What is an Operating System?

- Software that abstracts the computer hardware
  - Hides the messy details of the underlying hardware
  - Presents users with a resource abstraction that is easy to use
  - Extends or virtualizes the underlying machine
- Manages the resources
  - Processors, memory, timers, disks, mice, network interfaces, printers, displays, …
  - Allows multiple users and programs to share the resources and coordinates the sharing, provides protection

9/6/2012 CSC 2/456 1

Operating Systems Concepts/Components

- Processes/process management
- Memory management
- I/O device management
- File systems/storage management
- Network/communication management
- Security/protection

9/6/2012 CSC 2/456 2

Process Management

- A *process* is a program in execution
  - Unit of work – A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task
  - Protection domain
- OS responsibilities for process management:
  - Process creation and deletion
  - Process scheduling, suspension, and resumption
  - Process synchronization, inter-process communication

9/6/2012 CSC 2/456 3

Memory Management

- Memory
  - A large array of addressable words or bytes.
  - A data repository shared by the CPU and I/O devices.
- OS responsibility for memory management:
  - Allocate and deallocate memory space as requested
  - Efficient utilization when the memory resource is heavily contended
  - Keep track of which parts of memory are currently being used and by whom

9/6/2012 CSC 2/456 4

I/O System Management

- A computer needs I/O to interact with the outside world:
  - Console/terminal
  - Non-volatile secondary storage – disks
  - Networking
- The I/O system consists of:
  - A buffer-caching system
  - A general device-driver interface
  - Drivers for specific hardware devices

File and Secondary Storage Management

- A file is a collection of information defined by its user. Commonly, both programs and data are stored as files
- OS responsibility for file management:
  - Manipulation of files and directories
  - Map files onto (nonvolatile) secondary storage - disks
- OS responsibility for disk management:
  - Free space management and storage allocation
  - Disk scheduling
- They are not all always together
  - Not all files are mapped to secondary storage!
  - Not all disk space is used for the file system!

Networking and Communication

- A *distributed* system
  - A collection of processors that do not share memory
  - Processors are connected through a communication network
  - Communication takes place using a *protocol*
  - *OS provides communication end-points or sockets*
- Inter-process communication (msg, shm, sem, pipes)

System Calls and Interfaces/Abstractions

- Examples: Win32, POSIX, or Java APIs
- Process management
  - fork, waitpid, execve, exit, kill
- File management
  - open, close, read, write, lseek
- Directory and file system management
  - mkdir, rmdir, link, unlink, mount, umount
- Inter-process communication
  - sockets, ipc (msg, shm, sem, pipes)

Today

- Process management
  - Process concept
  - Operations on processes
- Signals
- Pipes

Assignment #1

- Exclusively outside of the OS
- Part I: observing the OS through the /proc virtual file system
- Part II: building a shell (command-line interpreter)
  - Support foreground/background executions
  - Support pipes

User Operating-System Interface

- Command interpreter – special program initiated when a user first logs on
- Graphical user interface
  - Common desktop environment (CDE)
  - K desktop environment (KDE)
  - GNOME desktop (GNOME)
  - Aqua (MacOS X)

System Boot

- How does the hardware know where the kernel is or how to load that kernel?
  - Use a *bootstrap* program or loader
  - Execution starts at a predefined memory location in ROM (read-only memory)
  - Read a single block at a fixed location on disk and execute the code from that boot block
  - Easily change operating system image by writing new versions to disk

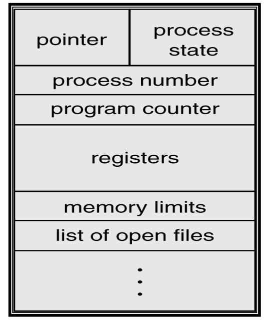
Processes

- Def: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science.
  - Not the same as “program” or “processor”
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.
- How are these Illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

Process Control Block (PCB)

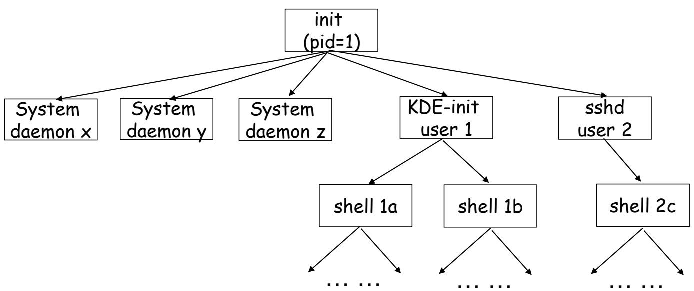
OS data structure (in kernel memory) maintaining information associated with each process.

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- Information about open files
- maybe kernel stack?



