Processes

Today

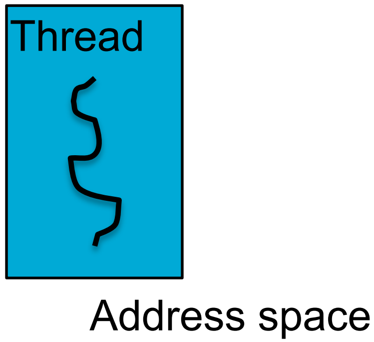
- Process concept
- Process model
- Implementing processes
- Multiprocessing once again

Next Time

- Scheduling processes

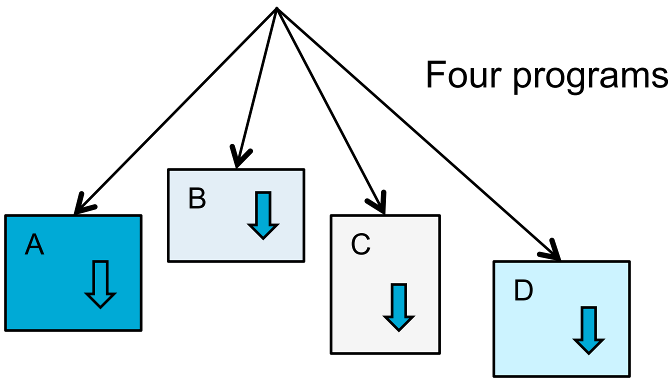
The process model

- 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 – the OS's abstraction for execution
  - A program in execution a.k.a. job, task
- Simplest (classic) case – a sequential process
  - An address space – abstraction of memory
  - A single thread of execution – abstraction of CPU

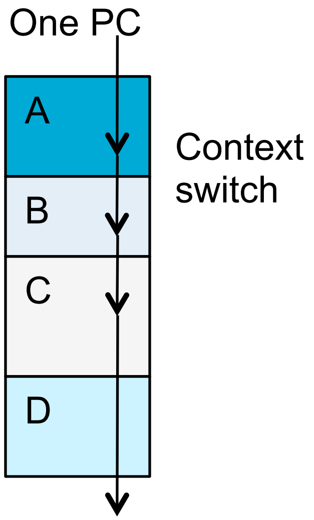


The process model

- Conceptually, every process has its own CPU
  - OS creates the illusion by virtualizing the CPU



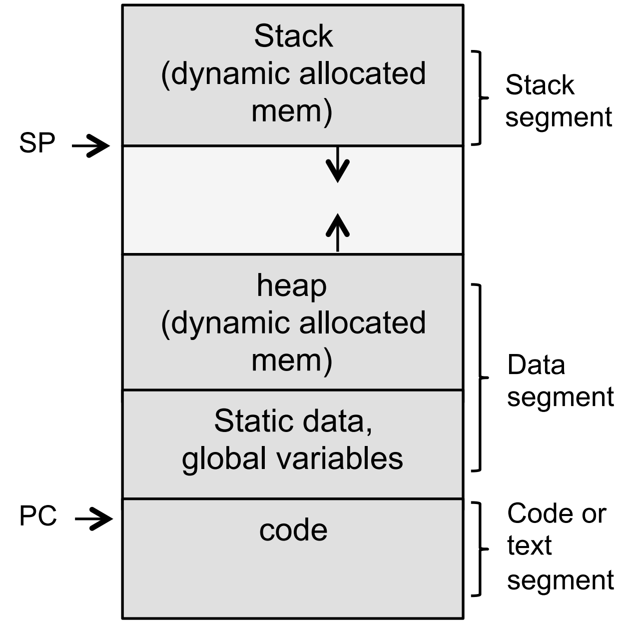
- In reality, CPU switches back & forth among processes
  - Pseudo-parallelism



- Multiprogramming on a single CPU
  - At any instant of time one CPU means one executing task, but over time …
- Process rate of execution – not reproducible

What's in a process

- A process consists of (at least)…
  - An address space
    - Running program code
    - Its' data
  - A thread state
    - Execution stack and stack pointer
    - Program counter
    - General purpose registers
  - A set of OS resources
    - Including open files, network connections, …
  - Other process metadata (e.g. signal handlers)



