CSE 451: Operating Systems
Winter 2001

Lecture 4
Processes

Steve Gribble
gribble@cs.washington.edu
323B Sieg Hall

Today's agenda

- Administrivia
  - ...
- Processes
  - units of exection
  - 'living, breathing embodiment of a program'

Process Management

- This lecture begins a series of topics on processes, threads, and synchronization
  - this is perhaps the most important part of the class
  - there definitely will be several questions on these topics on the midterms and exams
- Today: processes and process management
  - what are the OS units of execution?
  - how are they represented inside the OS?
  - how is the CPU scheduled across processes?
  - what are the possible execution states of a process?
    - and how does the system move between them?

The Process

- The process is the OS's abstraction for execution
  - the unit of execution
  - the unit of scheduling
  - the dynamic (active) execution context
    - compared with program: static, just a bunch of bytes
- Process is often called a job, task, or sequential process
  - a sequential process is a program in execution
    - defines the instruction-at-a-time execution of a program
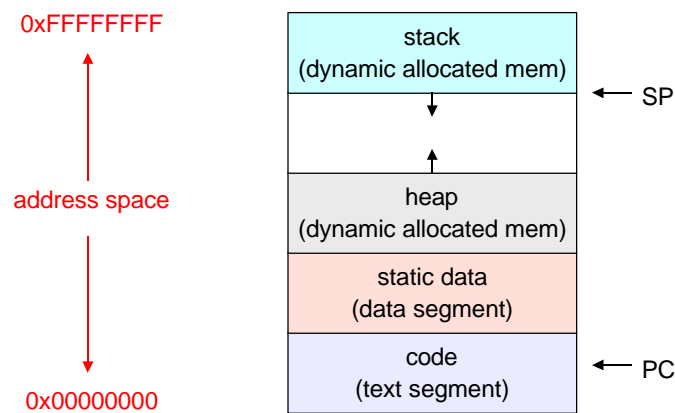
What's in a Process?

- A process consists of (at least):
  - an address space
  - the code for the running program
  - the data for the running program
  - an execution stack
    - traces state of procedure calls made
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values
  - a set of OS resources
    - open files, network connections, sound channels, …
- The process is a container for all of this state
  - a process is named by a process ID (PID)
    - just an integer (actually, typically a short)

A process's address space

0xFFFFFFFF

address space

0x00000000

| stack (dynamic allocated mem) | ← SP |
heap (dynamic allocated mem)	
static data (data segment)	
code (text segment)	← PC

Process states

- Each process has an execution state, which indicates what it is currently doing
  - ready: waiting to be assigned to CPU
    - could run, but another process has the CPU
  - running: executing on the CPU
    - is the process that currently controls the CPU
    - pop quiz: how many processes can be running simultaneously?
  - waiting: waiting for an event, e.g. I/O
    - cannot make progress until event happens
- As a process executes, it moves from state to state
  - UNIX: run ps, STAT column shows current state
  - which state is a process is most of the time?

Process state transitions
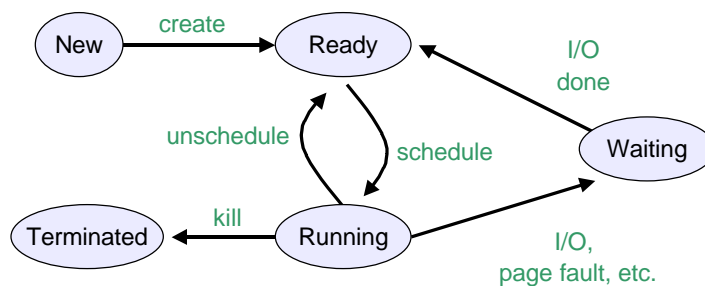
- What can cause schedule/unschedule transitions?

Process data structures

- How does the OS represent a process in the kernel?
  - at any time, there are many processes, each in its own particular state
  - the OS data structure that represents each is called the process control block (PCB)
- PCB contains all info about the process
  - OS keeps all of a process' hardware execution state in the PCB when the process isn't running
    - PC
    - SP
    - registers
  - when process is unscheduled, the state is transferred out of the hardware into the PCB

PCB

- The PCB is a data structure with many, many fields:
  - process ID (PID)
  - execution state
  - program counter, stack pointer, registers
  - memory management info
  - UNIX username of owner
  - scheduling priority
  - accounting info
  - pointers into state queues
- In linux:
  - defined in `task_struct` (`include/linux/sched.h`)
  - over 95 fields!!!

