Process Management

Dickinson College
Computer Science 354
Spring 2006

slides courtesy of Professor Grant Braught

Road Map

- Past:
  - What an OS is, why we have them, what they do.
  - Base hardware and support for operating systems
- Present:
  - Process Management
- Future:
  - Process Scheduling
  - Concurrent programming
  - Memory management
  - Storage management
  - Protection and Security

Process Management

- Outline
  - What is process management?
  - Creating and managing processes:
    - From user perspective
    - From program perspective
      - Project #1
        - C++ code examples
    - From OS perspective
  - Inter-process Communication

Process Management

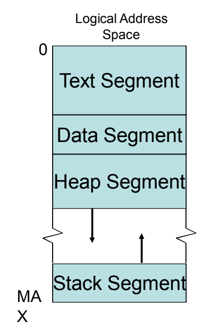
- The *process manager* must provide for:
  - Process creation
  - Process termination
  - Process synchronization
  - Inter-process communication
  - Process scheduling

Process

- A *process* is a program in execution.

A Process in Memory

- A process' *logical address space* consists of four segments.
  - Text segment: the executable program code (read-only).
    - a.k.a. Code segment
  - Data segment: global variables.
  - Heap segment: dynamically allocated memory (e.g. new)
  - Stack segment: actual parameters and local variables.



Creating Processes from the User's Perspective

Where do processes come from?

- The OS initialization routine starts some processes that perform system services (e.g. servers and daemons).
- Other processes are created by the user:
  - Double clicking an icon in a GUI
  - Entering a command in a shell
  - Voice commands
    - From the OS perspective, all processes are created in the same way.

Shells and User Created Processes

- Unix Shells (e.g. sh, bash, tsh, csh)
  - New processes are created when programs are executed by entering the name of their executable file on the command line.
    - Full paths:
      - /usr/XllR6/bin/xeyes
    - Relative paths:
      - bin/xeyes
        - Works if current directory is /usr/X11R6
      - ./xeyes
        - Works if current directory is /usr/XllR6/bin
        - . indicates the current directory
      - ../bin/xeyes
        - Works if current directory is a sub directory of /usr/X11R6
        - .. indicates the parent directory

Environment Variables

- Shells typically read a number of files on startup that define *environment variables* that make it easier to perform common operations:
  - The PATH variable
    - Defines a sequence of directories in which to look for programs that are being executed.
      - e.g. PATH=/bin:/usr/bin
  - The HOME variable
    - Defines the user's home directory.
      - Using cd without any argument returns to the HOME directory.

Useful Unix Commands

- Some Unix commands:
  - cd <dir>
  - pwd
  - ls [-al] [<dir>]
  - which <file>
  - man <topic>
  - rm <file>
  - rmdir <directory>
  - &

Shell built-ins vs. Programs

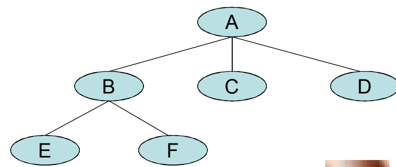
- Some Unix commands are built into the shell and others are external programs.
  - cd and pwd are built into the shell
  - ls, rm, which and lots of others are programs

Seeing Processes in Unix

- The ps command displays the processes currently executing on a Unix machine.
  - ps
  - ps -ax
  - ps -O pid,ppid,command
    - pid = process identifier
    - ppid = parent process identifier
    - Use man ps to find out about all the options.

Process Trees

- In general processes form a tree:
  - Parents, children, siblings, grand children, grand parents etc…
    - In Unix, the init process (pid=1) is at the root of the tree.



Terminating Processes

- The kill command can be used to terminate processes in Unix.
  - kill -s KILL <pid>

Random OS Quote

- I'm not one of those who think Bill Gates is the devil. I simply suspect that if Microsoft ever met up with the devil, it wouldn't need an interpreter.

