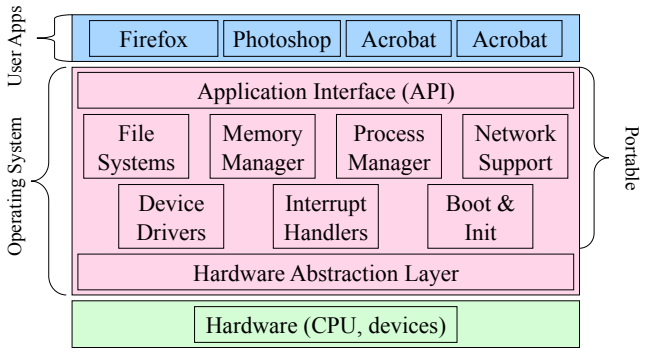
CS 537
Lecture 3: Processes

Michael Swift

Process Management

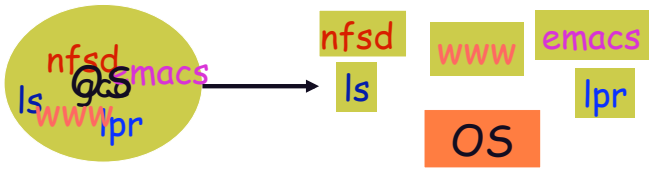
- Today: processes and process management
  - what are the OS units of execution?
  - how are they represented inside the OS?
  - how is the CPU scheduled across processes?
  - what are the possible execution states of a process?
    - and how does the system move between them?

Example OS in operation



Why Processes? Simplicity + Speed

- Hundreds of things going on in the system



- How to make things simple?
  - Separate each in an isolated process
  - Decomposition
- How to speed-up?
  - Overlap I/O bursts of one process with CPU bursts of another

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

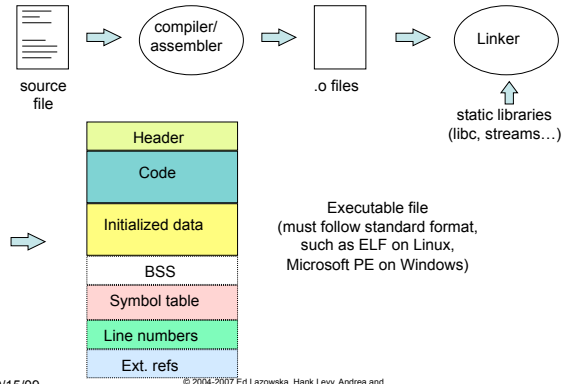
What is a program?

A program consists of:

- **Code:** machine instructions
- **Data:** variables stored and manipulated in memory
  - initialized variables (globals)
  - dynamically allocated variables (malloc, new)
  - stack variables (C automatic variables, function arguments)
- **DLLs:** libraries that were not compiled or linked with the program
  - containing code & data, possibly shared with other programs
- **mapped files:** memory segments containing variables (mmap())
  - used frequently in database programs

- Whats the relationship between a program and process?
  - *A process is a executing program*

Preparing a Program



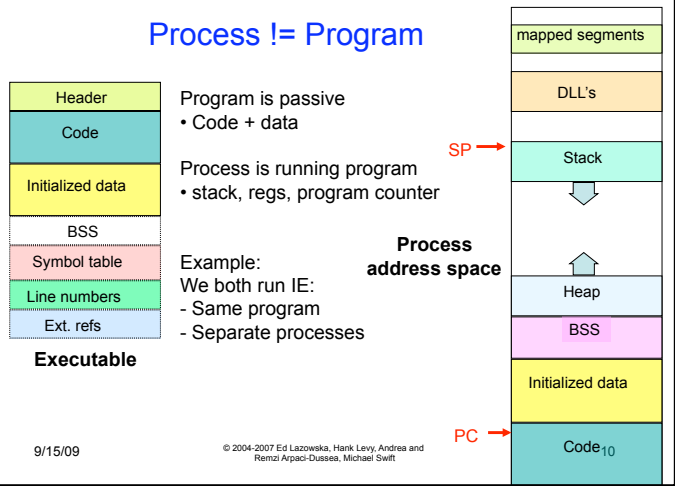
Running a program

- OS creates a "process" and allocates memory for it
- The loader:
  - reads and interprets the executable file
  - sets process's memory to contain code & data from executable
  - pushes "argc", "argv", "envp" on the stack
  - sets the CPU registers properly & calls "__start()" [Part of CRT0]
- Program start running at __start(), which calls main()
  - we say "process" is running, and no longer think of "program"
- When main() returns, CRT0 calls "exit()"
  - destroys the process and returns all resources

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
  - 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

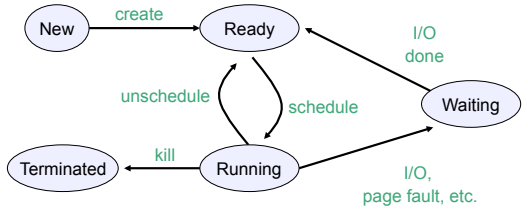
Process != Program



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!!!

