Processes & Threads

Today

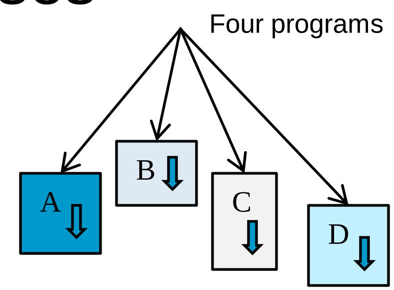
- Process concept
- Process model
- Implementing processes
- Multiprocessing once again

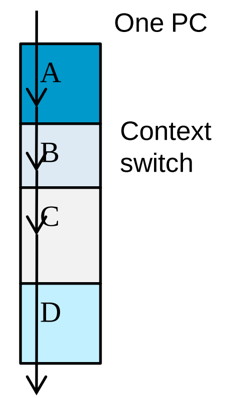
Next Time

- 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





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

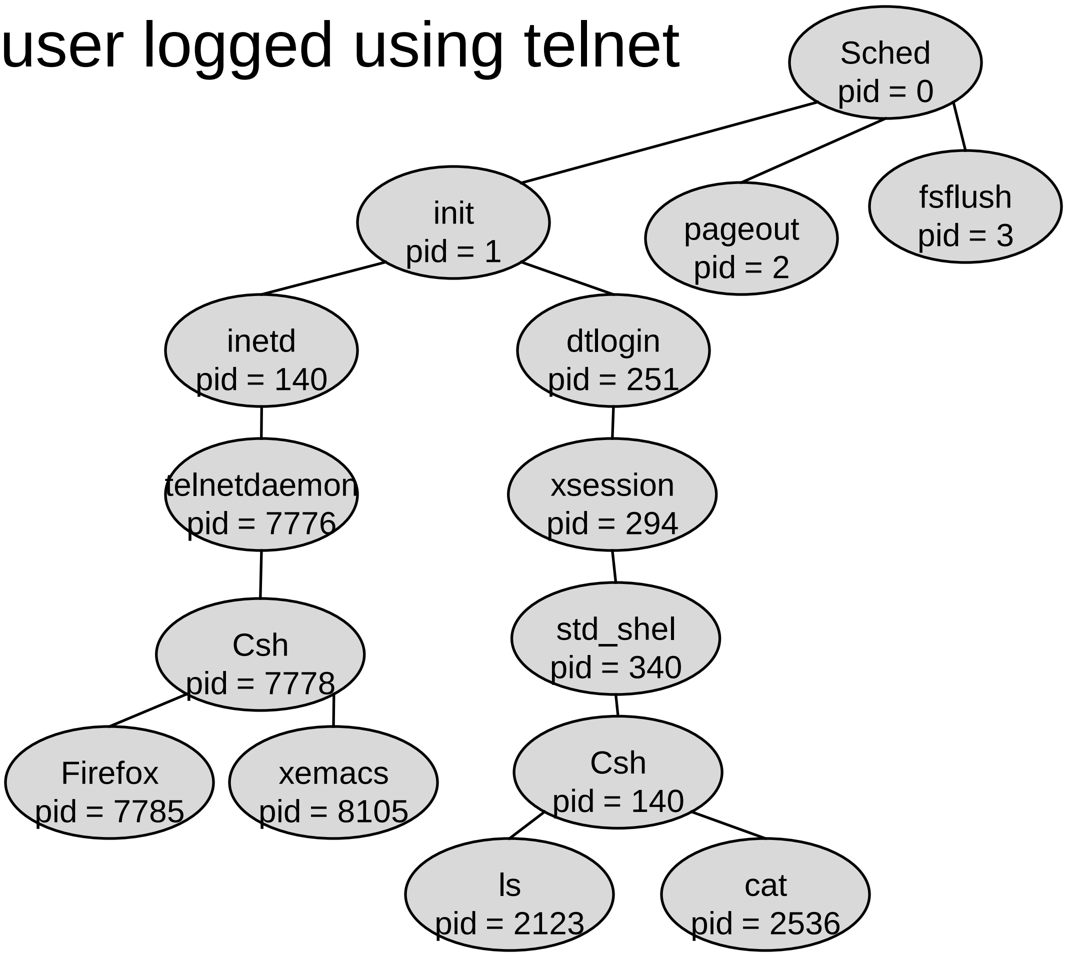
Why two steps?

