CS3600 — Systems and Networks
Spring 2013

Lecture 3: Processes

Prof. Alan Mislove (amislove@ccs.neu.edu)
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks

- Textbook uses the terms *job* and *process* almost interchangeably

- Process – a program in execution; process execution must progress in sequential fashion

- A process includes:
  - program counter
  - stack
  - data section
The Process

- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables (r/o and r/w)
  - **Heap** containing memory dynamically allocated during run time
- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program
Process in Memory

<table>
<thead>
<tr>
<th>max</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>heap</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>text</td>
</tr>
</tbody>
</table>
Storage of variables

#include <stdio.h>

int int1 = 1;
char *str1 = “hello”;
const char *str2 = “const”;

int main(int argc, char** argv) {
    int int2 = 0;
    char *str3 = “inner”;
    char *str4 = (char *) malloc(10*sizeof(char));
    printf(“%s -- %s
”, message, foo);
    return 0;
}

• Where are int1, int2, str1--4, and the char*s stored?
Process State

• As a process executes, it changes state
  • new: The process is being created
  • running: Instructions are being executed
  • waiting: The process is waiting for some event to occur
  • ready: The process is waiting to be assigned to a processor
  • terminated: The process has finished execution
Diagram of Process State

- **new**
- **admitted**
- **interrupt**
- **exit**
- **terminated**

- **ready**
- **running**
- **waiting**

- I/O or event completion
- scheduler dispatch
- I/O or event wait

Based on slides by Silbershatz, Galvin, and Gagne
Process Control Block (PCB)

- Kernel keeps information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

- Stored in a data structure call the Process Control Block (PCB)
Process Control Block (PCB)

- process state
- process number
- program counter
- registers
- memory limits
- list of open files
- ...

Based on slides by Silbershatz, Galvin, and Gagne
CPU Switch From Process to Process

Diagram showing the process of switching from process $P_0$ to process $P_1$. The diagram illustrates the steps involved in saving the state of $P_0$, transitioning to the operating system, and then reloading the state of $P_1$. The process starts with $P_0$ executing, then an interrupt or system call occurs, followed by saving the state into PCB$_0$. After idle time, the state is reloading from PCB$_1$. Another interrupt or system call occurs, transitioning back to executing the state of $P_1$. The process repeats with idle and reload steps.
Process Scheduling

• To maximize CPU use, want to quickly switch processes onto CPU for time sharing
• **Process scheduler** selects among available processes for next execution on CPU
• Maintains **scheduling queues** of processes
  • **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
• Processes migrate among the various queues
Process Representation in Linux

• Represented by the C structure

```c
struct task_struct {
    pid_t pid; /* process identifier */
    long state; /* state of the process */
    unsigned int time_slice /* scheduling information */
    struct task_struct *parent; /* this process’s parent */
    struct list head children; /* this process’s children */
    struct files_struct *files; /* list of open files */
    struct mm_struct *mm; /* address space of this process */
}
```
Ready Queue And Various I/O Device Queues
Representation of Process Scheduling
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into memory and put on the ready queue

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  
  - Sometimes the only scheduler in a system
Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)

- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)

- The long-term scheduler controls the *degree of multiprogramming*

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
Addition of Medium Term Scheduling
Context Switch

• When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.

• **Context** of a process represented in the PCB

• Context-switch time is overhead; the system does no useful work while switching
  • The more complex the OS and the PCB -> longer the context switch

• Time dependent on hardware support
  • Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes

- Generally, process identified and managed via a **process identifier** (pid)

- **Resource sharing**
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources

- **Execution**
  - Parent and children execute concurrently
  - Parent waits until children terminate
Process Creation (Cont.)

• Address space
  • Child duplicate of parent
  • Child has a program loaded into it

• UNIX examples
  • `fork` system call creates new process
    • How to tell apart new (child) and old (parent) process?
  
  • `exec` system call used after a `fork` to replace the process’ memory space with a new program
Process Creation
C Program Forking Separate Process

```c
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execl("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```
A Tree of Processes on Solaris

- **Sched**
  - **pid = 0**

- **init**
  - **pid = 1**
    - **inetd**
      - **pid = 140**
      - **telnetdaemon**
        - **pid = 7776**
      - **Csh**
        - **pid = 7778**
    - **Netscape**
      - **pid = 7785**
    - **emacs**
      - **pid = 8105**

- **pageout**
  - **pid = 2**
    - **dtlogin**
      - **pid = 251**
    - **Xsession**
      - **pid = 294**
    - **sdt_shel**
      - **pid = 340**
    - **Csh**
      - **pid = 1400**

- **fsflush**
  - **pid = 3**
    - **ls**
      - **pid = 2123**
    - **cat**
      - **pid = 2536**
Process Termination

• Process executes last statement and asks the operating system to delete it (exit)
  • Output data from child to parent (via wait)
  • Process’ resources are deallocated by operating system

• Parent may terminate execution of children processes (abort)
  • Child has exceeded allocated resources
  • Task assigned to child is no longer required
  • If parent is exiting
    • Some operating systems do not allow child to continue if its parent terminates

  All children terminated - cascading termination
Process I/O

- Open files with
  - int open(char *path, int flags)
  - flags allow process to specify read, write, truncate, append
  - returned int is file descriptor
    - Use in subsequent file I/O methods
    - File descriptors are inherited by children

- Other operations
  - int read (int fd, void *buf, int length)
  - int write (int fd, void *buf, int length)
  - int lseek (int fd, off_t pos)
  - int close(int fd)

- Special descriptors exist
  - 0 (stdin), 1 (stdout), 2 (stderr) -- normally attached to terminal