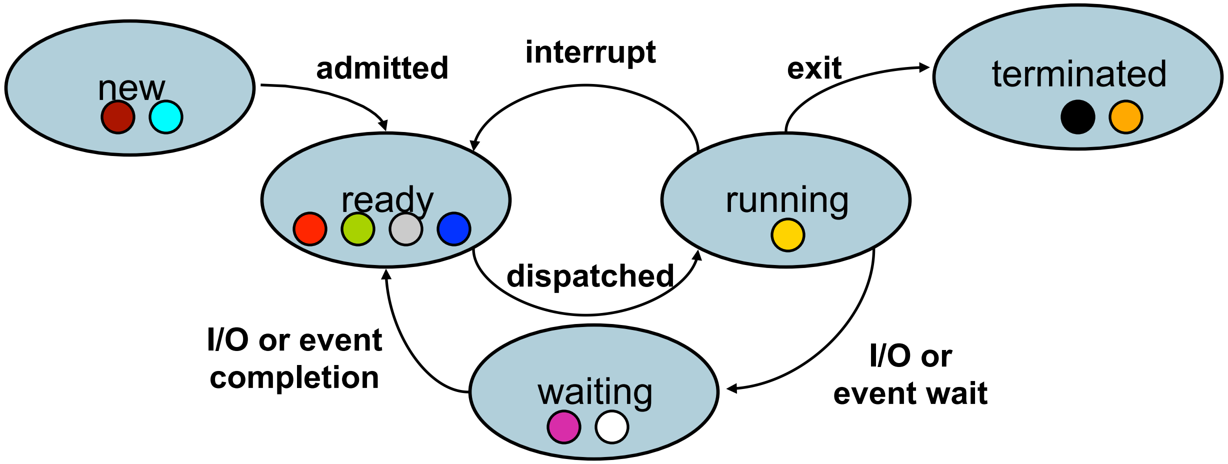
- i.e. all you need to run or restart a program if interrupted

Process identifiers

- Every process has a unique ID
  - The PID namespace is global to the system
  - Operations that create processes return a PID (e.g., fork)
  - Operations on processes take a PID as argument (e.g., kill)
- Creating process in Unix – fork
  - `pid_t fork(void);`
  - Call once, returns twice
  - Returns 0 in child, pid in parent, -1 on error
- Special process IDs: 0 – swapper, 1 – init
- Since it's unique sometimes used to guarantee uniqueness of other identifiers (`tmpnam/tmpfile`)

Process execution states

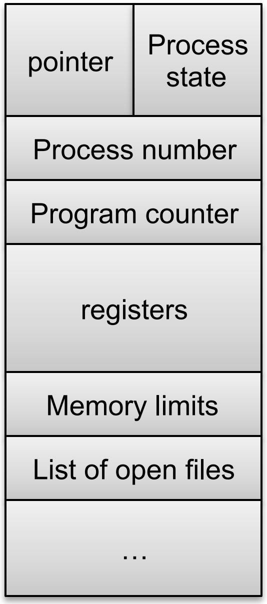
- Possible process states (in Unix run `ps`)
  - New – being created
  - Ready – waiting to get the processor
  - Running – being executed (*how many at once?*)
  - Waiting – waiting for some event to occur
  - Terminated – finished executing
- Transitions between states



Which state is a process in most of the time?

Implementing processes

- OS uses a data struct to keep track of process state
  - The Process Control Block
- PCB: information associated with each process
  - Process state: ready, waiting, …
  - Program counter
  - CPU registers
  - CPU scheduling information: e.g. priority
  - Memory-management information
  - Accounting information
  - I/O status information
  - …



- In Linux: defined in `task_struct` (`include/linux/sched.h`)

Processes in xv6 (~x86-port of Unix v6)

```
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)
};

…
```

Processes in xv6 (~x86-port of Unix v6)

