Signals, Processes, & Threads

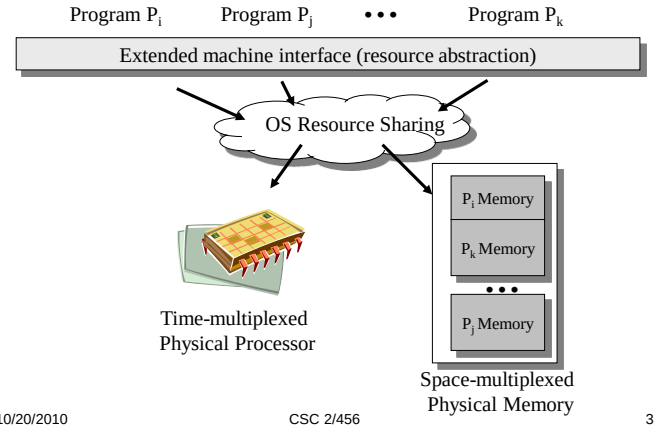
CS 256/456

Dept. of Computer Science, University of Rochester

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

Resource Sharing



Resource Abstraction

```
load(block, length, device);
seek(device, track);
out(device, sector)
```

```
write(char *block, int len, int device,
                  int track, int sector) {
   ...
  load(block, length, device);
  seek(device, track);
  out(device, sector);
   ...
}
```

```
write(char *block, int len, int device,int addr);
```

```
fprintf(fileID, "%d", datum);
```

Under the Abstraction

- functional complexity
- a single abstraction over multiple devices
- replication → reliability

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

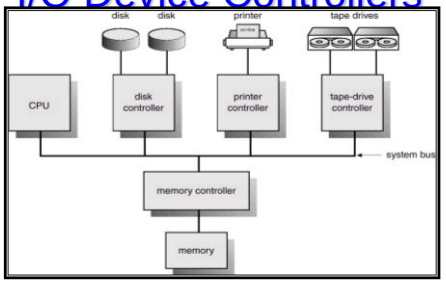
System Calls and Interfaces/Abstractions

- 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)

Today

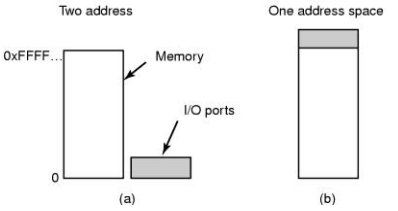
- System calls
  - Signals and pipes
- Process
  - Process concept
  - A process's image in a computer
  - Operations on processes

I/O Device Controllers



- I/O devices have both mechanical component & electronic component
- The electronic component is the device controller
  - It contains control logic, command registers, status registers, and on-board buffer space

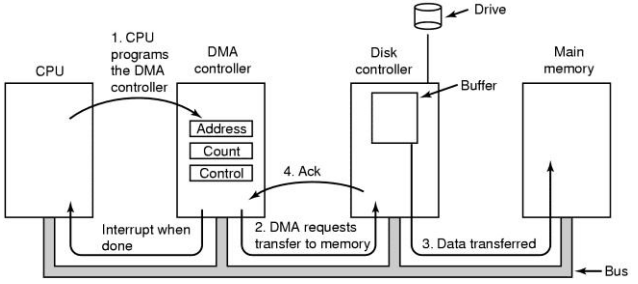
I/O Ports & Memory-Mapped I/O



I/O methods:
- Separate I/O and memory space; special I/O commands (IN/OUT)
- Memory-mapped I/O

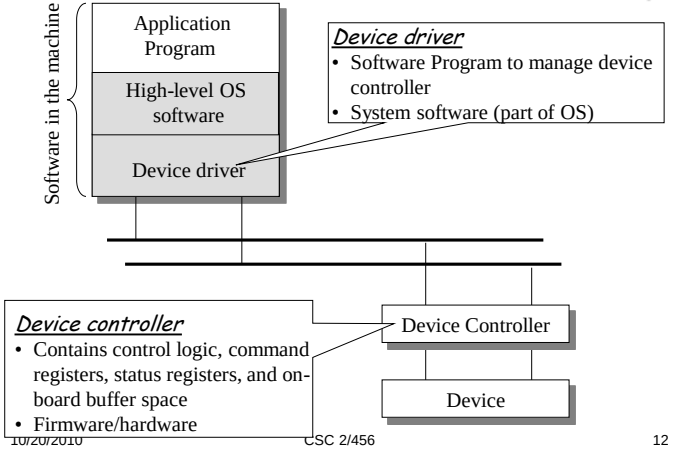
Issues with them:
- Convenience/efficiency when using high-level language;
- Protection mechanisms;
- Data caching

Direct Memory Access (DMA)



- Are the addresses CPU sends to the DMA controller virtual or physical addresses?
- Can the disk controller directly read data into the main memory (bypassing the controller buffer)?