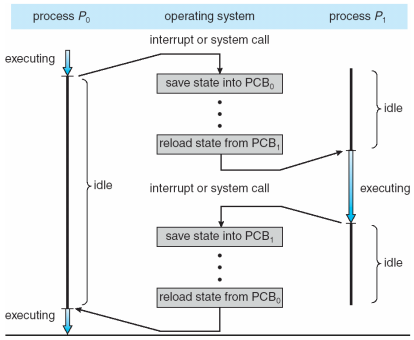
Simple Process Control Block

process state
process number
program counter
stack pointer
registers (general, FP)
memory management info
username of owner
queue pointers for state queues
scheduling info (priority, etc.)
accounting info

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

CPU Switch From Process to Process



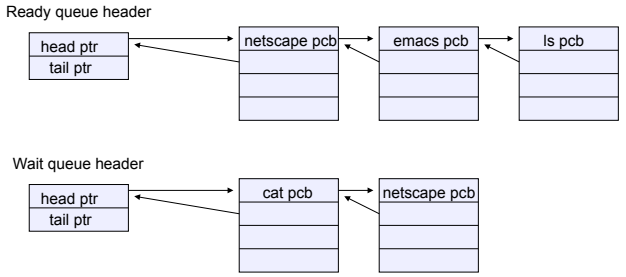
Details of Context Switching

- Very tricky to implement
  - OS must save state without changing state
  - Should run without touching any registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
      - Or way to save a register and then continue
- Overheads: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Wait for pipeline to drain in pipelined processors

State queues

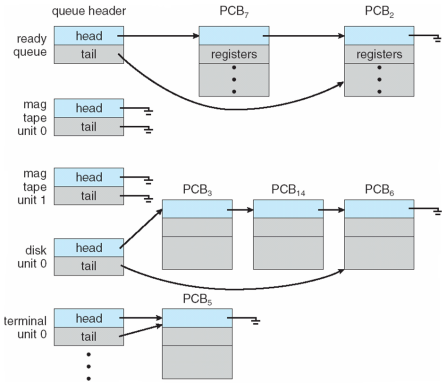
- The OS maintains a collection of queues that represent the state of all processes in the system
  - typically one queue for each state
    - e.g., ready, waiting, …
  - each PCB is queued onto a state queue according to its current state
  - as a process changes state, its PCB is unlinked from from queue, and linked onto another
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device

State queues



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

Ready Queue And Various I/O Device Queues



How to create a process?

- Double click on a icon?
- After boot OS starts the first process
  - E.g. sched for Solaris, ntoskrnel.exe for XP
- The first process creates other processes:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships is expressed by a process tree
- For example, in UNIX the second process is called *init*
  - it creates all the gettys (login processes) and daemons
  - it should never die
  - it controls the system configuration (#processes, priorities…)
- Explorer.exe in Windows for graphical interface

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);
    }
  }
}
```

Windows *CreateProcess* function

- Open the program file to be executed
- Create the Windows executive process object
- Create the initial thread (stack, context, ...)
- Notify Win32 subsystem about new process
- Start execution of the initial thread
- Complete initialization (eg, load dlls)
- Continue execution in both processes

Copied from *Inside Windows 2000*

Which is better?

Unix

```
int child_pid = fork();
if (child_pid == 0) {
    exec("myprog");
}
```

Which is better?

- More flexible?
- Easier?
- Faster?

Windows

```
STARTUPINFO si = {0};
PROCESS_INFORMATION pi = {0};
si.cb = sizeof(si);
if( !CreateProcess(
        NULL,
        "c:\\myprog.exe 1 2",
        NULL,
        NULL,
        FALSE,
        0, NULL,
        NULL,
        &si,
        &pi ) )
{
   printf( "CreateProcess failed
      (%d).\n", GetLastError() );
   return;
}
```


Process Termination

- Process executes last statement and OS decides(**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of child process (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some OSes don't allow child to continue if parent terminates
      - All children terminated - *cascading termination*