Process Tree on a Linux System

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



Unix: fork, exec; Win32API: CreateProcess

Process Creation

- When a process (parent) creates a new process (child)
  - Execution sequence?
  - Address space sharing?
  - Open files inheritance?
  - … …
- UNIX examples
  - **fork** system call creates new process with a duplicated copy of everything.
  - **exec** system call used after a **fork** to replace the process’ memory space with a new program.
  - child and parent compete for CPU like two normal processes.
- Copy-on-write

fork: Creating new processes

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's `pid` to the parent process

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

exec: Running new programs

- `int execl(char *path, char *arg0, char *arg1, …, 0)`
  - loads and runs executable at `path` with args `arg0`, `arg1`, …
    - `path` is the complete path of an executable
    - `arg0` becomes the name of the process
      - typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`
    - "real" arguments to the executable start with `arg1`, etc.
    - list of args is terminated by a `(char *)0` argument
  - returns `-1` if error, otherwise doesn't return!

```
main() {
   if (fork() == 0) {
      execl("/usr/bin/cp", "cp", "foo", "bar", 0);
   }
   wait(NULL);
   printf("copy completed\n");
   exit();
}
```


exit: Destroying Process

- `void exit(int status)`
  - exits a process
    - Normally return with status 0
  - `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

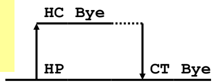

wait: Synchronizing with children

- `int wait(int *child_status)`
  - suspends current process until one of its children terminates
  - return value is the `pid` of the child process that terminated
  - if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

wait: Synchronizing with children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



Waitpid

- waitpid(pid, &status, options)
  - Can wait for specific process
  - Various options

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
       if ((pid[i] = fork()) == 0)
           exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
       pid_t wpid = waitpid(pid[i], &child_status, 0);
       if (WIFEXITED(child_status))
           printf("Child %d terminated with exit status %d\n",
                  wpid, WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
    }
```

Simple Shell eval Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) {   /* parent waits for fg job to terminate */
           int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else         /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

Problem with Simple Shell Example

- Shell correctly waits for and reaps foreground jobs.
- But what about background jobs?
  - Will become zombies when they terminate.
  - Will never be reaped because shell (typically) will not terminate.
  - Creates a memory leak that will eventually crash the kernel when it runs out of memory.
- Solution: Reaping background jobs requires a mechanism called a *signal*.

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from the kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID's
  - The only information in a signal is its ID and the fact that it arrived.

ID	Name	Default Action	Corresponding Event
2	`SIGINT`	Terminate	Interrupt from keyboard (`ctl-c`)
9	`SIGKILL`	Terminate	Kill program (cannot override or ignore)
11	`SIGSEGV`	Terminate & Dump	Segmentation violation
14	`SIGALRM`	Terminate	Timer signal
17	`SIGCHLD`	Ignore	Child stopped or terminated

Signal Concepts

- Sending a signal
  - Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process.
  - Kernel sends a signal for one of the following reasons:
    - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
    - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process.

Signal Concepts (cont)

- Receiving a signal
  - A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.
  - Three possible ways to react:
    - Ignore the signal (do nothing)
    - Terminate the process.
    - *Catch* the signal by executing a user-level function called a signal handler.
      - Akin to a hardware exception handler being called in response to an asynchronous interrupt.

Signal Concepts (cont)

- A signal is *pending* if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- A process can *block* the receipt of certain signals.
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
- A pending signal is received at most once.

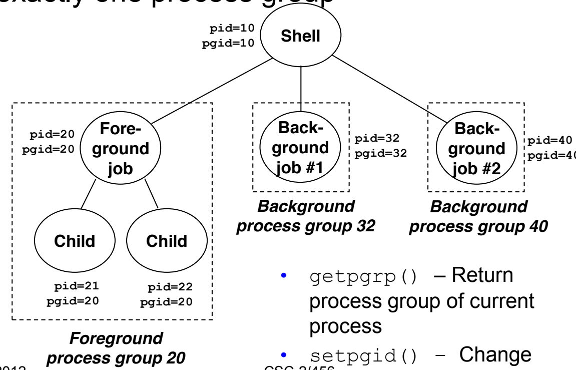
Signal Concepts (contd)

- Kernel maintains `pending` and `blocked` bit vectors in the context of each process.
  - `pending` – represents the set of pending signals
    - Kernel sets bit k in `pending` whenever a signal of type k is delivered.
    - Kernel clears bit k in `pending` whenever a signal of type k is received
  - `blocked` – represents the set of blocked signals
    - Can be set and cleared by the application using the `sigprocmask` function.

9/6/2012 CSC 2/456 29

Process Groups

- Every process belongs to exactly one process group



- `getpgrp()` – Return process group of current process
- `setpgid()` – Change process group of a process

9/6/2012 CSC 2/456 30

Sending Signals with `kill` Program

- `kill` program sends arbitrary signal to a process or process group
- Examples
  - `kill –9 24818`
    - `Send SIGKILL to process 24818`
  - `kill –9 –24817`
    - `Send SIGKILL to every process in process group 24817.`

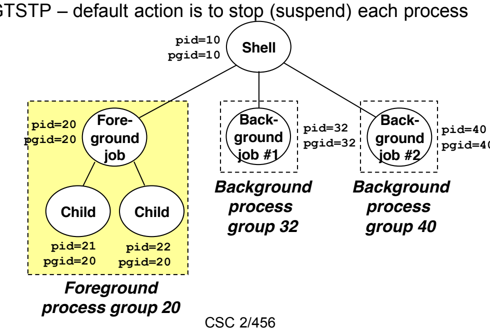
```
linux> ./forks 16
linux> Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