Process identifiers

- Every process has a unique ID
- Since it's unique sometimes used to guarantee uniqueness of other identifiers (tmpnam)
- 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
  - Another option - COW (copy-on-write)?

Hierarchy of processes in Solaris

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



Process termination

Conditions which terminate processes

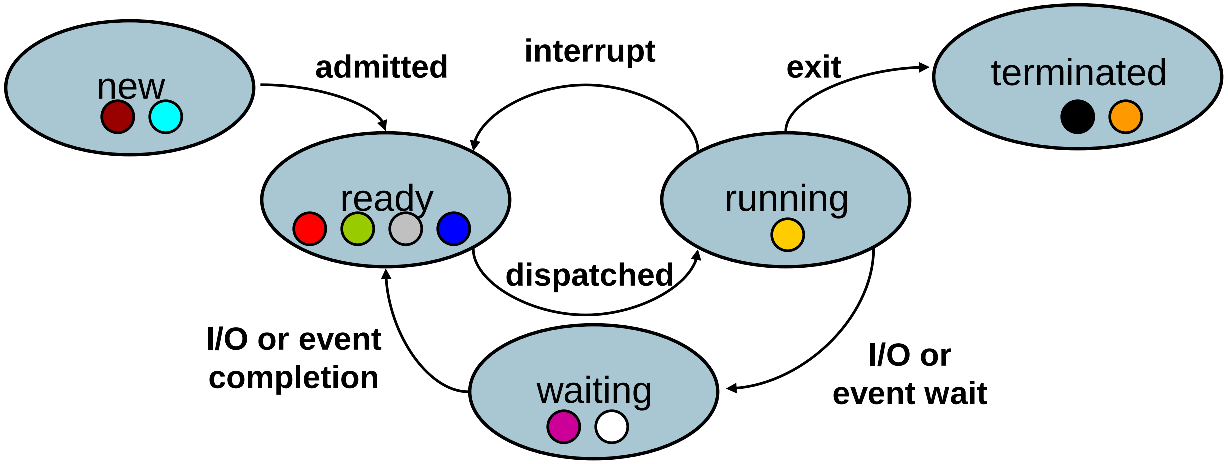
- Normal exit (voluntary)
  - the job is done
- Error exit (voluntary)
  - oops, missing file?
- Fatal error (involuntary)
  - Referencing non-existing memory perhaps?
- Killed by another process (involuntary)
  - "`kill -9`"

