Processes

Professor Jennifer Rexford

http://www.cs.princeton.edu/~jrex

1

Goals of Today's Lecture

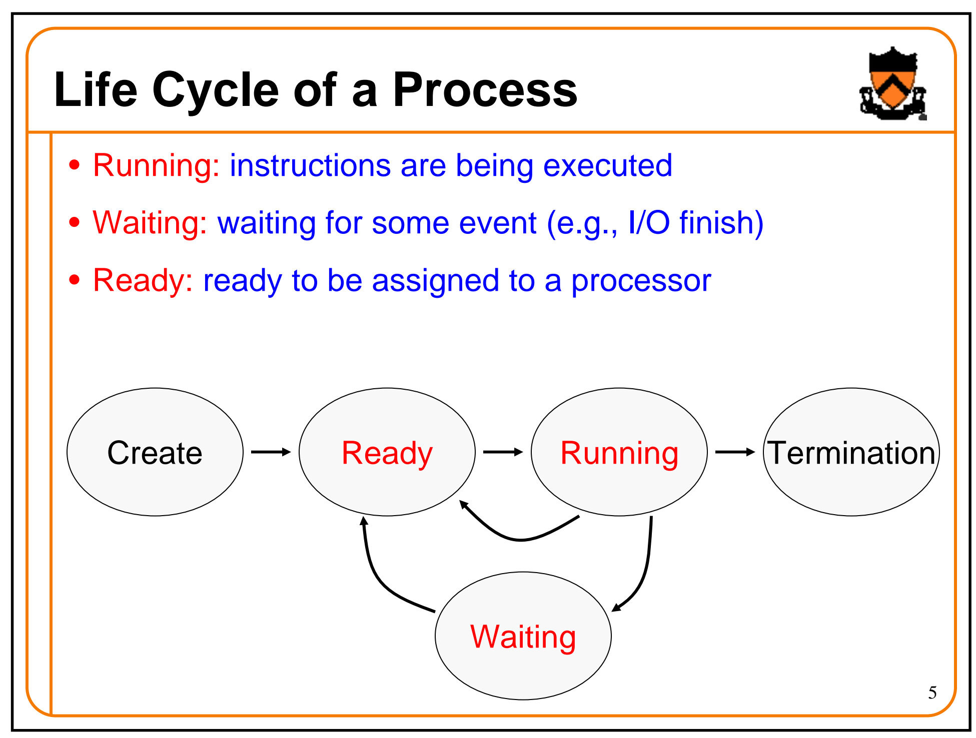
- Processes
  - Process vs. program
  - Context switching
- Creating a new process
  - Fork: process creates a new child process
  - Wait: parent waits for child process to complete
  - Exec: child starts running a new program
  - System: combines fork, wait, and exec all in one
- Communication between processes
  - Pipe between two processes
  - Redirecting stdin and stdout