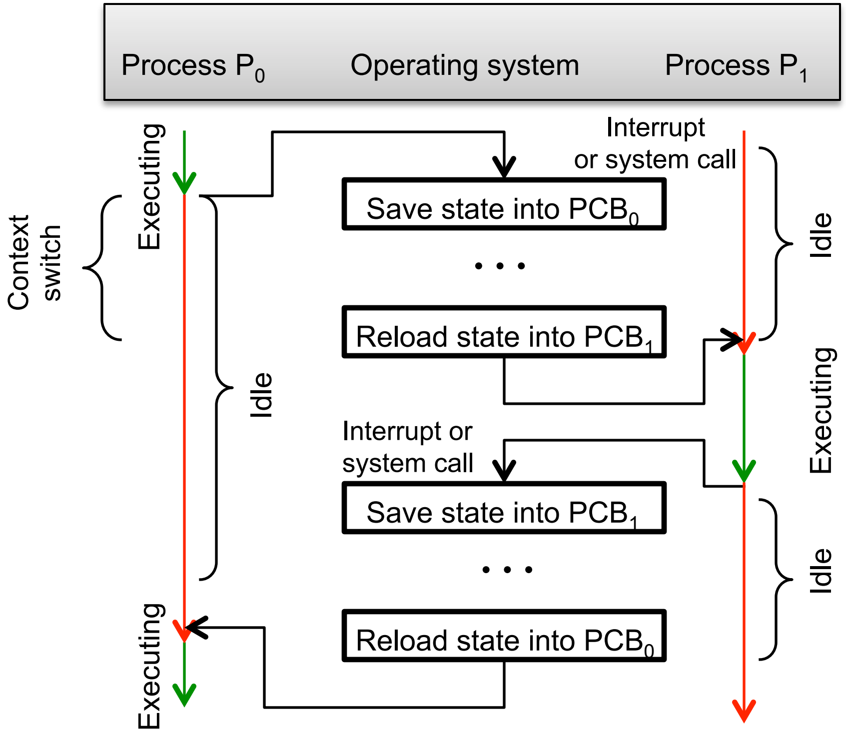
```
// The registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;              // program counter
  int esp;              // stack pointer
  int ebx;              // Base index
  int ecx;              // Counter
  int edx;              // Extend the precision of the accumulator
  int esi;              // Source index for string ops
  int edi;              // Destination index for string ops
  int ebp;              // Stack base pointer to current stack frame
}

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

Statically-size process table …

PCBs and CPU state

- When a process is running, hardware state is loaded on CPU and registers
- When process is waiting, state is saved in the PCB
- Switching a CPU between process: context switch
  – ~5 microseconds in 1996, now is sub-microsecs



- Choosing which process to run next – scheduling (Next lectures!)

Context switching in xv6

```
# Context switch
# void swtch(struct context **old, struct context *new);
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
   movl 4(%esp), %eax
   movl 8(%esp), %edx

