Lecture 2: Processes

CSE 120: Principles of Operating Systems

UC San Diego: Summer Session I, 2009
Frank Uyeda
Announcements

• PeerWise accounts are now live.
  – First PeerWise questions/reviews due tomorrow (Thursday) at 11:59pm
• Homework 1 is online (due on Monday 7/6)
• Project 1 is online (due next Friday 7/10)
  – Please email me project groups ASAP.
  – Milestone testing on Monday at Midnight.
Announcements (cont.)

• No lecture on Friday!
  – Happy 4th of July

• Discussion section tomorrow @ 1pm in CSE 2154.
Review

• Operating Systems provide:
  – Resource management
  – Abstractions for application development

• Protection
  – User mode vs Kernel mode

• Hardware Support
  – Interrupts & Exceptions
Goals for Today

• What is concurrency and when do we want it?
• What are the basic units of execution in an OS?
• How are these units stored and managed in an OS?
• What are the possible states of these?
• How do they transition from state to state?
• How are they created and destroyed?
• Mechanisms for inter-process communication
Concurrency

• Uniprogramming
  – OS runs a single job until it is completed.

• Multiprogramming
  – OS has a queue of jobs
  – Selects one job to run
  – If the job has to wait for I/O or has run for awhile, the OS saves its state and puts it back into the queue.
  – Repeat last 2 steps until all jobs complete.
Which to use?

• When does it make sense to use a uniprogramming model?
  – Don’t want performance overhead of switching jobs
  – Don’t need / want additional OS complexity.

• When does it make sense to use a multiprogramming model?
  – Want to see progress on many jobs at the same time.
  – Better utilize resources when jobs wait for I/O
Concurrency (cont)

• Uniprogramming: one job/task at a time
  – Batch schedulers, such as in many supercomputing clusters, jobs run sequentially to completion; MS/DOS, execute one command at a time

• Multiprogramming: many jobs/tasks at a time
  – Time-sharing, where jobs coexist and resources are time-sliced
  – Used in personal computers and supported by nearly* all modern Operating Systems

• What do you need to support concurrency?
  – Don’t want to force applications to deal with it
  – We’d like to have some way for OS to represent (perhaps through a programming abstraction) a job or task
Process

- A process is the chosen OS abstraction for a job/task
  - Unit of execution
  - Unit of scheduling
  - Unit of accounting

- A process is the instantiation of a program
  - All processes are associated with a program
  - A program or application can have multiple processes
What’s in a Process?

Program state
- Address space
- Code
- Data
- Stack
- Registers (SP, PC, FP,..)

Program Identifier (PID)

Address Space
- 0xFFF..... (Ending Address)

- Stack
  - SP

- Heap

- Data Segment

- Text Segment
  - 0x00...... (Starting Address)
What is in a Process (2)?

• Program Information ✔

• Operating System resources
  – Open files, open network sockets

• Execution state
  – **Running**: currently using the CPU
  – **Ready**: ready to use the CPU (waiting for it)
    • Another process has CPU
  – **Waiting**: waiting for an event (not the CPU) such as I/O
    • Cannot make progress until signaled
### Unix Processes: ps command

```
[aauyoung@csegrad ~]$ ps ux
USER   PID   CPU   MEM  VSZ RSS TTY  STAT START  TIME COMMAND
aauyoung 28552  8.5  0.3  19912 12836 pts/33  S+ 15:17  0:10 pine
aauyoung 28762  0.0  0.0  12024  2448  ?    S   15:17  0:00 sshd: aauyoung@pts/10
aauyoung 28837  0.0  0.0  4004  1444 pts/10  Ss  15:17  0:00 -tcsh
aauyoung 30019  0.0  0.0   3404   760 pts/10  R+ 15:19  0:00 ps ux
aauyoung 32402  0.0  0.0  11668  2548  ?    S   13:48  0:00 sshd: aauyoung@pts/33
aauyoung 32417  0.0  0.0   4124  1500 pts/33  Ss  13:48  0:00 -tcsh
[aauyoung@csegrad ~]$  
```

