CS 162 Operating Systems and Systems Programming

Professor: Anthony Joseph

Spring 2001

Lecture 3: Concurrency: Processes, Threads, and Address Spaces

2.0 Main point:

What are processes?
How are they related to threads and address spaces?

2.1 Concurrency

2.1.1 Definitions

Uniprogramming: *one process at a time* (e.g., MS/DOS, Macintosh)

Easier for operating system builder: get rid of problem of concurrency by defining it away. For personal computers, idea was: one user does only one thing at a time.

Harder for user: can't work while waiting for printer

Multiprogramming: *more than one process at a time* (UNIX, OS/2, Windows NT) (Often called multitasking, but multitasking sometimes has other meanings – see below – so not used in this course).

2.1.2 The basic problem of concurrency:

- Hardware: single CPU, I/O interrupts.
- API: users think they have machine to themselves.

OS has to coordinate all the activity on a machine: multiple users, I/O interrupts, etc.

How can it keep all these things straight?

Answer: Decompose hard problem into simpler ones. Instead of dealing with everything going on at once, separate into logical abstractions that we can deal with one at a time.

2.2 Processes

The notion of a "process" is a central concept for Operating Systems.

Process: *Operating system abstraction to represent what is needed to run a single program (this is the traditional UNIX definition)*

Formally, a process is a sequential stream of execution in its own address space.

2.2.1 Two parts to a (traditional Unix) process:

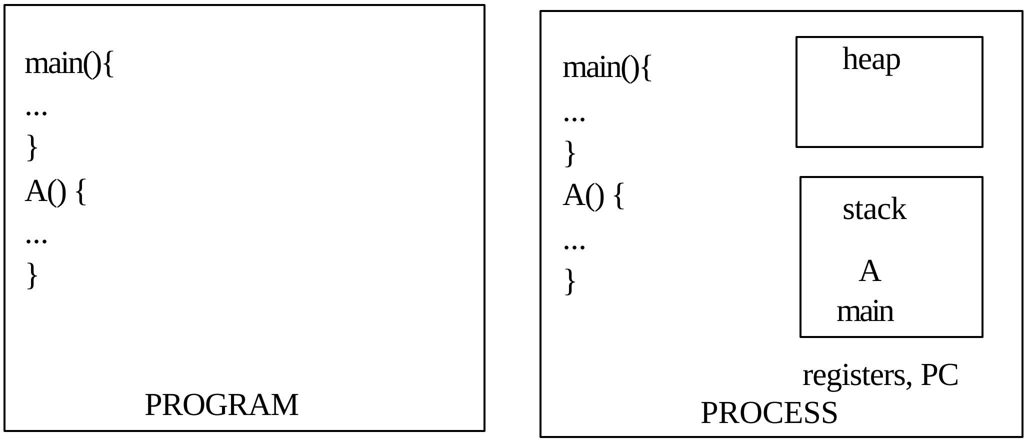
1. Sequential program execution: the code in the process is executed as a *single*, *sequential* stream of execution (no concurrency inside a process). This is known as a thread of control.

2. State Information: everything specific to a particular execution of a program: *Encapsulates protection: address space*

- CPU registers
- Main memory (contents of address space)
- I/O state (in UNIX this is represented by file descriptors)

2.2.2 Process =? Program

A **program** is, for example, a set of C statements or commands (vi, ls)



1. More to a process than just a program:
   - Program is just part of process state.
   - I run emacs on lecture.txt, you run emacs on homework.c – same program, different processes.
2. Less to a process than a program:
   - A program can invoke more than one process to get the job done
   - cc starts up cpp, cc1, cc2, as, ld (each are programs themselves)

2.3 Multiple Threads of Control

The traditional notion of a Process can be extended to allow for additional concurrency:

Thread: *a sequential execution stream within a process* (concurrency) (Sometimes called: a "**lightweight**" process.). Provides the illusion that each activity (or thread) is running on its own CPU, entirely sequentially.

Address space: all the state needed to run a program (literally, all the addresses that can be touched by the program). Provides the illusion that a program is running on its own machine (protection).

2.3.1 Why separate the concept of a thread from that of a process?

1. Discuss the "thread" part of a process (concurrency), separately from the "address space" part of a process (protection).

2. Many situations where you want multiple threads per address space. *Question: Why would you want this?*

Multithreading: *a single program made up of a number of different concurrent activities* (sometimes called multitasking, as in Ada, just to be confusing!)

2.3.2 Examples of multithreaded programs

1. Embedded systems: elevators, planes, medical systems, wristwatches, etc. Single program, concurrent operations.

2. Most modern OS kernels: internally concurrent because have to deal with concurrent requests by multiple users. But no protection needed within kernel.
3. Database Server: provides access to shared data by potentially many concurrent users. Also has background utility processing that must get done.

4. Network servers: user applications that get multiple requests concurrently off the network. Again, single program, multiple concurrent operations (examples: file servers, Web server, and airline reservation systems)

5. Parallel programming: split program into multiple threads to make it run faster. This is called **multiprocessing**.

Multiprogramming = multiple jobs or processes
Multiprocessing = multiple CPUs

Some multiprocessors are in fact uniprogrammed – multiple threads in one address space, but only run one program at a time.

2.3.3 Thread State

What state does a thread have?

Some state *shared by all threads* in a process/address space:

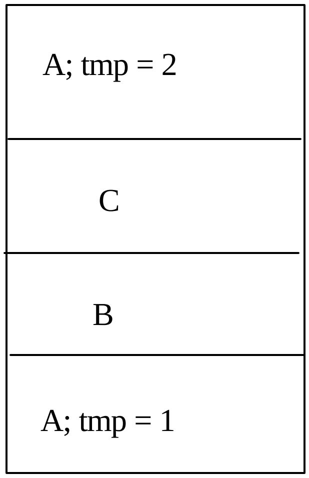
- Contents of memory (global variables, heap)
- I/O state (file system)

Some state *"private" to each thread* – each thread has its own copy

- CPU registers (**including, program counter**)
- Execution stack – *what is this?*

Execution stack: where parameters, temporary variables, and return PC are kept, while called procedures are executing (for example, where are A's variables kept, while B, C are executing?)

```
A(int tmp) {
   if  ( tmp< 2)
    B();
 printf(tmp);
}
B() {
    C();
}
C() {
  A(2);
}
A(1);
```



Execution stack

Threads encapsulate concurrency; address spaces encapsulate protection:

Keeps a buggy program from trashing everything else on the system.

Address state is passive; thread is active

2.4 Classification

Real operating systems have either

- one or many address spaces
- one or many threads per address space

# of address spaces: # of threads per address space:	one	many
One	MS/DOS, Macintosh	traditional UNIX
Many	embedded systems JavaOS, Pilot (PC)	Mach, OS/2 Windows 95, Windows NT, Solaris, Linux, HP-UX, ...

Examples:

1. MS/DOS – one thread, one address space
2. Traditional UNIX – one thread per address space, many address spaces
3. Mach, Microsoft NT, new UNIX (Linux, Solaris, HPUX) – many threads per address space, many address spaces
4. Embedded systems (Geoworks, VxWorks, JavaOS, etc.). Also, Pilot (the operating system on the first personal computer ever built) – many threads, one address space (idea was: no need for protection if single user)

2.5 Summary

Processes have two parts: threads and address spaces.

Book talks about processes: when this concerns concurrency, really talking about thread portion of a process; when this concerns protection, really talking about address space portion of a process.