Processes & Threads

Today

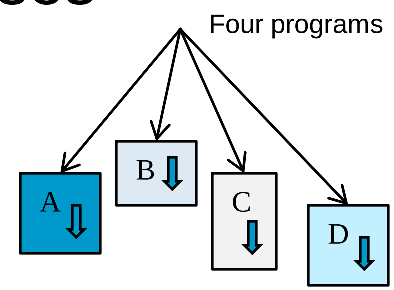
- Process concept
- Process model
- Implementing processes
- Multiprocessing once again

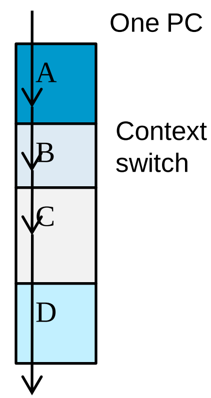
Next Time

- More of the same ☺

The process model

- Most computers can do more than one thing at a time
  - Hard to keep track of multiple tasks
- How do you call each of them?
  - Process - program in execution
  - a.k.a. job, task
- CPU switches back & forth among processes
  - Pseudo-parallelism
- Multiprogramming on a single CPU
  - At any instant of time one CPU means one executing task, but over time …
  - Every processes as if having its own CPU
- Process rate of execution – not reproducible





What’s in a process

- A process consists of (at least)…
  - An address space
  - The code of the running program
  - The data for the running program
  - Execution stack and stack pointer
  - Program counter
  - A set of general purpose registers
  - A set of OS resources including open files, network connections, …
  - Other process metadata (e.g. signal handlers)

Process creation

- Principal events that cause process creation
  – System initialization
  – Execution of a process creation system
  – User request to create a new process
  – Initiation of a batch job
- In all cases – a process creates another one
  – Running user process, system process or batch manager process
- Process hierarchy
  – UNIX calls this a "process group"
  – No hierarchies in Windows - all created equal (parent does get a handle to child, but this can be transferred)

Process creation

- Resource sharing
  - Parent and children share all resources, a subset or none
- Execution
  - Parent and children execute concurrently or parent waits
- Address space
  - Child duplicate of parent or one of its own from the start
- UNIX example
  - fork system call creates new process; a clone of parent
  - Both processes continue execution at the instruction after the fork
  - execve replaces process' memory space with new one

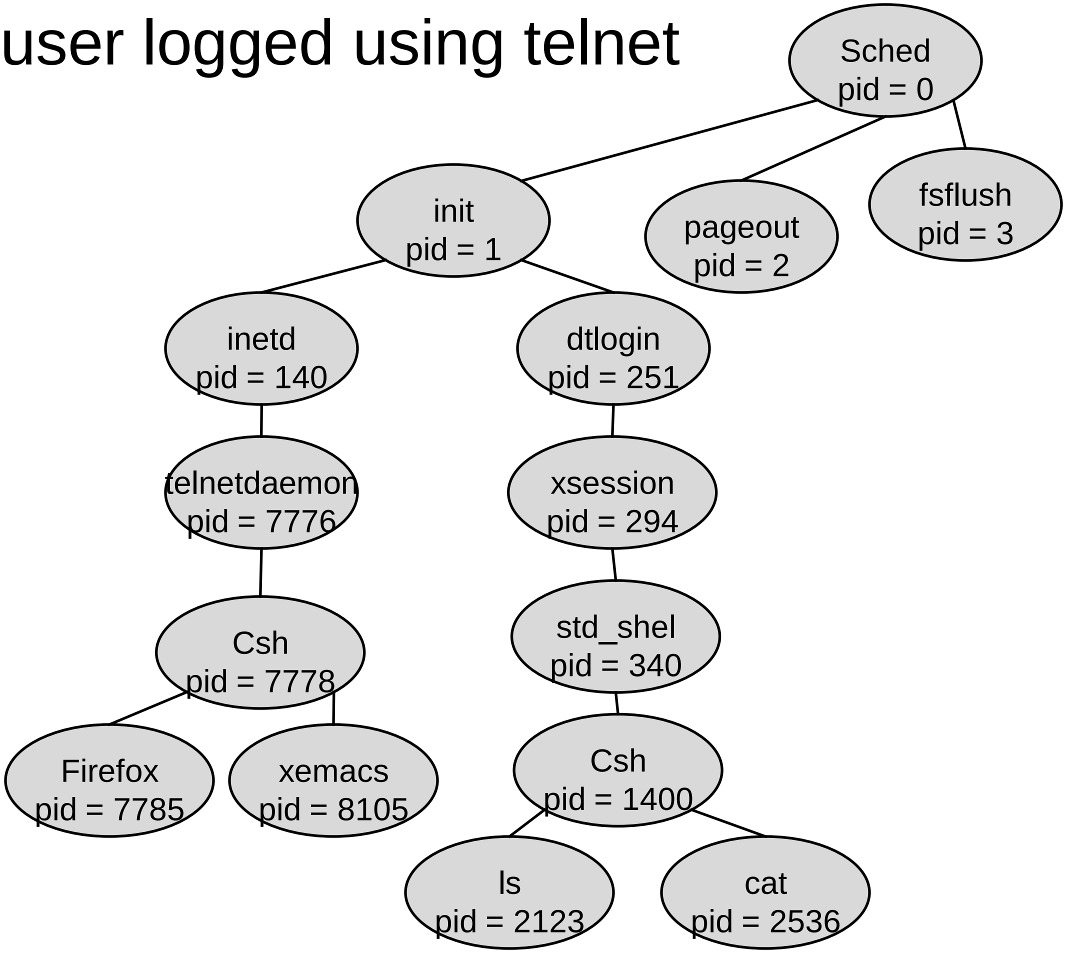
Why two steps?

