient** as possible.
A Useless Access Control Mechanism

Here is a solution that does not work!

- memory contention is not solved, it is just shifted to the avail flag
- waiting threads have to poll the avail flag which is wasteful and does not guarantee fairness.
Solutions to the Mutex Problem

An efficient and safe control mechanism to solve the mutual exclusion (mutex) problem must satisfy three main requirements:

- only one thread must be allowed to enter a critical section at any one time
- access to the control mechanism must itself be mutually exclusive
- any thread which is blocked from entry to a critical section must not waste cpu

All solutions to this problem 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.
**Semaphores**

The *Semaphore* is the classic solution to the mutex problem

class Semaphore {
public:
    void enter();
    void leave();
private:
    bool avail; // initially true
    ThreadQueue q;
}

void Semaphore::enter()
{
    if (avail) {
        avail = false;
    } else {
        put caller’s thread record on q;
        resume next thread in Ready queue;
    }
}

void Semaphore::leave ()
{
    if (q is empty) {
        avail = true;
    } else {
        remove next thread from q;
        place it in Ready queue;
    }
}
Using semaphores to protect the critical section provides a safe implementation of the “Today” problem:

However, mutual exclusion is not the only problem requiring solution. Efficient multiprogramming also requires some mechanism for a thread to notify another thread that some specific event has occurred.
Event Signalling

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

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

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

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

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

Note that \texttt{if (q is not empty)} could be \texttt{while (q is not empty)}

Need to check the exact semantics of actual implementation.
**Data Pipelines**

Threads are often set-up as pipelines with the output of one thread passed as input to the next. This can often simplify design since:

- each thread can run at full speed without waiting for other threads
- real-time response conditions are easier to manage if individual program components run (mostly) independently.

Example - a speech recogniser

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.
The bounded buffer example highlights a small problem regarding the interaction of semaphores and signals.

When waiting for a signal inside a critical section, the operating system must ensure that the semaphore is released, and then re-acquired when the signal is received. ******
Monitors

Semaphores and signals are low level operating system primitives which are workable and efficient. However, they are easy to abuse.

Monitors are a high level language mechanism designed to replace semaphores. They are a type of object for which mutually exclusive access is guaranteed.

Monitors still rely on low level signals for synchronisation!

Note that monitors provide the basis for thread support in Java.
Implementation of the Buffer class using a Monitor:

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

void Buffer::put(Datum x) {
    while (used == size)
        wait(notfull);
    buf[inx] = x;
    inx = (inx+1) % size;
    ++used;
    send(notempty);
}

Datum Buffer::get() {
    Datum x;
    while (used == 0)
        wait(notempty);
    x = buf[outx];
    outx = (outx+1) % size;
    --used;
    send(notfull);
    return x;
}
```

The buffer operations `put` and `get` can now only be invoked serially so there is no need for semaphores.
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:

```
x = m1.receive();   
process(x
 );
y = m2.receive();   
process(y
 );
z = m3.receive();   
process(z
 );
...
```

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

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

```plaintext
select {
    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 at random. Otherwise, the caller waits and selects the first message to arrive.

This is the solution adopted in Ada which uses the Ren dezvous model. This a mailbox variation where size==0 always!

(Based on the mathematical model : Communicating Sequential Processes by Tony Hoare)
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. From this view, semaphores and signals are arguably the best because they are the lowest level and hence most flexible and efficient.
Summary

- Concurrency is essential for providing real-time interaction with asynchronous external processes (e.g., humans, control system, etc).

- 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.