   # Save old callee-save registers
   pushl %ebp
   pushl %ebx
   pushl %esi
   pushl %edi

   # Switch stacks
   movl %esp, (%eax)
   movl %edx, %esp

   # Load new callee-save registers
   popl %edi
   popl %esi
   popl %ebx
   popl %ebp
   ret
```

Loads arguments off the stack into %eax and %edx before changing stack pointer

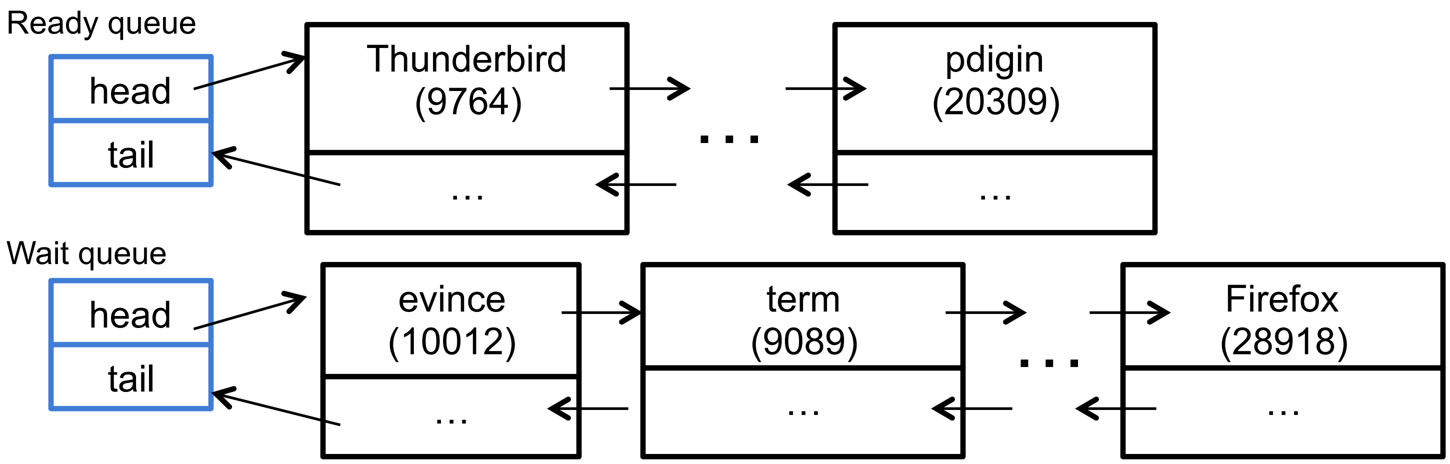
Pushes register state creating a context structure on the current stack;
%esp is save implicitly to *old;
%eip was saved by `call` instruction that invoked swtch and is above %ebp

Switch stacks

New stack has same format, so just undo;
`ret` has the %eip at the top

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/move between state queues according to current/new state of the associated process



- 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 frees its PCB

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)

Chicken and egg – What creates the first process and when?

Process creation

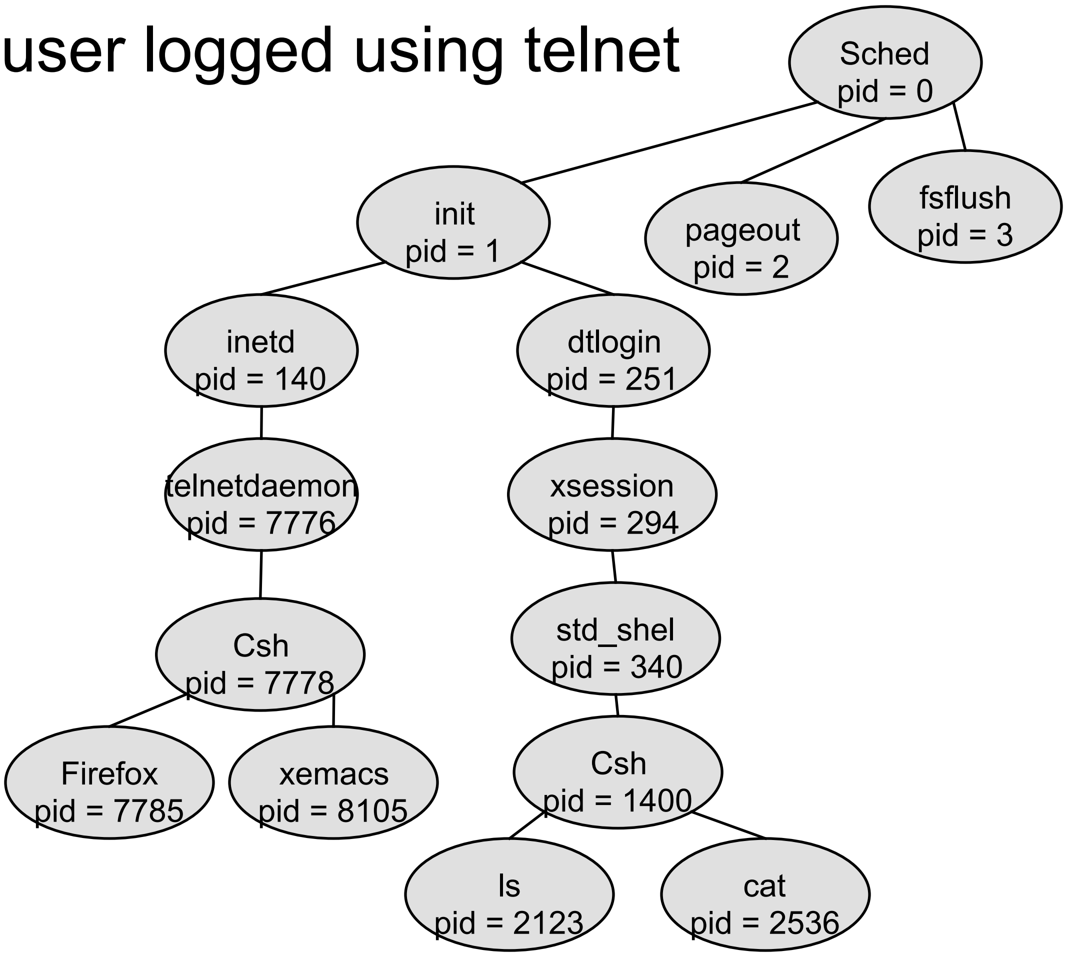
- Resource sharing
  - Parent and children share all resources, a subset or none
- Execution
  - Parent and children execute concurrently or parent waits
- Address space
  - Child duplicate of parent or one of its own from the start
- Unix example