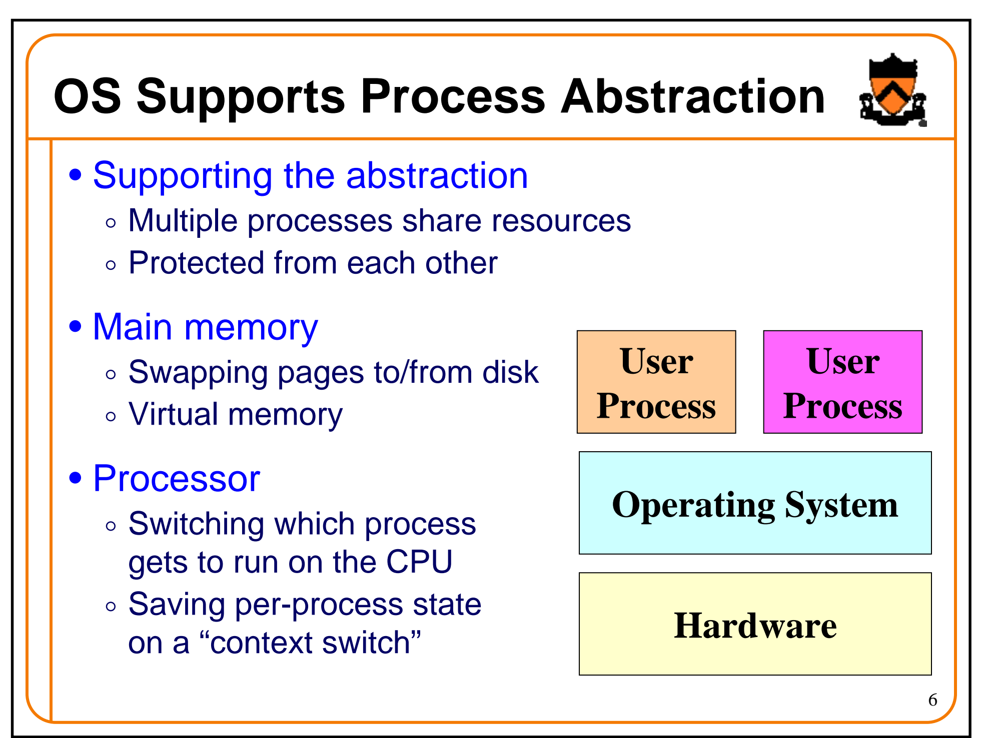
2

Processes

Program vs. Process

- Program
  - Executable code, no dynamic state
- Process
  - An instance of a program in execution, with its own
    - Address space (illusion of a memory)
      - Text, RoData, BSS, heap, stack
    - Processor state (illusion of a processor)
      - EIP, EFLAGS, registers
    - Open file descriptors (illusion of a disk)
  - Either running, waiting, or ready…
- Can run multiple instances of the same program
  - Each as its own process, with its own process ID


Life Cycle of a Process
• Running: instructions are being executed
• Waiting: waiting for some event (e.g., I/O finish)
• Ready: ready to be assigned to a processor
Create
Ready
Running
Termination
Waiting
5


OS Supports Process Abstraction
• Supporting the abstraction
◦ Multiple processes share resources
◦ Protected from each other
• Main memory
◦ Swapping pages to/from disk
◦ Virtual memory
• Processor
◦ Switching which process gets to run on the CPU
◦ Saving per-process state on a "context switch"
User Process
User Process
Operating System
Hardware
6

When to Change Which Process is Running?

- When a process is stalled waiting for I/O
  - Better utilize the CPU, e.g., while waiting for disk access

- When a process has been running for a while
  - Sharing on a fine time scale to give each process the illusion of running on its own machine
  - Trade-off efficiency for a finer granularity of fairness

7

Switching Between Processes

- Context
  - State the OS needs to restart a preempted process
- Context switch
  - Saving the context of current process
  - Restoring the saved context of some previously preempted process
  - Passing control to this newly restored process

8

Context: What the OS Needs to Save

- Process state
  - New, ready, waiting, halted
- CPU registers
  - EIP, EFLAGS, EAX, EBX, …
- I/O status information
  - Open files, I/O requests, …
- Memory management information
  - Page tables
- Accounting information
  - Time limits, group ID, ...
- CPU scheduling information
  - Priority, queues

Creating a New Process

Why Start a New Process?