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Process execution state</th>
</tr>
</thead>
<tbody>
<tr>
<td>28552</td>
<td>S+</td>
</tr>
<tr>
<td>28762</td>
<td>S</td>
</tr>
<tr>
<td>28837</td>
<td>Ss</td>
</tr>
<tr>
<td>30019</td>
<td>R+</td>
</tr>
<tr>
<td>32402</td>
<td>S</td>
</tr>
<tr>
<td>32417</td>
<td>Ss</td>
</tr>
</tbody>
</table>

What state do you think processes are in most of the time?
Process Representation

• The OS represents a process in the kernel using a Process Control Block (PCB)
  – Program information
  – OS resources
  – Execution state
  – Memory management
  – Scheduling information
  – Accounting information

• Process Control Blocks are large
  – referred to as heavyweight abstractions
  – Contains everything needed to restore the hardware to the state that it was in when the process was last running.
When/how does a process change execution state?

- **Ready**
  - unschedule process
  - schedule process
  - event occurred (e.g., I/O received)

- **Waiting**
  - blocking on event (I/O)

- **Running**
Process Transitions

• A running process has hardware state
  – Registers, memory management
  – What happens to this state when it transitions to other state?

• If the OS moves a process out of running state
  – Its hardware state must be saved in the PCB

• If the OS moves a process into the running state
  – Its hardware state is loaded into the PCB

• The process of changing the hardware state from one process to another is a context switch
  – Context switches can occur as many as 100-1000 times/sec
Context Switch

Two processes: $P_0$ and $P_1$

The OS consumes a fair amount of processing time

This time is referred to as context switching overhead

Too much context switching overhead is a bad thing (why?)

Note: image courtesy of Silberschatz, 2005
Scheduling Queues

The OS keeps track of non-running processes in scheduling queues.

There may be *many* wait queues.
- See Nachos.threads.ThreadedQueue
PCB’s and State Queues

• PCB’s are data structures dynamically allocated in OS memory.
• When a process is created, the OS allocates a PCB for it, initializes it, and places it in the Ready queue.
• As the process executes, does I/O, etc., its PCB is moved from one queue to another.
• When the process terminates, it’s PCB is deallocated.
Process Creation

• A process is created by another process
  – Created process is called child
  – Creating process is called parent
  – In Unix, parent process ID is stored as PPID

```
[aauyoung@csegrad ~]$ ps lx
  PID   PPID   PRI    NI    VSZ   RSS   WCHAN   STAT   TTY     TIME COMMAND
  17   16131  17     0    4172 648    - R+    pts/29 0:00   ps lx
  16   1819  24977  16   0 11808 2548    - S    ?      0:00   sshd: aauyoung@pts/29
  15   1819  24977  15   0  3964 1620   rt_sig Ss  pts/29 0:00   -tcsh
```

• parent-child relationships: sshd->shell->ps
Process Creation (2)

• How are parent and child processes related?
  – Is any of the state from the PCB inherited?
  – Unix: Process UID is inherited, and children inherit parent’s privileges

• What else can a parent do with a child?
  – Parent can invoke system call to wait for completion of child
  – Parent can also execute in parallel with child
Process Creation in Unix

• fork() system call is used to create a new process

```c
int fork()
```

– Creates and initializes a new PCB
– Creates a new address space
  – *Fills child’s address space with a copy of the parent’s entire address space*
  – Initializes kernel resources to point to parent’s
  – Puts PCB on the ready queue

• So…..how do we know which one is which?
  – Each process has a different PID
  – fork() returns *twice*:
    • child’s PID to the parent
    • “0” to the child
Unix fork() example

```c
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, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

What does this program print?

Note: This example courtesy of Alex Snoeren
Unix fork() example (2)

```
[aauyoung@csegrad ~/scratch]$ ./forktest
Child of ./forktest is 27766
My child is 27766
[aauyoung@csegrad ~/scratch]$ 
```

Is the order of the printed statements always going to be the same?
Unix fork() example (3)

Parent

```c
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, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

Child

```c
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, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```
Unix fork() example (4)

Parent

```c
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, getpid());
        return 0;
    }
    else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

Child

```c
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, getpid());
        return 0;
    }
    else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```
Unix fork() example (5)

Parent

```c
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, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