The Device-Controller-Software Relationship



I/O Operations

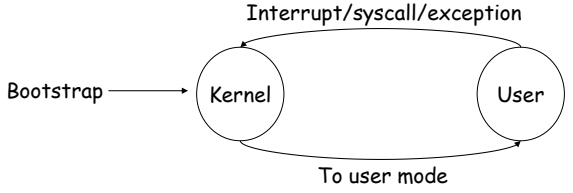
- How is I/O done?
  - I/O devices are much slower than CPU
- Synchronous (polling)
  - After I/O starts, busy-wait while polling the device status register until it shows the operation completes
- Asynchronous (interrupt-driven)
  - After I/O starts, control returns to the user program without waiting for I/O completion
  - Device controller later informs CPU that it has finished its operation by causing an *interrupt*
  - When an interrupt occurs, current execution is put on hold; the CPU jumps to a service routine called an "interrupt handler"

System Protection

- User programs (programs not belonging to the OS) are generally not trusted
  - A user program may use an unfair amount of resource
  - A user program may maliciously cause other programs or the OS to fail
- Need protection against untrusted user programs; the system must differentiate between at least two modes of operations
  1. *User mode* – execution of user programs
     - untrusted
     - not allowed to have complete/direct access to hardware resources
  2. *Kernel mode* (also *system mode* or *monitor mode*) – execution of the operating system
     - trusted
     - allowed to have complete/direct access to hardware resources
- Hardware support is needed for such protection

Transition between User/Kernel Mode

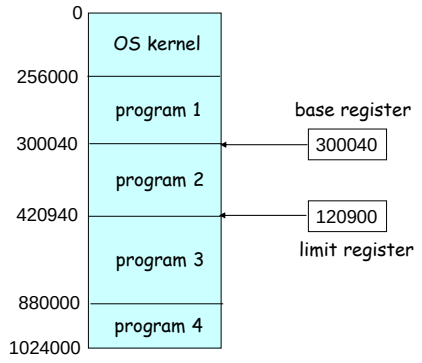
- When does the machine run in kernel mode?
  - after machine boot
  - interrupt handler
  - system call
  - exception



Memory Protection

- Goal of memory protection?
  - A user program can't use arbitrary amount of memory
  - A user program can't access data belonging to the operating system or other user programs
- How to achieve memory protection?
  - Indirect memory access
    - Memory access with a virtual address which needs to be translated into physical address
  - Add two registers that determine the range of legal addresses a program may access:
    - Base register – holds the smallest legal physical memory address
    - Limit register – contains the size of the range
    - Memory outside the defined range is protected

Hardware Address Protection

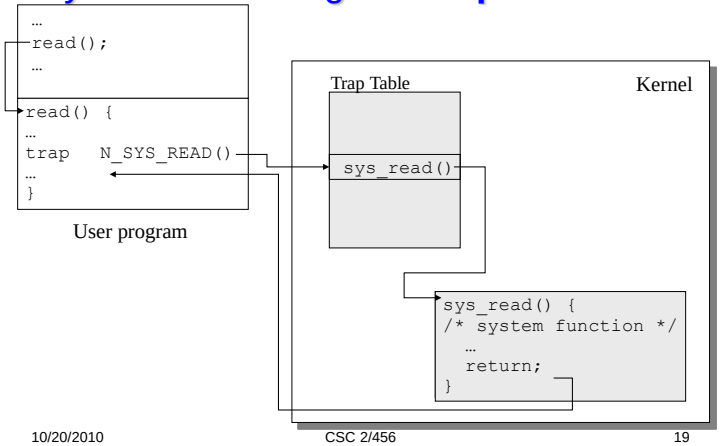


- Address of each memory address is checked against "base" and "base+limit"
- Trap to the OS kernel if it falls outside of the range (an exception)

Protection of I/O Devices

- User programs are not allowed to directly access I/O devices
  - Special I/O instructions can only be used in kernel mode
  - Controller registers can only be accessed in kernel mode
- So device drivers, I/O interrupt handlers must run in kernel mode
- User programs perform I/O through requesting the OS (using system calls)

System Call Using the **Trap** Instruction



CPU Protection

- Goal of CPU protection
  - A user program can't hold the CPU for ever
- *Timer* – interrupts computer after specified period to ensure the OS kernel maintains control
  - Timer is decremented every clock tick
  - When timer reaches the value 0, an interrupt occurs
  - CPU time sharing is implemented in the timer interrupt

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.