  - `fork()` system call creates new process; a clone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

Why two steps?

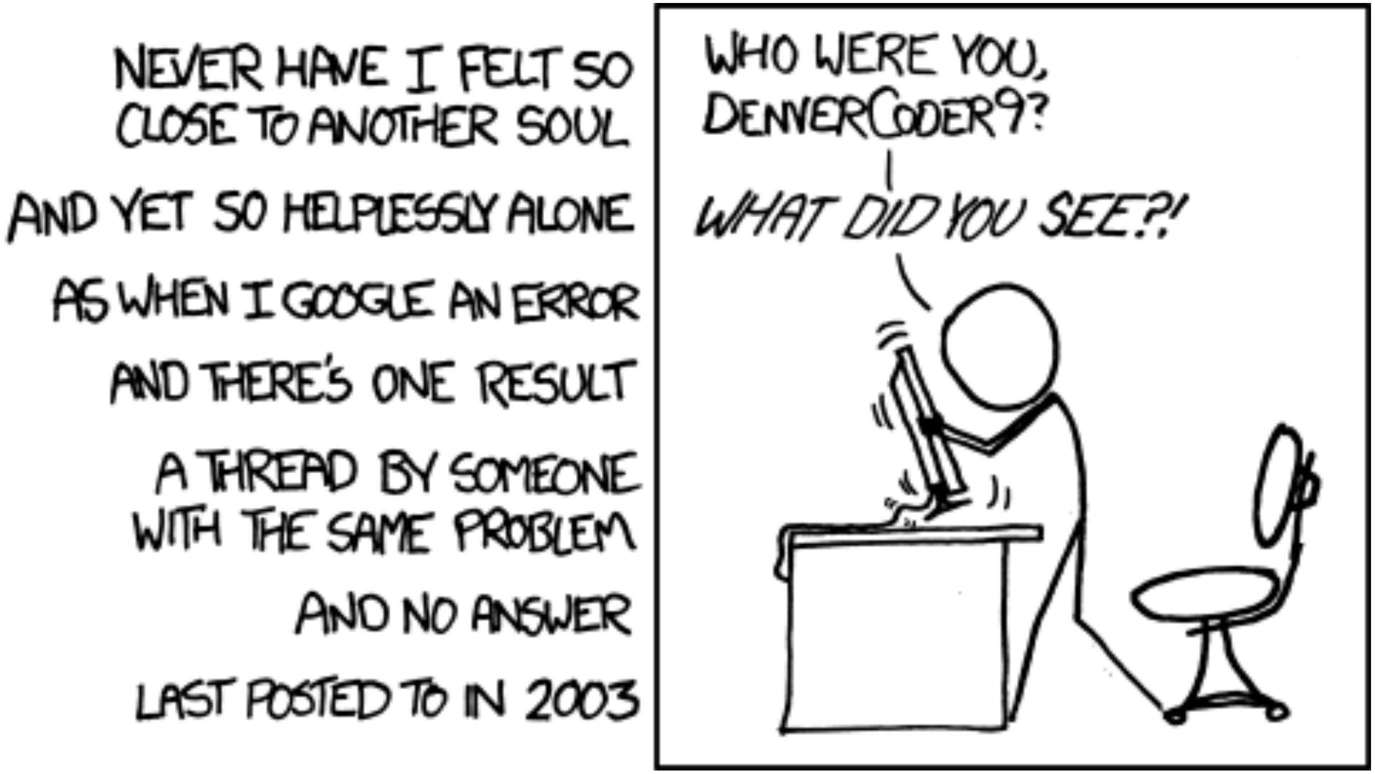
Can you think of an everyday example where fork is enough?

Hierarchy of processes in Solaris

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



And now a short break …



Process creation in UNIX

- Processes are created by existing processes
- UNIX creation through `fork()`
  - Creates and initializes a new PCB
  - Creates a new address space and initializes it with content of parent's
  - Initializes kernel resources with those of the parent
  - Places PCB in ready queue
- the `fork()` call once, returns twice
  - Once into the parent, and once into the child
    - Returns child's PID to the parent
    - And 0 to the child
- `fork()` ~ clone me

Process creation in UNIX

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

int main (int argc, char* argv[])
{
  int pid; int ppid = getpid();

  if ((pid = fork()) < 0){
    perror("fork failed");
    return 1;
  } else {
     if (pid == 0){    /* Return 0 to the child */
       printf(”I am %d the child of %d\n”, getpid(), ppid);
       return 0;
  } else {              /* And the child PID to the parent */
      printf(”I am %d, the parent of %d\n”, ppid, pid);
      return 0;
    }
  }
}
```