Process identifiers

- Every process has a unique ID
- Since it's unique sometimes used to guarantee uniqueness of other identifiers (`tmpnam/tmpfile`)
- Special process IDs: 0 – swapper, 1 – init
- Creating process in Unix – fork
  - `pid_t fork(void);`
  - Call once, returns twice
  - Returns 0 in child, pid in parent, -1 on error
- Child is a copy of the parent
  - Expensive task!

Hierarchy of processes in Solaris

- `sched` is first process (`initcode` in xv6)
- Its children pageout, fsflush, init …
- `csh` (pid = 7778), user logged using telnet
- …



Process termination

Conditions which terminate processes

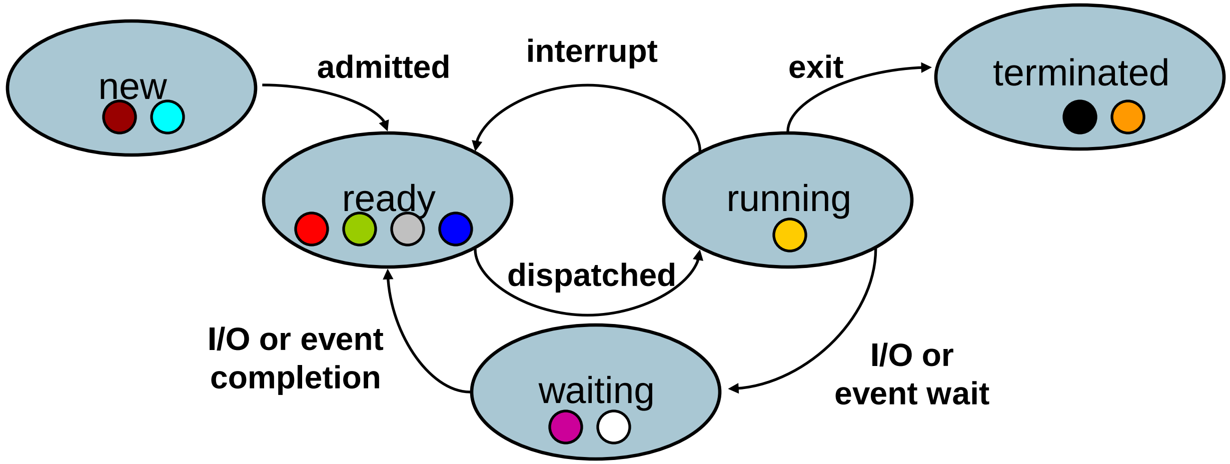
- Normal exit (voluntary)
  - the job is done
- Error exit (voluntary)
  - oops, missing file?
- Fatal error (involuntary)
  - Referencing non-existing memory perhaps?
- Killed by another process (involuntary)
  - “`kill -9`”

Unix – ways to terminate

- Normal – return from main, calling exit (or _exit)
- Abnormal – calling abort, terminated by a signal