PCBs and Hardware State

- When a process is running, its hardware state is inside the CPU
  - PC, SP, registers
  - CPU contains current values
- When the OS stops running a process (puts it in the waiting state), it saves the registers' values in the PCB
  - when the OS puts the process in the running state, it loads the hardware registers from the values in that process' PCB
- The act of switching the CPU from one process to another is called a context switch
  - timesharing systems may do 100s or 1000s of switches/s
  - takes about 5 microseconds on today's hardware

State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
  - typically one queue for each state
    - e.g., ready, waiting, …
  - each PCB is queued onto a state queue according to its current state
  - as a process changes state, its PCB is unlinked from from queue, and linked onto another
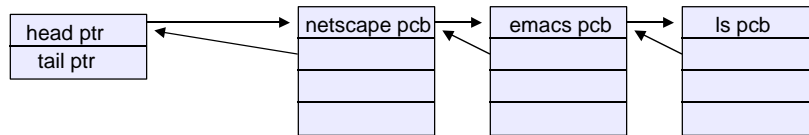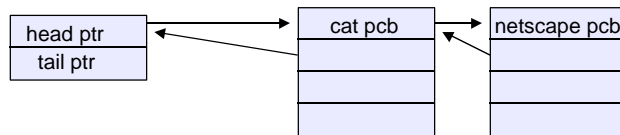
State queues

Ready queue header

head ptr	
tail ptr	

netscape pcb

emacs pcb

ls pcb

Wait queue header

head ptr	
tail ptr	

cat pcb

netscape pcb

- There may be many wait queues, one for each type of wait (particular device, timer, message, …)

PCBs and State Queues

- PCBs are data structures
  - dynamically allocated inside OS memory
- When a process is created:
  - OS allocates a PCB for it
  - OS initializes PCB
  - OS puts PCB on the correct queue
- As a process computes:
  - OS moves its PCB from queue to queue
- When a process is terminated:
  - OS deallocates its PCB

Process creation

- One process can create another process
  - creator is called the parent
  - created process is called the child
  - UNIX: do `ps`, look for PPID field
  - what creates the first process, and when?
- In some systems, parent defines or donates resources and privileges for its children
  - UNIX: child inherits parents userID field, etc.
- when child is created, parent may either wait for it to finish, or it may continue in parallel, or both!

UNIX process creation

- UNIX process creation through `fork()` system call
  - creates and initializes a new PCB
  - creates a new address space
  - initializes new address space with a copy of the entire contents of the address space of the parent
  - initializes kernel resources of new process with resources of parent (e.g. open files)
  - places new PCB on the ready queue
- the `fork()` system call returns twice
  - once into the parent, and once into the child
  - returns the child's PID to the parent
  - returns 0 to the child

fork( )

```
int main(int argc, char **argv)
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s is %d\n",
           name, child_pid);
    return 0;
  } else {
    printf("My child is %d\n", child_pid);
    return 0;
  }
}
```

output

```
spinlock% gcc -o testparent testparent.c
spinlock% ./testparent
My child is 486
Child of testparent is 0
spinlock% ./testparent
Child of testparent is 0
My child is 486
```

Fork and exec

- So how do we start a new program, instead of just forking the old program?
  - the exec() system call!
  - int exec(char *prog, char ** argv)
- exec()
  - stops the current process
  - loads program 'prog' into the address space
  - initializes hardware context, args for new program
  - places PCB onto ready queue
  - note: does not create a new process!
- what does it mean for exec to return?
  - what happens if you "exec csh" in your shell?
  - what happens if you "exec ls" in your shell?

UNIX shells

```
int main(int argc, char **argv)
{
  while (1) {
    char *cmd = get_next_command();
    int child_pid = fork();
    if (child_pid == 0) {
       manipulate STDIN/STDOUT/STDERR fd's
       exec(cmd);
       panic("exec failed!");
    } else {
       wait(child_pid);
    }
  }
}
```

© 2001 Steve Gribble