Where does the newly created process start?

Testing `fork()` - output

```
[fabianb@eleuthera tmp]$ gcc -o createone createone.c
[fabianb@eleuthera tmp]$ ./createone
I am 6647, the parent of 6648
I am 6648 the child of 6647
```

Process creation in UNIX

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

int main (void)
{
  pid_t childpid, mypid;

  mypid = getpid();
  childpid = fork();
  if (childpid == -1) {
    perror("Failed to fork\n");
    return 1;
 }
  if (childpid == 0) /* child */
    printf("Child %d, ID = %d\n", getpid(), mypid);
  else /* parent */
    printf("Parent %d, ID = %d\n", getpid(), mypid);
  return 0;
}
```

Both IDs should be the same

The dangers with sharing …

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

What?!?

```
  ...
  mypid = getpid();
  childpid = fork();
  if (childpid == -1) {
    perror("Failed to fork\n");
    return 1;
 }
  if (childpid == 0) /* child */
    printf("Child %d, ID = %d\n", getpid(), mypid);
  else /* parent */
    printf("Parent %d, ID = %d\n", getpid(), mypid);
  return 0;
}
```

Process creation in UNIX + `exec()`

- Beyond cloning – first fork, then exec
- `int execv(char *prog, char *argv[])`
  - (a family of functions, front-ends for `execve`)
  - Stops current process
  - Loads prog into the address space (overwriting what's there)
  - Initializes hardware content, args for new program
  - Places PCB onto ready queue
- To run a new program, then
  - `fork` to create a child
  - Child does an `exec`
  - Parent can wait for child to complete or not

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;
    }
  }
}
```

fork() + exec() – output

```
[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 – version 1

- Vfork() – oldest approach, redefine the problem
  - “child address space is a copy of the parent’s” ➔ “child address space *is* the parent’s”
  - Parent suspended until child exits or calls `execve`
  - Child promises not to modify the address space before that
  - Not enforced, use at your own peril
  - Saves the effort of duplicating parent’s address space when child is going to exec anyway
  - Uncommon today

Faster creation – version 2

- Keep old semantic, but implement it differently
  - Copy only what you need, on demand
- COW – copy on write
  - Create new address space
  - Initialize page tables to the same mappings as parent's and set both parents and child page tables to read-only
  - If either parent or child tries to write – page fault
  - When a page fault occurs
    - Allocate new physical page for child
    - Copy content
    - Mark entries as writable
    - Restart process
  - *Page are copied only as needed*

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);
     }
  }
}
```

UNIX shells

```
...
// Read and run input commands.
while(getcmd(buf, sizeof(buf)) >= 0){
  if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
    // Clumsy but will have to do for now.
    // Chdir has no effect on the parent if run in the child.
  buf[strlen(buf)-1] = 0; // chop \n
  if(chdir(buf+3) < 0)
     printf(2, "cannot cd %s\n", buf+3);
  continue;
  }
  if(fork1() == 0)
    runcmd(parsecmd(buf));
  wait();
}
...
```

Fork and Linux' clone

- Clone is replacing fork (and vfork)

```
#include <sched.h>
int clone (int (*fn) (void *), void *child_stack, int
flags, void *arg);
```

- Starting at a different point, with a given stack, and after inheriting something from the parent
- Not typically called directly

`exec` and company

- exec is not a system call

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
          ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
           char *const envp[]);
```

- execve is the only “exec-like” system call
  - The rest are front-ends to it
  - execve knows whether you have done a fork or a vfork by a flag in the PCB

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
  - Scheduling …