Unix – ways to terminate

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

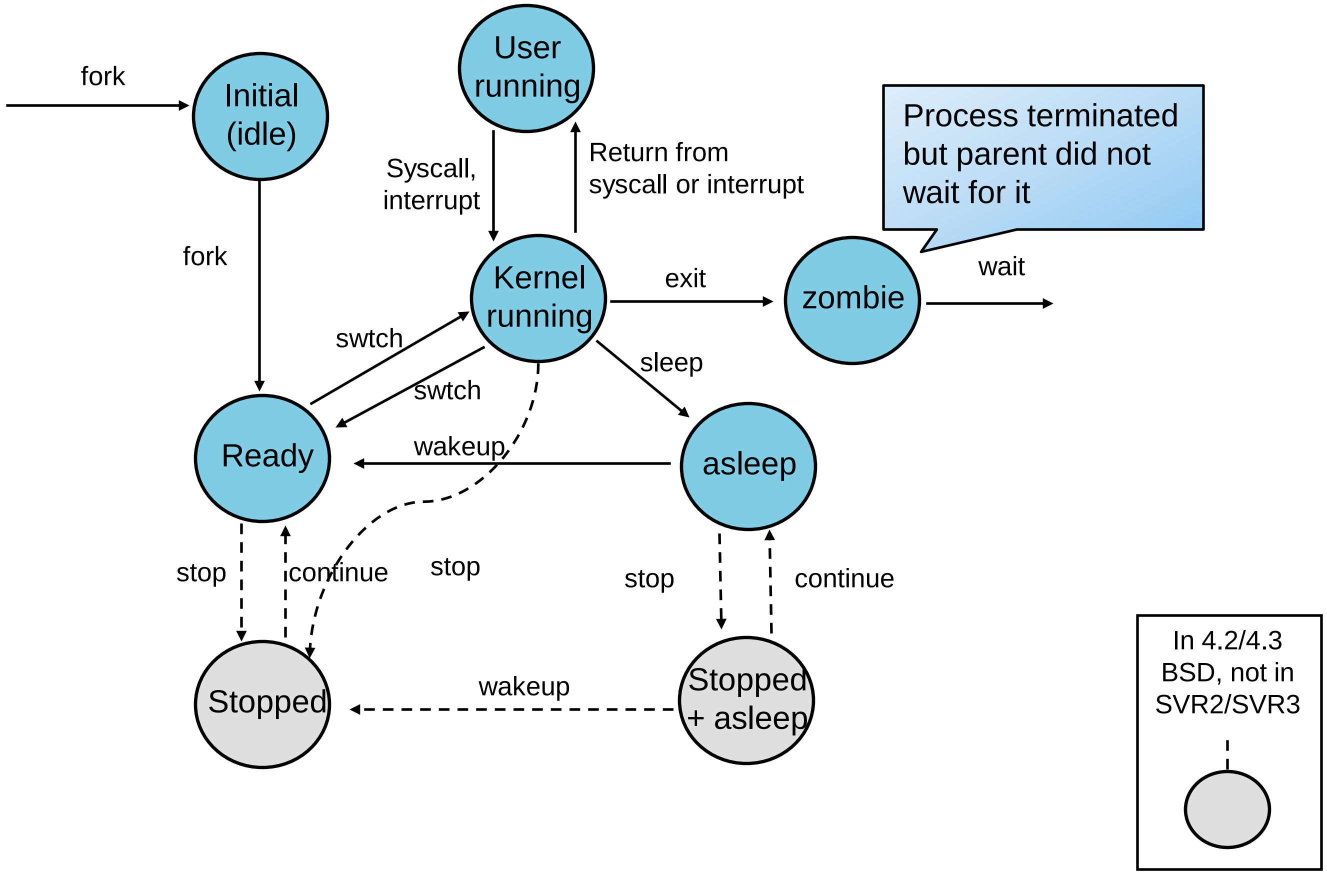
Process states

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



Which state is a process in most of the time?

Process states in Unix



From U. Vahalia, UNIX Internal, Prentice-Hall, 1996

Execution mode and context

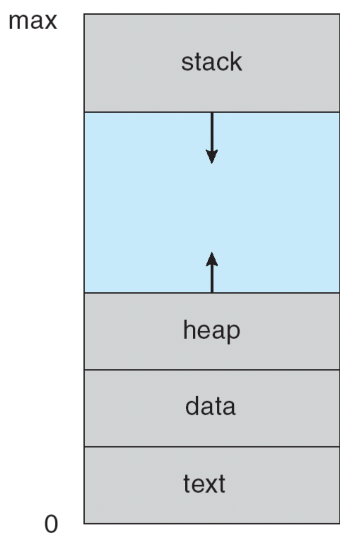
Execution context	Mode of execution: User mode	Mode of execution: Kernel mode
Process context	Applications (user) code (access process space only)	System calls, exceptions (access process & system space)
System context	*Not allowed*	Interrupts, system tasks (access system space only)

Kernel acts on behalf of the process

Kernel performs some system-wide task

Implementing processes

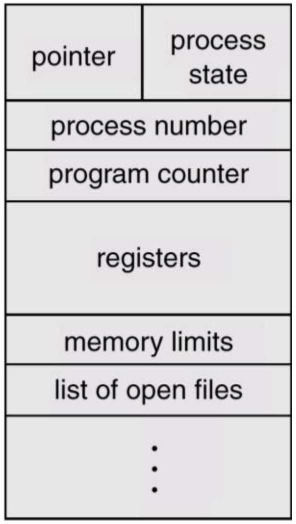
- Process
  – A program in execution (i.e. more than code, text section)
  – Program: passive; process: active
- Current activity
  – Program counter & content of processor's registers
  – Stack – temporary data including function parameters, return address, …
  – Data section – global variables
  –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