- Run a new program
  - E.g., shell executing a program entered at command line
  - Or, even running an entire pipeline of commands
  - Such as "`wc -l * | sort | uniq -c | sort -nr`"
- Run a new thread of control for the same program
  - E.g., a Web server handling a new Web request
  - While continuing to allow more requests to arrive
  - Essentially time sharing the computer
- Underlying mechanism
  - A process runs "fork" to create a child process
  - (Optionally) child process does "exec" of a new program

Creating a New Process

- Cloning an existing process
  - Parent process creates a new child process
  - The two processes then run concurrently
- Child process inherits state from parent
  - Identical (but separate) copy of virtual address space
  - Copy of the parent's open file descriptors
  - Parent and child share access to open files
- Child then runs independently
  - Executing independently, including invoking a new program
  - Reading and writing its own address space

parent → child

Fork System Call

- Fork is called once
  - But returns twice, once in each process
- Telling which process is which
  - Parent: fork() returns the child's process ID
  - Child: fork() returns a 0

```
pid = fork();
if (pid != 0) {
   /* in parent */
  …
} else {
   /* in child */
   …
}
```

13

Fork and Process State

- Inherited
  - User and group IDs
  - Signal handling settings
  - Stdio
  - File pointers
  - Root directory
  - File mode creation mask
  - Resource limits
  - Controlling terminal
  - All machine register states
  - Control register(s)
  - …
- Separate in child
  - Process ID
  - Address space (memory)
  - File descriptors
  - Parent process ID
  - Pending signals
  - Time signal reset times
  - …

14

Example: What Output?

```
int main()
{
   pid_t pid;
   int x = 1;

   pid = fork();
   if (pid != 0) {
     printf("parent: x = %d\n", --x);
     exit(0);
   } else {
     printf("child: x = %d\n", ++x);
     exit(0);
   }
}
```

15

Executing a New Program

- Fork copies the state of the parent process
  - Child continues running the parent program
  - … with a copy of the process memory and registers
- Need a way to invoke a new program
  - In the context of the newly-created child process
- Example


```
execlp("ls", "ls", "-l", NULL);
fprintf(stderr, "exec failed\n");
exit(1);
```

16

Waiting for the Child to Finish

- Parent may want to wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for any some child to terminate: wait()
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for a specific child to terminate: waitpid()
  - Blocks till a child with particular process ID terminates

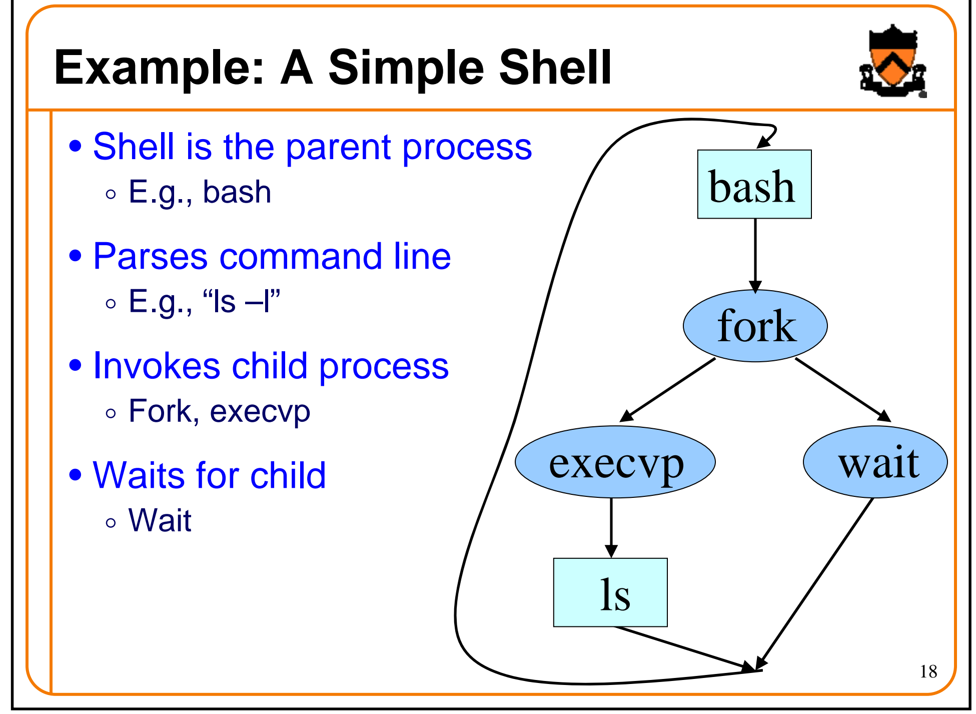
```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

17

Example: A Simple Shell

- Shell is the parent process
  - E.g., bash
- Parses command line
  - E.g., "ls –l"
- Invokes child process
  - Fork, execvp
- Waits for child
  - Wait



18

Example: A Simple Shell