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)

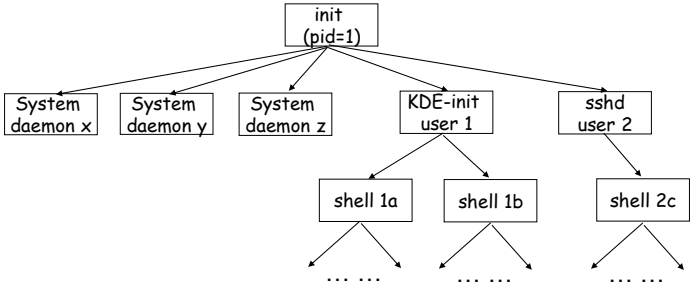
- A signal is *pending* if it has been sent but not yet received
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded
- 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

- 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.

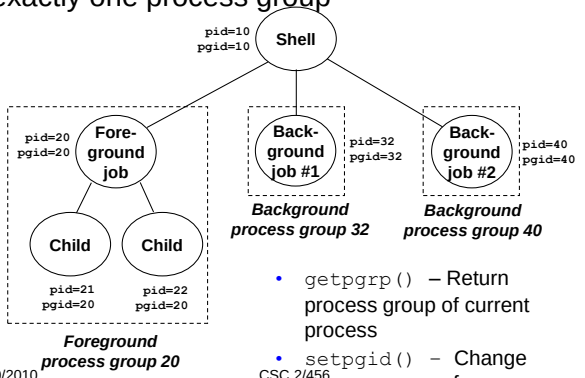
Process Tree on a Linux System

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



Process Groups

- Every process belongs to exactly one process group



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

Sending Signals with `kill` Program

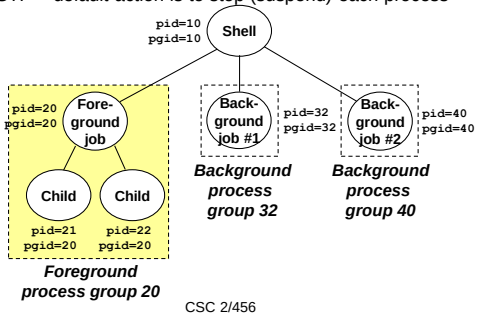
- `kill` program sends arbitrary signal to a process or process group
- Examples
  - `kill -9 24818`
    - Send SIGKILL to process 24818
  - `kill -9 -24817`
    - Send SIGKILL to every process in process group 24817.

```
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>
```


Sending Signals from the Keyboard

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



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.

Some Common Signals and Their Defaults

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

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 the 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.

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 */
  }
}
```

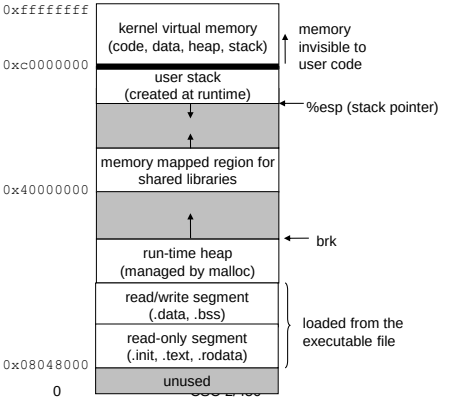
```
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```

Process and Its Image

- An operating system executes a variety of programs:
  - A program that browses the Web
  - A program that serves Web requests
- Process – a program in execution.
- A process's state/image in a computer includes:
  - User-mode address space
  - Kernel data structure
  - Registers (including program counter and stack pointer)
- Address space and memory protection
  - Physical memory is divided into user memory and kernel memory
  - Kernel memory can only be accessed when in the kernel mode
  - Each process has its own exclusive address space in the user-mode memory space (sort-of)

Private Address Spaces

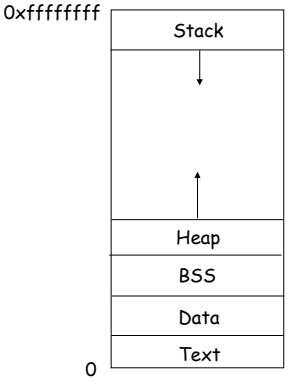
- Each process has its own private address space.



User-mode Address Space

User-mode address space for a process:

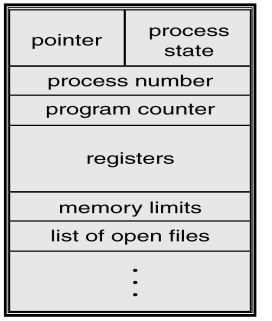
- Text: program code, instructions
- Data: initialized global and static variables (those data whose size is known before the execution)
- BSS (block started by symbol): uninitialized global and static variables
- Heap: dynamic memory (those being malloc-ed)
- Stack: local variables and other stuff for function invocations



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 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

Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, Dave O'Hallaron, Randal Bryant, 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).