Child

```c
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, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

Parent Pid = 27766
Child Pid = 0

My child is 27766
Child of forktest is 27766
fork() Abstraction

Very useful if

• Child is cooperating with parent
• Child shares state with parent

```c
while (1) {
    int sock = accept();
    if (child_pid == 0) {
        // handle client request
        ...
    }
    else {
        // close socket
        ...
    }
}
```

Note: This example courtesy of Alex Snoeren
Process synchronization in Unix

- wait() suspends the current process until a child process ends
  - `int wait(int *status)`
  - wait() returns -1 if error or pid of child
  - status is child’s exit status
- What if parent process exits before child?
  - PCB of child process remains
  - Child process becomes an “orphan” or “zombie” process
### Unix fork() and wait() example

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

What does this program print?

Note: This example courtesy of Alex Snoeren
Process creation in Unix (2)

• exec() system call used to stop and “restart” process
  
  ```c
  int exec(char *prog, char *argv[])
  ```
  
  – Stops the current process
  – Loads program `prog` into existing address space
  – Reinitializes hardware state
  – Puts PCB on the ready queue

• What should exec() return?
  – What about “exec csh”?
  – “exec ls”?
while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        waitpid(child_pid);
    }
}
Process Termination

• `exit(int status)` system call terminates process
  – `exit()` frees all resources consumed by process
    • Close/release all OS resources
    • Free allocated memory
    • Terminate any threads (next lecture)
    • Remove and delete PCB

• Note that the OS does not assume that a process will free everything above.
  – Why?
Process Transitions

New → Ready
- create process
- unschedule process
- schedule process
- process exit
- event occurred (e.g., I/O received)

Ready → Running
- event occurred (e.g., I/O received)
- blocking on event (I/O)

Running → Waiting

Waiting → New

Terminated → Ready

Interprocess Communication

• We can create process with fork(), but how do we get them to work together?
  – Want to pass information between processes.
• Child gets a copy of parent’s address space.
  – What if they need to communicate after fork()?
• What happens if sibling processes want to communicate?
Interprocess Communication (IPC)

• Operating System needs to provide support
  – Why?

• OS mechanisms:
  – Shared Memory
  – Message Passing
Shared Memory

Processes read and write to a region of shared memory

- A process needs to create a shared memory segment
  - Unix: `shmget()` syscall
  - Kernel creates shared memory
- Each process needs to attach to this segment
  - Unix: `shmat()` syscall
- Either process can read or write to this region
  - A: `printf(shared_mem, “hello world”);`
  - B: `printf(“%s
”, shared_mem);`
Message Passing

Processes pass messages through OS

- Create a communication link between A and B
  - OS can create mailbox for indirect communication
  - Use other means for direct communication (shared memory, network, bus)

- `send(B, message)`
  - Does it wait until received?

- `receive(A, message)`
  - Does it wait until available?

- Synchronization
  - Blocking/Synchronous
  - Non-Blocking/Asynchronous
Interprocess Communication

• Each cooperating must know the name of the other
• Shared Memory
  – Can perform frequent and large transfers
  – Must synchronize
• Message Passing
  – No explicit need for synchronization
  – Kernel overhead per message
• Other common variations of IPC
  – Unix pipes (e.g., “ls | wc”)
  – Windows Local Procedure Calls (LPCs)
Interprocess Communication

• Who uses IPC mechanisms?
  – Parallel programs (e.g., Web server, staged computations)
  – Modular programs (i.e., compose together subtasks, each being a separate processes)

• In general, IPC mechanisms are time-consuming
  – Involve system calls (kernel involved)
  – Processes need to context switch to send/receive messages
Summary

• What are the basic units of execution in an OS?
  – A process

• How are these units of execution stored and managed in an OS?
  – PCBs and in Scheduling Queues

• What are the possible states of a process?
  – Running, ready, waiting, new, terminated

• How do processes transition from state to state?
  – Scheduling events, I/O signals, process creation, termination

• How do process communicate?
  – Shared memory or Message Passing
Next Time

• Read Chapters 4 (Threads)
• First PeerWise question & reviews due tomorrow (Thursday).
• Check Web site for course announcements
  – http://www.cs.ucsd.edu/classes/su09/cse120