  Nicholas Petreley

Creating Processes from a Program's Perspective

Process Creation in Unix

- Programs use four system calls when creating and managing new processes in Unix.
  - fork
  - wait
  - exit
  - exec

The fork System Call

- The fork system call:
  - Constructs a new logical address space and context for the child that are identical to that of the parent.
    - Data, stack and heap segments are cloned.
    - Code segment may be cloned or shared with the parent.
    - Register contents are identical.
    - PC value is identical.
  - Returns different values in the parent and the child
    - Child gets return value of 0.
    - Parent gets child's pid as the return value.

fork Example

```
#include <iostream>        // needed for cout
#include <unistd.h>        // needed for fork
#include <sys/wait.h>      // needed for wait

using namespace std;

int main() {
  cout << "Parent running" << endl;

  int pid = fork();

  if (pid != 0) {
    cout << "Parent running after fork" << endl;
    wait(NULL);
    cout << "Parent done" << endl;
  }
  else {
    cout << "Child running" << endl;
    sleep(5);
    cout << "Child done" << endl;
  }
}
```

wait and exit

- wait(NULL)
  - Causes a process to wait until any one of its child processes has completed.
    - The waitpid system call can be used to wait for a specific child process to complete.
- exit(int)
  - Causes the program to exit with the main method returning the specified value.
    - e.g. exit(-1);
    - Reaching the end of the main method results in an implicit exit(0).

fork Puzzle

- What is the output of this program?

```
int main() {
  int x = 27;
  int pid = fork();
  if (pid != 0) {
    cout << "Parent's x before wait is " << x << endl;
    x = x + 5;
    wait(NULL);
    cout << "Parent's x after wait is " << x << endl;
  }
  else {
    cout << "Child's x before sleep is " << x << endl;
    sleep(5);
    x = x + 10;
    cout << "Child's x after sleep is " << x << endl;
  }
}
```

Another fork Puzzle

- What will the output of this program look like?

```
int main() {
  int pid = fork();
  if (pid != 0) {
    for (int i=0; i<10000; i++) {
      cout << "Parent process running." << endl;
    }
    wait(NULL);
  }
  else {
    for (int i=0; i<10000; i++) {
      cout << "Child process running." << endl;
    }
  }
}
```

The exec System Call

- The exec system call transforms the calling process into a new process.
  - Code and data segments are determined by specifying a new executable file.
  - Stack and heap segments are initially empty.
  - PC is set to the start of the new program.
  - PID & PPID are inherited from calling process.
  - Typically, a process will use fork to create a child process and then the child will use an exec call to load and execute a new program.

C/C++ Library Interface

- C/C++ provides a variety of library functions that wrap exec system calls:
  - execl, execlp, execle, exect, execv, execvp
  - We will be using the execv function:
    - int execv(<prog>, <args>)
      - <prog>: A *C-style string* indicating the executable file for the new process.
      - <args>: An array of *C-style strings* providing the command line arguments to the new process.

execv Example

```
int main() {
   cout << "Parent running" << endl;
   int pid = fork();
   if (pid != 0) {
      wait(NULL);
      cout << "Parent done" << endl;
   }
   else {
      cout << "Child running" << endl;
      char *prog = "/bin/ls";  // full or relative path
      char *args[3];
      args[0]="ls";  // args[0] is name of program
      args[1]="-l";  // command line arguments…
      args[2]=NULL;        // args must end with NULL
      int rv = execv(prog, args);
      cout << "Problem with execv" << endl;
   }
}
```

What Other Systems Do

- Windows libraries provide the functions:
  - CreateProcess
  - TerminateProcess
  - WaitForSingleObject
- The Java class libraries provide the methods:
  - Runtime.exec
  - Process.waitFor
  - System.exit

Random OS Quote

- Operating systems are like underwear — nobody really wants to look at them.