9/6/2012 CSC 2/456 31

Sending Signals from the Keyboard

- Typing ctrl-c (ctrl-z) sends a SIGINT (SIGTSTP) to every job in the foreground process group.
  - SIGTERM – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process



9/6/2012 CSC 2/456 32

Example of `ctrl-c` and `ctrl-z`

```
linux> ./forks 17
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
 <typed ctrl-z>
Suspended
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24867 pts/2    T      0:01 ./forks 17
24868 pts/2    T      0:01 ./forks 17
24869 pts/2    R      0:00 ps a
bass> fg
./forks 17
<typed ctrl-c>
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24870 pts/2    R      0:00 ps a
```

Sending Signals with `kill` Function

```
void fork12()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Receiving Signals

- Suppose kernel is returning from exception handler and is ready to pass control to process *p*.
- Kernel computes `pnb = pending & ~blocked`
  - The set of pending nonblocked signals for process *p*
- If (`pnb == 0`)
  - Pass control to next instruction in the logical flow for *p*.
- Else
  - Choose least nonzero bit *k* in `pnb` and force process *p* to receive signal *k*.
  - The receipt of the signal triggers some *action* by *p*
  - Repeat for all nonzero *k* in `pnb`.
  - Pass control to next instruction in logical flow for *p*.

Default Actions

- Each signal type has a predefined *default action*, which is one of:
  - The process terminates
  - The process terminates and dumps core.
  - The process stops until restarted by a SIGCONT signal.
  - The process ignores the signal.

Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`
- Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum`
  - SIG_DFL: revert to the default action on receipt of signals of type `signum`.
  - Otherwise, handler is the address of a *signal handler*
    - Called when process receives signal of type `signum`
    - Referred to as "*installing*" the handler.
    - Executing handler is called "*catching*" or "*handling*" the signal.
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

9/6/2012 CSC 2/456 37

Signal Handling Example

```
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
           getpid(), sig);
    exit(0);
}

void fork13()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);

    . . .
}
```

```
linux> ./forks 13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
linux>
```

Signal Handler Funkiness

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received signal %d from process %d\n",
           sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause();/* Suspend until signal occurs */
}
```

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal

Living With Nonqueuing Signals

- Must check for all terminated jobs
  - Typically loop with `wait`

```
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = wait(&child_status)) > 0) {
       ccount--;
       printf("Received signal %d from process %d\n",
sig, pid);
    }
}

void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

A Program That Reacts to Externally Generated Events (ctrl-c)

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
  printf("You think hitting ctrl-c will stop the bomb?\n");
  sleep(2);
  printf("Well...");
  fflush(stdout);
  sleep(1);
  printf("OK\n");
  exit(0);
}

main() {
  signal(SIGINT, handler); /* installs ctl-c handler */
  while(1) {
  }
}
```

A Program That Reacts to Internally Generated Events

```
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  printf("BEEP\n");
  fflush(stdout);

  if (++beeps < 5)
    alarm(1);
  else {
    printf("BOOM!\n");
    exit(0);
  }
}
```

```
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {
    /* handler returns here */
  }
}
```

```
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```

Interprocess Communication: Pipes

- Conduit allowing two processes to communicate
  - Unidirectional or bidirectional
  - Full-duplex or half-duplex two-way communication
  - Is parent-child relationship required?
  - Is communication across a network allowed?

Unix Pipes

- A unidirectional data channel that can be used for interprocess communication
- Treated as a special type of file, accessed using read() and write()
- Cannot be accessed from outside the process that created it unless inherited (by a child)
- Pipe ceases to exist once closed or when process terminates
- System calls
  - pipe (int fd[])
  - dup2

Example

- pipe(int fd[]) – fd[0] = read_end, fd[1]=write_end

```
    int fd[2];
pid_t pid;

pipe(fd);
pid = fork();
if (pid > 0) {/* parent process */
    close(fd[0]);
    write(fd[1], write_msg, strlen(write_msg)+1);
    close(fd[1]);
}
else {/* child process */
    close(fd[1]);
    read(fd[0], read_msg, BUFFER_SIZE);
    printf("read %s", read_msg);
    close(fd[0];
}
```

9/6/2012 CSC 2/456 45

Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, Dave O'Hallaron, Randal Bryant, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, and Kai Shen. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).

9/6/2012 CSC 2/456 68