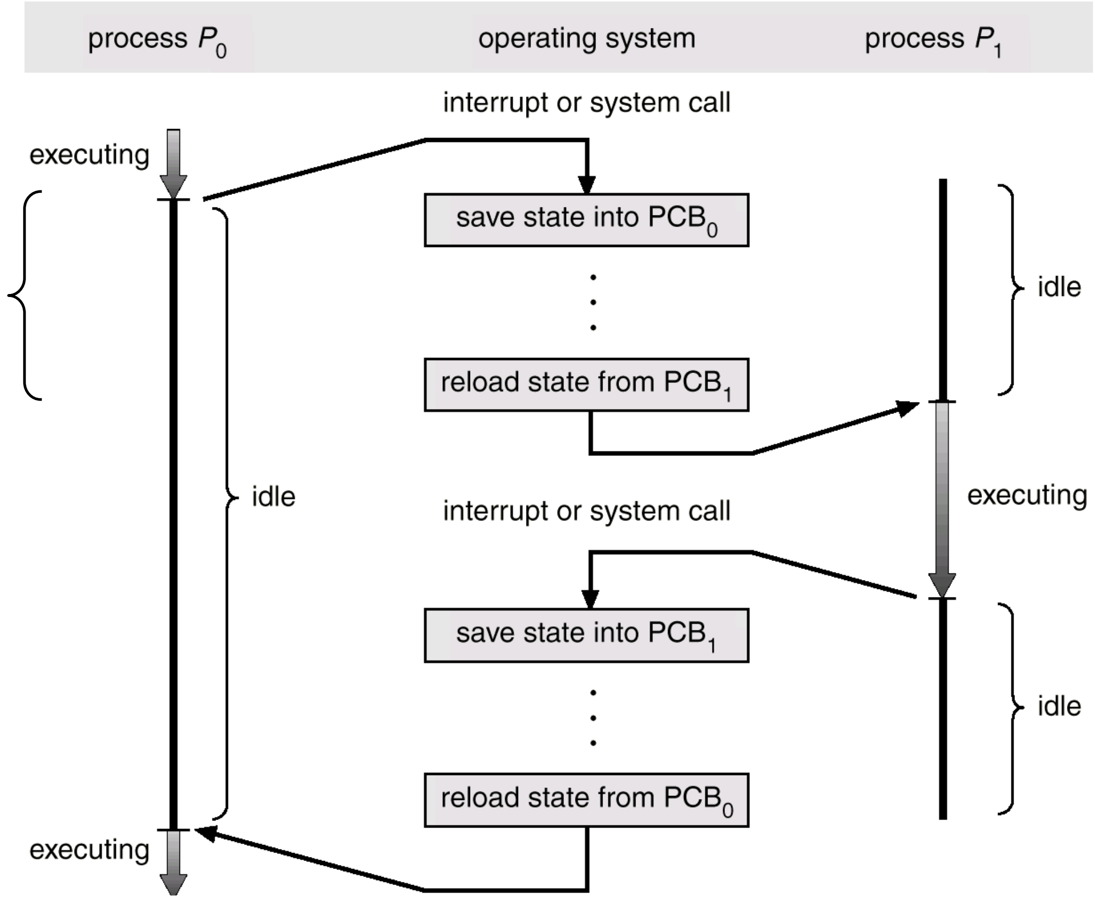
http://minnie.tuhs.org/UnixTree/V6/usr/sys/proc.h.html



Switch between processes

- Save current process' state before
- Restore the state of a different one

Context switch



Handling interrupts - again

What gets done when an interrupt occurs

1. HW stacks PC, etc
2. HW loads new PC from interrupt vector
3. Assembly lang. procedure saves registers
4. Assembly lang. procedure sets up new stack
5. C interrupt service runs
6. Scheduler decides which process to run next
7. C procedure returns to assembly code
8. Assembly code starts up new current process

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, …)

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");
  …
```

```
…
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 */
```

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

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
```

Summary

- Today
  - The process abstraction
  - Its implementation
  - Processes in Unix
- Next time
  - Threads