Process states

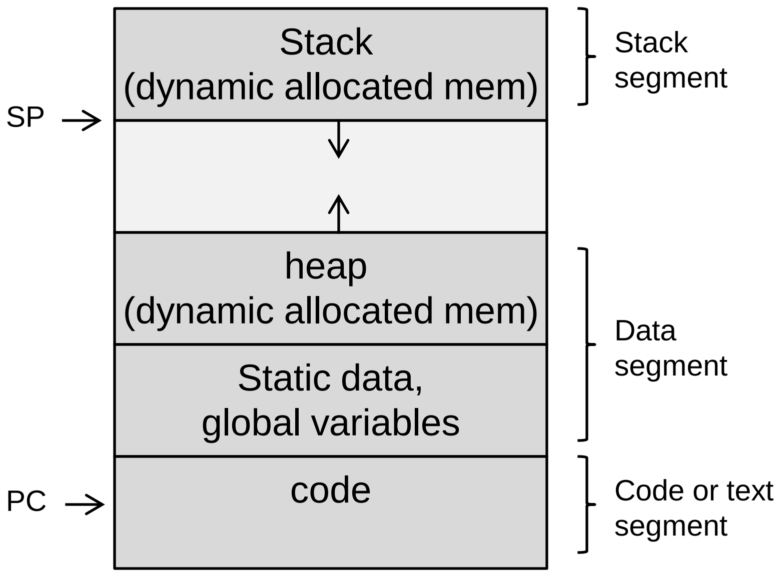
- Possible process states (in Unix run `ps`)
  - New – being created
  - Ready – waiting to get the processor
  - Running – being executed (*how many at once?*)
  - Waiting – waiting for some event to occur
  - Terminated – finished executing
- Transitions between states



Which state is a process in most of the time?

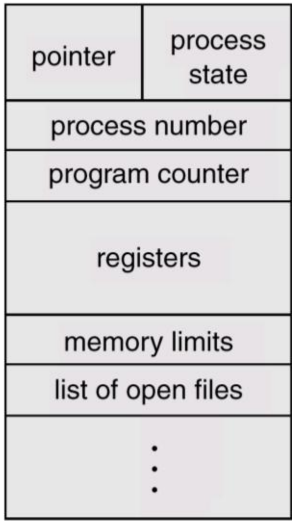
Implementing processes

- Process
  - A program in execution (i.e. more than code, text section)
  - Program: passive; process: active
- Current activity
  - Program counter & content of processor's registers
  - Stack – temporary data including function parameters, return address, …
  - Data section – global variables
  - Heap – dynamically allocated memory



Implementing processes

- OS maintains a process table of Process Control Blocks (PCB)
- PCB: information associated with each process
  - Process state: ready, waiting, …
  - Program counter: next instruction to execute
  - CPU registers
  - CPU scheduling information: e.g. priority
  - Memory-management information
  - Accounting information
  - I/O status information
  - …



Processes in xv6

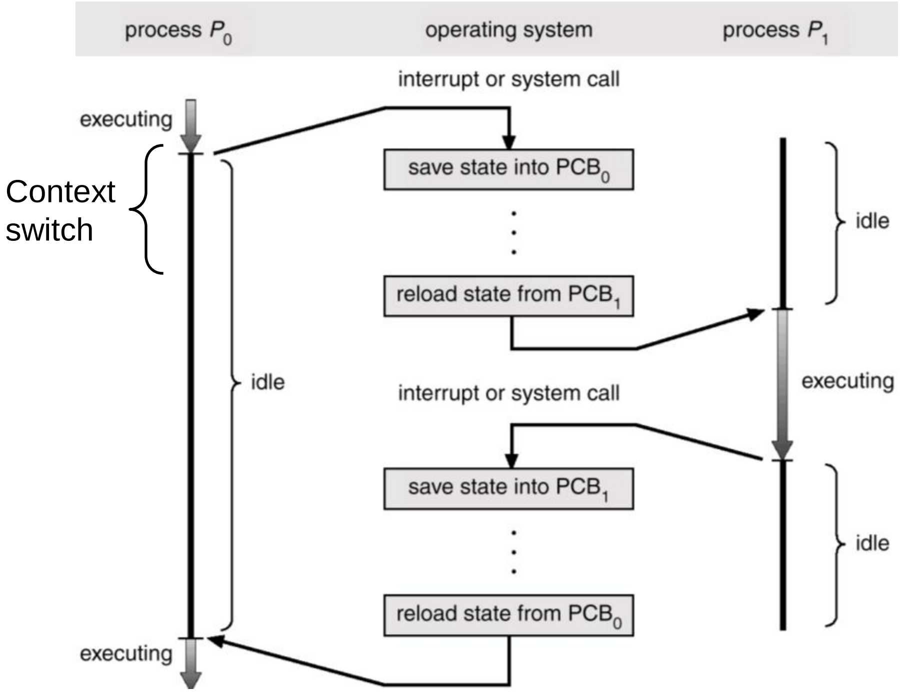
```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
   char *mem;                        // Start of process memory (kernel address)
   uint sz;                          // Size of process memory (bytes)
   char *kstack;                     // Bottom of kernel stack for this process
   enum procstate state;             // Process state
   volatile int pid;                 // Process ID
   struct proc *parent;              // Parent process
   struct trapframe *tf;             // Trap frame for current syscall
   struct context *context;          // Switch here to run process
   void *chan;                       // If non-zero, sleeping on chan
   int killed;                       // If non-zero, have been killed
   struct file *ofile[NOFILE];       // Open files
   struct inode *cwd;                // Current directory
   char name[16];                    // Process name (debugging)
};

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

Statically-size process table

EECS343/repos/xv6-rev4.pdf

Switch between processes

- When a process is running, its hardware state is loaded on a CPU
- When the process is transitioned to waiting, the OS saves the register values in the PCB
- The act of switching a CPU from one process to another – context switch
  - ~5 microseconds
- Choosing which process to run next – scheduling



State queues

- OS maintains a collection of queues that represent the state of processes in the system
  - Typically one queue for each state
  - PCBs are queued onto state queues according to current state of the associated process
  - As a process changes state, its PCB is unlinked from one queue, and linked onto another
- There may be many wait queues, one for each type of wait (devices, timer, message, …)

PCB and state queues

- PCB are data structures
  - Dynamically allocated inside OS memory
- When a process is created
  - OS allocates and initializes a PCB for it
  - OS places it on the correct queue
- As process computes
  - OS moves its PCB from queue to queue
- When process terminates
  - PCB may hang around for a while (exit code …)
  - Eventually OS deallocates its PCB
  - *Check out xv6/proc.c:exit()*

Process creation in UNIX

