Lecture 2: Concurrency: Threads, Address Spaces and Processes

2.0 Main point:

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

2.1 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 so deal with one at a time.

2.2 Processes

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 process:

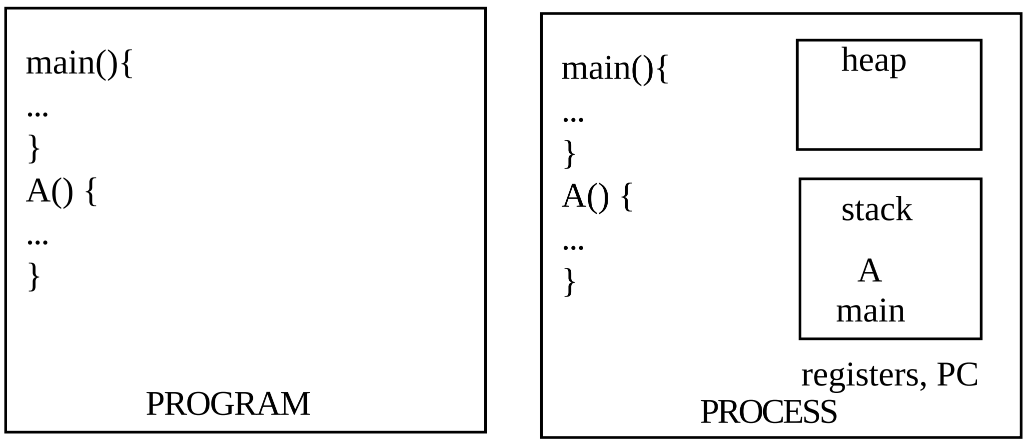
1. **sequential execution:** No concurrency inside a process -- everything happens sequentially.

2. process state: everything that interacts with process.

registers
main memory
files in UNIX

2.2.2 Process =? Program

A **program** is C statements or commands (vi, ls)



1. More to a process than just a program:

   program is just part of process state.

   I run ls; you run ls -- 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 (each are programs themselves)

2.2.3 Definitions

Uniprogramming: one process at a time (ex: 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)
(often called multitasking, but multitasking sometimes has other meanings -- see below -- so not used in this course).

2.3 Threads

Thread: a sequential execution stream within a process (concurrency) (Sometimes called: a "lightweight" process.)

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

2.3.1 Why separate these concepts?

1. Discuss the "thread" part of a process, separately from the "address space" part of a process.

2. Many situations where you want multiple threads per address space.

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

4. 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:

For example: contents of memory (global variables, heap), file system

Some state "private" to each thread -- each thread has its own copy

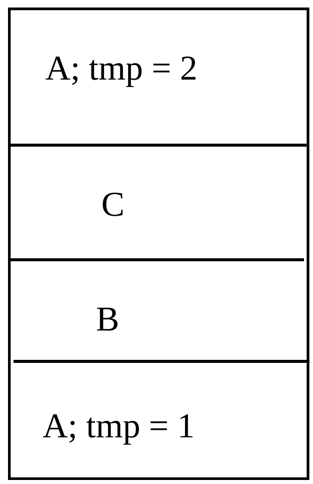
Program counter

Registers

Execution stack -- *what is this?*

Execution stack: where parameters, temporary variables, 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) {
  B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
```



Excecution stack

2.3.4 Address space state

Threads encapsulate concurrency; address spaces encapsulate protection -- keep a buggy program from trashing everything else on the system.

Address space state:

Contents of main memory

UNIX files

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 Pilot	VMS, Mach, OS/2 Windows NT, Solaris, 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 (Solaris, HPUX) -- many threads per address space, many address spaces
4. Embedded systems (Geoworks, VxWorks, 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.

Lecture 2 ended here