Module 4
Processes

Ed Lazowska
lazowska@cs.washington.edu
Allen Center 570

1/2/2006 © 2006 Gribble, Lazowska, Levy

Process management

• This module begins a series of topics on processes, threads, and synchronization
  – this is the most important part of the class
  – there definitely will be several questions on these topics on the midterm
• 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 and stack pointer (SP)
  – traces state of procedure calls made
  – the program counter (PC), indicating the next instruction
  – general-purpose processor registers and their values
  – a set of OS resources
    • open files, network connections, sound channels, ...
• In other words, it’s all the stuff you need to run the program
  – or to re-start it, if it’s interrupted at some point

The process control block

• There’s a data structure called the process control block (PCB) that holds all this stuff
  – The PCB is identified by an integer process ID (PID)
• OS keeps all of a process’s hardware execution state in the PCB when the process isn’t running
  – PC, SP, registers, etc.
  – when a process is unscheduled, the state is transferred out of the hardware into the PCB
• Note: It’s natural to think that there must be some esoteric techniques being used
  – fancy data structures that’d you’d never think of yourself
    Wrong! It’s pretty much just what you’d think of!

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
  – 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 in most of the time?
States of a process

running
  dispatch/schedule
  interrupt (unschedule)

ready
  exception (I/O, page fault, etc.)

interrupt (I/O complete)

blocked

You can create and destroy processes!

The PCB revisited

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

A process’s address space

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’s 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/sec.
  – 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 the current state of the process it represents
  – as a process changes state, its PCB is unlinked from one queue, and linked onto another
• Once again, this is just as straightforward as it sounds! The PCBs are moved between queues, which are represented as linked lists. There is no magic!
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 parent's userID field, etc.
- When child is created, parent may either wait for it to finish, or may continue in parallel, or both!

UNIX process creation

- UNIX process creation through \texttt{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 \texttt{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
- \texttt{fork()} = "clone me"

\texttt{testparent} – use of \texttt{fork()}

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

\texttt{testparent} 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 571
```
Exec vs. fork

• 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!

UNIX shells

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