```
#include <stdio.h>
#include <sys/types.h>

int tglob = 6;

int main (int argc, char* argv[])
{
  int pid, var;

  var = 88;
  printf("write to stdout\n");
  fflush(stdout);
  printf("before fork\n");
  …
```

```
[fabianb@eleuthera tmp]$ ./creatone
a write to stdout
before fork
pid = 31848, tglob = 7, var = 89
pid = 31847, tglob = 6, var = 88
```

```
…
if ((pid = fork()) < 0){
    perror("fork failed");
    return 1;
  } else {
    if (pid == 0){
      tglob++;
      var++;
    } else  /* parent */
       sleep(2);
  }
  printf("pid = %d, tglob = %d, var
   = %d\n",
   getpid(), tglob, var);
  return 0;
} /* end main */
```

Process creation in UNIX

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
  pid_t childpid;
  pid_t mypid;

  mypid = getpid();
  childpid = fork();
  if (childpid == -1) {
    perror("Failed to fork\n");
    return 1;
 }

  if (childpid == 0) /* child code */
    printf("Child %ld, ID = %ld\n", (long) getpid(), (long) mypid);
  else /* parent code */
    printf("Parent %ld, ID = %ld\n", (long) getpid(), (long) mypid);
  return 0;
}
```

```
[fabianb@eleuthera tmp]$ ./badpid 4
Child 3948, ID = 3947
Parent 3947, ID = 3947
```

Process creation in UNIX

```
...

  if ((pid = fork()) < 0) {
    perror(“fork failed”);
    return 1;
  } else {
    if (pid == 0) {
      printf(“Child before exec … now the ls output\n”);
      execlp("/bin/ls", "ls", NULL);
    } else {
      wait(NULL); /* block parent until child terminates */
      printf("Child completed\n");
      return 0;
    }
  }
} /* end main */
```

```
[fabianb@eleuthera tmp]$ ./creattwo
Child before exec ... now the ls output
copy_shell       creatone.c~  p3id    skeleton
copy_shell.tar   creattwo     p3id.c  uwhich.tar
creatone         creattwo.c   p3id.c~
creatone.c       creattwo.c~
Child completed
```

Faster creation

- The semantics of fork() says that the child's address space is a copy of the parent's
- Expensive (i.e. slow) implementation
  - Allocate physical memory for the new address space
  - Copy one into the other
  - Set up child's page tables to map to new address space
- To make it faster …

Faster creation …

- To make it faster
  - vfork() - change problem definition a bit
    - "child address space is a copy of the parent's" -> "child address space *is* the parent's"
    - Promise the child won't modify the address space before doing an exec
  - COW – copy on write
    - Retains the semantics
    - Copy only what's necessary
      - Initialize page tables to the same mappings as parent's and set both parents and child page tables to read-only
      - If anybody tries to write – page fault
        - » Allocate new physical page for child
        - » Copy content, mark entries as writable, restart process

UNIX shells

```
int
main(int argc, char **argv)
{
  while (1) {
    printf("% ");
    char *cmd = get_next_cmd();
    int pid = fork();
    if (pid == 0) {
       exec(cmd);
       panic("exec failed!");
     } else {
       wait(pid);
     }
  }
}
```

Xv6/sh.c has a bit but not much more!

Summary

- Today
  - The process abstraction
  - Its implementation
    - How they are represented
    - How the CPU is scheduled across processes
    - …
  - Processes in Unix
  - Perhaps the most important part of the class
- Coming up
  - Threads & synchronization