Bill Joy

Project #1 and C/C++ Program Examples

Project #1

- Part #1:
  - Write some C/C++ programs for practice with using fork, wait and execv.
- Part #2:
  - Write a shell program in C/C++.
    - Read and execute commands entered by the user.
  - Assignment is on-line.

string in C++

```
#import <iostream>
#import <string>

using namespace std;

int main() {
    string s1 = "Test String";
    string s2 = "Another thing";

    int len1 = s1.length();
    cout << len1 << endl;    // 11

    string s3 = s1 + " " + s2;
    cout << s3 << endl;      // "Test String Another Thing"

    string s4 = s1.substr(0,4);      // "Test"
    cout << s4 << endl;
    string s5 = s2.substr(4,3);      // "her"
    cout << s5 << endl;
    string s6 = s2.substr(8);  // "thing"
    cout << s6 << endl;
    …
}
```

string in C++

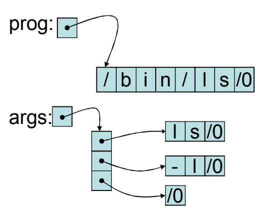
```
int main() {
    string s1 = "Test String";
    string s2 = "Another thing";
    …
    int loc1 = s1.find('t');
    cout << loc1 << endl;        // 3
    int loc2 = s2.find("th");
    cout << loc2 << endl;        // 3
    int loc3 = s2.find("zz");
    cout << loc3 << endl;        // -1 == string::npos
    int loc4 = s1.rfind('t');
    cout << loc4 << endl;        // 6
    int loc5 = s2.rfind("th");
    cout << loc5 << endl;        // 8
    int loc6 = s2.rfind("zz");
    cout << loc6 << endl;        // -1 == string::npos
}
```

Strings in C

- In C, a string is a pointer to an array of characters.
  - ✓ The last character in the array must always be NULL ('/0') to indicate the end of the string.

```
char *prog = "/bin/ls";

char *args[3];
args[0]="ls";
args[1]="-l";
args[2]=NULL;
```



Converting a C++ string to a C-style string

- C++ strings are easy to work with, but sometimes a function will require C-style strings as arguments (e.g. execv).
  - ✓ The c_str function in the string class creates a new C-style string and returns the pointer to it.

```
string cppStr = "Test String";
char *cStr;
cStr = (char *)cppStr.c_str();
```

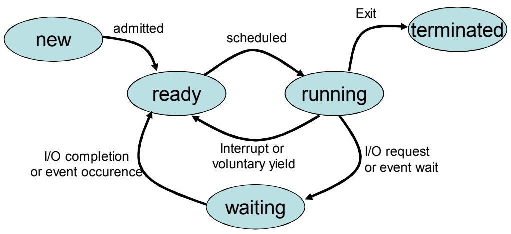
SimpleShell.cpp

- The SimpleShell.cpp program:
  - ✓ Reads one line of input from the user
  - ✓ Attempts to create a new process and execute the program indicated by the input.
    - Waits for the child process to complete.
    - Exits
- Project asks you to extend the SimpleShell:
  - ✓ Read commands until exit command is entered.
  - ✓ Allow working directory to be changed.
  - ✓ Implement PATH and HOME functionality.
  - ✓ Add & functionality.

Creating Processes from the Operating System's Perspective

Process States in the OS

- From the OS perspective a process moves among five different logical states during its lifetime.

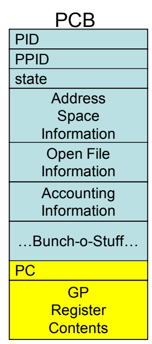


Process Manager Data Structures

- To keep track of processes the process manager relies on several data structures:
  - ✓The Process Control Block (PCB)
  - ✓The Ready Queue
  - ✓Device Queues

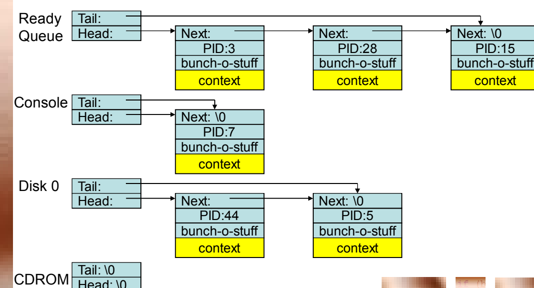
Process Control Block

- The PCB is a structure used by the OS to hold all of the information it needs to know about a process:
  - ✓PID, PPID
  - ✓Process state (e.g. ready)
  - ✓Address space info
    - Base/Limit or VM info
  - ✓Open files
  - ✓Accounting Information:
    - CPU/Real time used
    - Time/Resource limits
  - ✓Context when suspended
    - PC, GP Register values



Ready and Device Queues

- The OS stores each PCB in the *ready queue*, a *device queue* or one of several other types of queues, reflecting the currrent state of the process.