```
… parse command line …
pid = fork();
if (pid == -1)
   fprintf(stderr, "fork failed\n");
else if (pid == 0) {
   /* in child */
   execvp(file, argv);
   fprintf(stderr, "exec failed\n");
} else {
   /* in parent */
   pid = wait(&status);
}
… return to top of loop
```

19

Combined Fork/Exec/Wait

- Common combination of operations
  - Fork to create a new child process
  - Exec to invoke new program in child process
  - Wait in the parent process for the child to complete
- Single call that combines all three
  - int system(const char *cmd);
- Example

```
int main()
{
   system("echo Hello world");
}
```

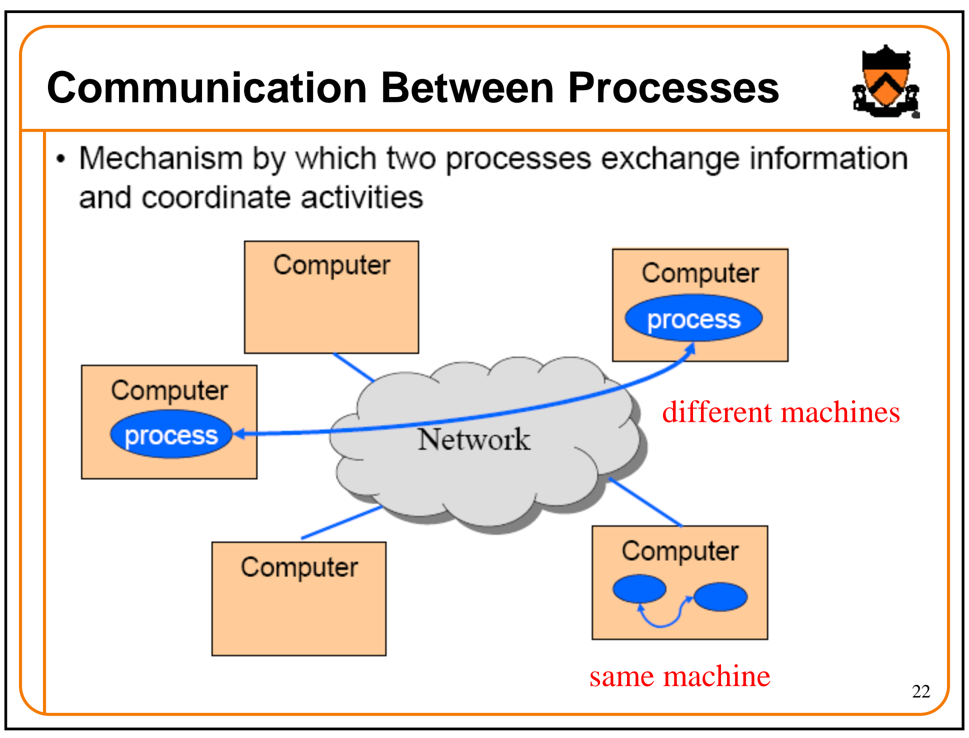
20

Communication Between Processes

21

Communication Between Processes

- Mechanism by which two processes exchange information and coordinate activities



22

Interprocess Communication

- Pipes
  - Processes on the same machine
  - One process spawns the other
  - Used mostly for a pipeline of filters
- Sockets
  - Processes on any machines
  - Processes created independently
  - Used for client/server communication (e.g., Web)

Both provide abstraction of an "ordered stream of bytes."

Pipes

- Provides an interprocess communication channel

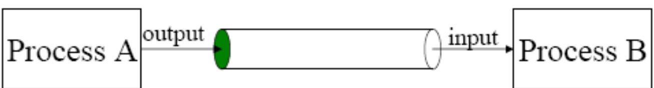
- A filter is a process that reads from `stdin` and writes to `stdout`

Example Use of Pipes

- Compute a histogram of content types in my e-mail
  - Many e-mail messages, consisting of many lines
  - Lines like "Content-Type: image/jpeg" indicate the type
- Pipeline of UNIX commands
  - Identifying content type: `grep -i Content-Type *`
  - Extracting just the type: `cut -d" " -f2`
  - Sorting the list of types: `sort`
  - Counting the unique types: `uniq -c`
  - Sorting the counts: `sort -nr`
- Simply running this at the shell prompt:
  - `grep -i Content-Type * | cut -d" " -f2 | sort | uniq -c | sort -nr`

25

Creating a Pipe



- Pipe is a communication channel abstraction
  - Process A can write to one end using "write" system call
  - Process B can read from the other end using "read" system call
- System call

