Concurrency

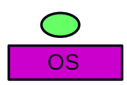
Arvind Krishnamurthy
Spring 2004

Today's lecture

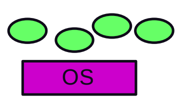
- Semester roughly divided into:
  - Threads, synchronization, scheduling
  - Virtual memory
  - File systems
  - Networking
- Next 2.5 weeks: threads and synchronization
- Today: thread basics
- Friday: C++ tutorial
- Wednesday: thread implementation

Process Environments

- Uniprogramming: 1 process at a time
  - "Cooperative timesharing": mostly PCs, vintage OSes
  - Easy for OS, hard for user (generally)
  - Does not deal with concurrency (by defining it away)
  - Violates isolation: Infinite loops



- Multiprogramming: > 1 process at a time
  - Time-sharing: CTSS, Multics, Unix, VMS, NT, ...



Why concurrency?

- Operating systems have two general functions:

  A. Standard services: Provide standard facilities that everyone needs
  Examples: standard libraries, file systems, windowing systems

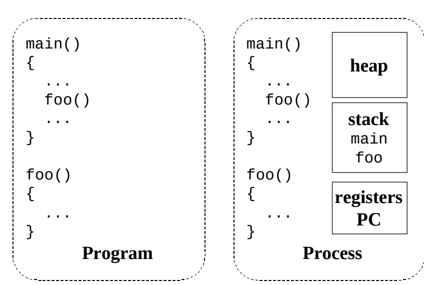
  B. Coordinator: allow several things to work together
  Examples: memory protection, virtual memory

- OS has to coordinate all the activity on a machine
  - Requests from multiple users, I/O interrupts
- Decompose hard problems into simpler ones
  - Instead of dealing with everything at once, deal with one at a time

Processes

- Process: OS abstraction to represent what is needed to run a single program
- Formally, a process is a *sequential stream* of execution in its own *address space*
- Two parts to a process:
  - Sequential execution: no concurrency inside a process – everything happens sequentially
  - Process state: everything that is necessary to run the process
- Question: What is typically part of the process state?
  - In other words, suppose you want to "migrate" a process what would you have to transport to the new machine?

Program vs. process



Program vs. process (cont'd)

- Process > program
  - Program is just part of process state
  - Example: many users can run the same program (creating different processes)
- Process < program
  - A program can invoke more than one process
  - Example: cc starts up cpp, cc1, cc2, as, ld (each are programs themselves)

Process creation in Unix

- Each process has its own state
- Even if two processes are created from the same executable, they will still have different state associated with them
- When a process clones itself using "fork", a separate copy is created

```
int  x = 1;
int  pid;
main() {
   pid = fork();
   x = x + 1;        <------  Updates made on
}                             different variables
```

Process creation in Unix (contd.)

- How to make processes:
  - fork clones a process
  - exec overlays the current process

```
if((pid = fork()) == 0) {
        /* child process */
        exec("foo");            /* exec does not return */
else
        /* parent */
        wait(pid);              /* wait for child to finish */
```

- Question: Is this a good interface?

Announcements

- Zheng Ma is the TA for the class
  - Email: zheng.ma@yale.edu
  - Please put "cs422" in subject line
- Assignment 0 is on the class website
  - Each person needs to this separately
  - Due: Jan 26th

Processes, Threads, Address Spaces

- Process: Thread(s) + Address space
- Thread: sequential execution stream within a process
  - Provides concurrency
- Address space: state needed to run a program
  - "Execution context" or "Container" for execution stream
  - Literally, all the memory addresses that can be touched by the program
  - Provides illusion of program having its own machine
- Multithreading: single program composed of a number of different concurrent activities

Thread creation in Nachos

- Thread creation in Nachos is very explicit

```
void SimpleThread (int which) {
  int num;
  for (num=0; num<5; num++)
     printf("*** thread %d looped %d times\n", which, num);
}

void ThreadTest() {
  Thread *t = new Thread("forked thread");
  t->Fork(SimpleThread, 1);
  SimpleThread(0);
}
```

- What is the output? Depends on the implementation…

Explicit Yields

```
void SimpleThread (int which) {
  int num;
  for (num=0; num<5; num++) {
    printf("*** thread %d looped %d times\n", which, num);
    currentThread->Yield();
  }
}


void ThreadTest() {
  Thread *t = new Thread("forked thread");
  t->Fork(SimpleThread, 1);
  SimpleThread(0);
}
```

Thread State

- Can be classified into two types:
  - Private
  - Shared
- Shared state
  - Contents of memory (global variables, heap)
  - File system
- Private state
  - Program counter
  - Registers
  - Stack

Threads Share Memory

```
int done[2] = {0, 0};
void SimpleThread (int which) {
  int num;
  for (num=0; num<5; num++)
    printf("*** thread %d looped %d times\n", which, num);
  done[which] = 1;
}

void ThreadTest() {
  Thread *t = new Thread("forked thread");
  t->Fork(SimpleThread, 1);
  SimpleThread(0);
  while (!done[1]);
  // Perform work that incorporates results from two threads
}
```

Classifying State

```
int   x;              <------  global variable

void foo() {
   int   y;           <------  stack variable
   x = 1;
   y = 1;
}

main() {
   int   *p;
   p = (int *)malloc(sizeof(int));
   *p = 1;            <------  heap access
}
```

Address Space Properties

- Addresses of global variables defined at link-time
- Addresses of heap variables defined at malloc-time

```
int  x;

main() {
  …
  int *p = malloc(sizeof(int));
  printf("%x %x", &x, p);
}
//  &x will never change across executions, p might
```

Stack Variables

Addresses of stack variables defined at "call-time"

```
void foo() {
   int x;
   printf("%x", &x);
}
void bar() {
   int y;
   foo();
}
main() {
   foo();
   bar();
}       // different addresses will get printed
```

Thread State

- Shared state:
  - Global variables
  - Heap variables
- Private state:
  - Stack variables
  - Registers
- When thread switch occurs, OS needs to save and restore private state

Thread State

```
int done[2] = {0, 0};
void SimpleThread (int which) {
  int num;
  for (num=0; num<5; num++)
    printf("*** thread %d looped %d times\n", which, num);
  done[which] = 1;
  printf("%x %x\n", done, &num);   // diff. Addresses for num
}
void ThreadTest() {
  Thread *t = new Thread("forked thread");
  t->Fork(SimpleThread, 1);
  SimpleThread(0);
  while (!done[1]);
  // Perform work that incorporates results from two threads
}
```

Wrap Up

- Threads encapsulate concurrency
- Address spaces encapsulate protection
- Thread is active, address space is passive
- Examples:
  - MS/DOS --- one thread, one address space
  - Old Unix --- one thread per address space, many address spaces
  - New Unix (Linux, Solaris), Win NT, Mach --- many threads per address space, many address spaces
  - Embedded systems (VxWorks, JavaOS) --- many threads, one address space (either no need for protection or it is achieved via other means)