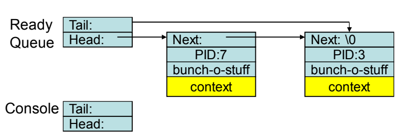


Queue Example

- Process #7 is running
  - ✓Executes getln instruction
    - Call to C++ library
      - C++ Library makes a system call to read from console.
        - Routine in OS kernel is invoked.

<u>Process #7</u>

```
int main() {
  …
  string cmd;
  getln(cin,cmd);
  cout << cmd
  …
}
```

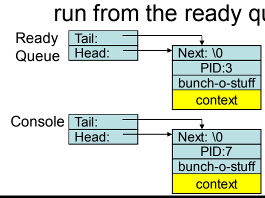


Queue Example

- The OS handles system call
  - ✓Process #7 is *blocked* on the Console device queue.
  - ✓OS selects new process to run from the ready queue.

<u>Process #7</u>

```
int main() {
  …
  string cmd;
  getln(cin,cmd);
  cout << cmd;
  …
}
```

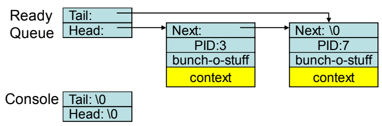


Queue Example

- User enters text on console
  - ✓ Console generates an interrupt
  - ✓ Interrupt handler in OS is invoked.
    - OS makes data available to process #7
    - Moves process #7 back to ready queue
    - Process #7 resumes within C++ libraray
      - Library code copies data into cmd.
      - Returns control to main

<u>Process #7</u>

```
int main() {
  …
  string cmd;
  getln(cin,cmd);
  cout << cmd;
  …
}
```



Random OS Quote

- "There are people who don't like capitalism, and people who don't like PCs. But there's no-one who likes the PC who doesn't like Microsoft"

  Bill Gates

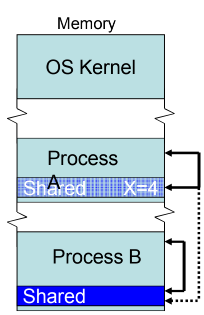
Interprocess Communication Mechanisms

IPC Mechanisms

- There are two main ways that operating systems use to implement interprocess communication (IPC).
  - ✓ Message Passing IPC
  - ✓ Shared Memory IPC

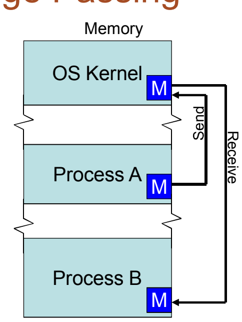
IPC via Shared Memory

- With shared memory:
  - ✓ Process B uses a system call to agree to share part of its address space.
  - ✓ Process A uses a system call to attach B's shared memory to its own address space.
  - ✓ Shared memory is then accessed like any other portion of the process' address space.



IPC via Message Passing

- In message passing:
  - ✓ Process A creates a messsage M.
  - ✓ Process A uses a send system call to send the message to process B.
    - The message is copied into memory in the kernel's address space.
  - ✓ Process B uses a receive system call to retrieve the message from A.
    - The message is copied into B's address space.



Road Map

- Past:
  - What an OS is, why we have them, what they do.
  - Base hardware and support for operating systems
  - Process Management
- Present:
  - Process Scheduling
- Future:
  - Concurrent programming
  - Memory management
  - Storage management
  - Protection and Security