```
int pipe( int fd[2] );
return 0 upon success –1 upon failure
fd[0] is open for reading
fd[1] is open for writing
```

- Two coordinated processes created by **fork** can pass data to each other using a pipe.

26

Pipe Example

```
int pid, p[2];
...
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    ... read using p[0] as fd until EOF ...
}
else {
    close(p[0]);
    ... write using p[1] as fd ...
    close(p[1]); /* sends EOF to reader */
    wait(&status);
}
```

child

parent

parent — write → [pipe] → read — child

27

Dup

- Duplicate a file descriptor (system call)

  `int dup( int fd );`

  duplicates `fd` as the lowest unallocated descriptor
- Commonly used to implement redirection of stdin/stdout
- Example: redirect stdin to "foo"

`a.out < foo`

```
int fd;
fd = open("foo", O_RDONLY, 0);
close(0);
dup(fd);
close(fd);
```

28

Dup2

- For convenience…

```
dup2( int fd1, int fd2 );
```

use `fd2 (new)` to duplicate `fd1 (old)`

closes `fd2` if it was in use

- Example: redirect stdin to "foo"

```
fd = open("foo", O_RDONLY, 0);
dup2(fd,0);
close(fd);
```

29

Pipes and Stdio

```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0],0);
    close(p[0]);
    ... read from stdin ...
}
else {
    close(p[0]);
    dup2(p[1],1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child makes stdin (0) the read side of the pipe

parent makes stdout (1) the write side of the pipe

fd=1
fd=0
parent
write
stdout
read
stdin
child

30

Pipes and Exec

```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0],0);
    close(p[0]);
    execl(...);
}
else {
    close(p[0]);
    dup2(p[1],1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child process

invokes a new program

fd=1

fd=0

parent

write
stdout

read
stdin

child

31

The Beginnings of a UNIX Shell

- A shell is mostly a big loop
  - Parse command line from stdin
  - Expand wildcards (‘*’)
  - Interpret redirections (‘|’, ‘<‘, and ‘>’)
  - Pipe, fork, dup, exec, and wait, as necessary
- Start from the code in earlier slides
  - And edit till it becomes a UNIX shell
  - This is the heart of the last programming assignment

32

Conclusion

- Processes
  - An instance of a program in execution
  - Shares CPU with other processes
  - May also communicate with other processes
- System calls for creating processes
  - Fork: process creates a new child process
  - Wait: parent waits for child process to complete
  - Exec: child starts running a new program
  - System: combines fork, wait, and exec all in one
- System calls for inter-process communication
  - Pipe: create a pipe with a write end and a read end
  - Open/close: to open or close a file
  - Dup2: to duplicate a file descriptor

33