3F6 - Software Engineering and Design

Handout 13
Concurrent Systems I
With Markup

Steve Young
Contents

1. Concurrency
2. Processes
3. Context Switching and Scheduling
4. Threads
5. Multi-threaded Programming

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

A concurrent system consists of several elements which operate at the same time and communicate with each other.

<table>
<thead>
<tr>
<th>Class of system</th>
<th>Examples</th>
</tr>
</thead>
</table>
| Distributed System/Transaction System | • BA Flight Reservation System  
                                         • HSBC Internet Banking System |
| Web-based System             | • Second Life                                 |
| Operating System             | • Linux or Windows                            
                                         • Mobile phone                         |
| Real time system             | • Process control system                      
                                         • Engine management system             |
| Applications                 | • Microsoft Office                            |
| Embedded system              | • Dishwasher                                  
                                         • Video recorder                        |
Levels of Concurrency

Concurrency may be viewed at the logical level and the physical level. There is a rough correspondence between the two but it is not exact.

**Physical**

- Networks
- Computers
- CPUs
- Processes
- Threads

**Logical**

- Distributed Systems/
  - Database Systems
- Operating Systems
- Realtime Systems
- Programs

Communication is via
- shared memory
- local databusses
- external network connections
Why is Concurrency an issue?

Concurrency is essential for

- efficient operation of large scale systems
- design of real time systems

Concurrent systems must communicate with each other, and share resources. However, uncontrolled concurrency leads to errors:

- simultaneous access to the same memory location leads to unpredictable results.

```c
struct Time { int hr, int min } t;
// thread 1
++t.min
if (t.min==60) {
  t.min=0; ++t.hr;
}
// thread 2;
mytime.min = t.min
mytime.hr = t.hr
```
- simultaneous use of shared resources leads to errors.

```c
// thread 1           // thread 2;
f=open("printer");  g=open("printer");
print(f,"hello");   print(g,"hello");
```

- uncontrolled allocation of shared resources can generate deadlock

```c
// thread 1           // thread 2;
secure("printer");  secure("scanner");
secure("scanner");  secure("printer");
```

These concurrency problems were also highlighted in the lecture on Transaction Processing.
Implementation of Concurrency

1. At the cpu level and above, concurrency arises naturally since there are multiple physical processors. This is real concurrency.

2. Below the cpu level, a single cpu is shared amongst many software processes. This is simulated concurrency.

Why do we simulate concurrency if it creates problems?

Because overall it greatly simplifies many real world applications

- real time systems which have to respond to external asynchronous events
- desktop window-based applications which have to do several tasks at once

These lectures will focus on mainly on simulated concurrency, how it is implemented on a single processor, and the programming constructs and methods that allow concurrency to be used safely and effectively.

Note that the programmer should always assume that the concurrency is real! eg Core Duo
Processes and Execution Context

When a program is executing, the program state is determined by memory, the cpu state and the contents of its registers.

- **Program Stack** Part of the memory is allocated for procedure variables, parameters and return values. This memory is called the *Program Stack* since it grows and shrinks as procedures call and return.

- **Stack Pointer (SP)**. The SP is stored in the CPU and it points to the current top of the program stack. When a procedure is called, the SP is decremented by an amount equal to the total local memory requirement of that procedure. When the procedure exits, the SP is incremented by the same amount.

- **Program Memory** The program instructions are stored in the *Program Memory* which is often Read-only memory (ROM).

- **Program Counter (PC)** The PC points to the next instruction to execute in program memory. When each instruction completes, the PC is normally incremented automatically by the size of that instruction. Exceptions are jumps and procedure calls, in which case the PC is loaded with the address of the instruction to jump to or call.
If we make a copy of the execution context, the single physical CPU can execute multiple processes by switching between them.
Context Switching

Context Switch: (a) save current cpu reg; (b) load new cpu reg
**Process States**

Each process is represented by a Process Record.

A process can be in one of three possible states:

- **Running**
- **Blocked**
- **Ready**
- **Wait for event e**
- **Scheduled**
- **Preempted**

The Operating System (OS) maintains a number process record queues

- **ready queue** (every process except running process is
- **one queue for each possible event** in one of these queues)

The OS schedules processes by moving them

1. from the ready queue to the physical cpu;
2. from the physical cpu to the ready queue;
3. from the physical cpu to an event queue.
Process Scheduling Example

This example illustrates pre-emptive scheduling. Embedded real time systems often use cooperative non-preemptive scheduling
Process Scheduling Policy

Process scheduling policy is determined by the Operating System.

The scheduling strategy tries to fulfil the following criteria:

- fairness: every process gets a reasonable slice of cpu time
- efficiency: keep processor busy as much as possible
- response time: avoid holding a process in ready queue for too long
- control: allow programmer reasonable control

Programmer control is usually provided by allowing processes to be assigned priorities.

Note that priorities should never be used to solve “race conditions”.
Execution Context Revisited

For completeness, it should be noted that there are other areas of memory used by processes:

- **Heap**: Stores dynamically allocated objects. It grows and shrinks under the control of a garbage collector.
- **Global Memory**: This is a fixed size area used to store globally allocated variables.
- **OS Interface**: Used by the OS to pass/store execution environment parameters.

MMR = Memory Management Register.
Processes and Threads

Process switching is computationally expensive since

- Every process has its own memory space and when a process is switched, memory management will often need to bring in cached memory from disk
- Communication between processes must use the OS and/or Middleware (Eg shared files, sockets, remote procedure calls). This can be cpu intensive.

A process can be divided into threads:

- Threads share the same memory space
- Context switching is cheap since only the PC and registers are switched
- Communication is simple and efficient via shared memory

Threads can be thought of as lightweight processes and most multi-programming on modern OS’s involves using threads. Multiprogramming at the process-level is usually done using CORBA or Microsoft .NET.

The presentation here will therefore concentrate on threads.
Multi-threaded Programming

Every operating system has its own set of primitives for multi-threaded programming. The names and details will vary but all provide similar core primitives.

Threads are defined like functions, but instead of calling them a thread is *spawned* using `create` and then later merged using `join`.

```c
// define a thread as a function
void ChildThread(int i) {
    // define thread operations
}

// ------- main thread -------
Thread t = create(ChildThread, 0, normal);
// ChildThread is now executing in parallel
// with main thread at normal priority.
...
...
// wait for ChildThread t to terminate
join(t);
// now there is just one thread again
....
```

Other commonly provided thread management functions include:

- `kill(t);`  // kill thread t
- `pause(n);`  // pause calling thread for n msecs
- `exit();`  // calling thread terminates
- `self();`  // a reference to the caller
Example - Watchdog timer

const int N = 10;
Thread wd, t[N];
bool ok[N];

// Watchdog thread - checks workers are alive
void Watchdog(int i) {
    pause(1000);
    for (int i=0; i<N; i++) {
        if (!ok[i]) {
            kill(t[i]); RaiseAlarm(i);
        }
        ok[i]=false;
    }
}

// Worker threads - do the hard work
void Worker(int i) {
    DoCompute1(); ok[i] = true;
    DoCompute2(); ok[i] = true;
    ...
}

// main program
void main() {
    for (int i=0; i<N; i++) {
        t[i] = create(Worker,i,normal);
        ok[i] = false;
    }
    wd = create(Watchdog,0,high);
    // wait here for all workers to stop
    for (int i=0; i<N; i++) join(t[i]);
    kill(wd);
}