Logistics

- Project 0 due on Friday at 5pm
- Assignment 1 posted by COB today
  - Homework exercises
  - Intended to cover material from OSPP and lectures
- Project 1 to be posted on Thursday
Outline

- Process concept
- Process operation
- System calls to create processes
- Process management
- Process scheduling
Overview of Processes

- We have programs, so why do we need processes?
- Questions that we explore
  - How are processes created?
    - from binary program to executing process
  - How is a process represented and managed?
    - process creation, process control block
  - How does the OS manage multiple processes?
    - process state, ownership, scheduling
  - How can processes communicate?
    - interprocess communication, concurrency, deadlock
Superview and User Modes

- OS runs in supervisor mode
  - Has access to protected instructions only available in that mode (ring 0 where the kernel executes)
  - Can manage the entire system
- OS loads processes to run in user mode
  - Many processes can run in user mode
- How does OS get programs loaded into processes in user mode and keep them straight?
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Terms job and process almost used interchangeably
- Process – a *program in execution*
  - Process execution can result in more processes being created
- Multiple parts of a process
  - Program code
  - Current activity (program counter, processor registers, …)
  - Stack containing temporary data (call stack frames)
    - function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time
Process Execution 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 run
  - **Terminated**: The process has finished execution
Process State

- Process state consists of:
  - Address space
  - Execution state
  - Resources being used

- Address space contains code and data of a process

- Processes are individual execution contexts
  - Threads are also include here (we will talk about later)

- Resources are physical support necessary to execute
  - memory
  - disk, files, …
  - processor
Process Address Space

- **Process address space** is all locations addressable by the process
  - AKA the *logical address space*
  - Every running program has its own private address space
- Can restrict use of addresses so as to isolate different areas
  - Restrictions enforced by OS
  - **Text segment** is where read only program instructions are stored
  - **Data segment** hold the data for the running process (read/write)
    - heap allows for dynamic data expansion
  - **Stack segment** is where the stack lives
- Process (logical) address space starts at 0 and runs to a high address
Process Address Space

- Program (Text)
- Global Data (Data)
- Dynamic Data (Heap)
  - Grows up
- Thread-local Data (Stack)
  - Grows down
- Each thread has its own stack
- # address bits determine the addressing range

![Address Space Diagram]

0xFFFFFFF

OS kernel [protected]
stack
shared libraries
heap (malloc/free)
read/write segment
  .data, .bss
read-only segment
  .text, .rodata

0x00000000
Process Address Space

```c
int value = 5;  // Global

int main()
{
    int *p;  // Stack

    p = (int *)malloc(sizeof(int));  // Heap

    if (p == 0) {
        printf("ERROR: Out of memory\n");
        return 1;
    }

    *p = value;
    printf("%d\n", *p);
    free(p);
    return 0;
}
```
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + Stack

#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ... 
    free(ncopy);
    return 0;
}
Heap + Stack

```
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...)
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
```

OS kernel [protected]

<table>
<thead>
<tr>
<th>Stack</th>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc, argv</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

nums

ncopy

read/write segment
globals

heap (malloc/free)

read-only segment
(main, f, g)
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size * sizeof(int));
    if (a2 == NULL) return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + Stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Process Creation

- What happens?
  - New process object in the kernel
    - build process data structures
  - Allocate address space (abstract resource)
    - later, allocate memory (physical resource)
  - Add to execution queue
    - runnable?
Process Creation Options

- Process hierarchy options
  - Parent process create children processes
  - Child processes create other processes
  - Tree of processes

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

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
Process Control Block

- Information associated with each process
  - AKA *task control block*
- Process state: running, waiting, …
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information – priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files
Program Creation System Calls

- `fork()`
  - Copy address space of parent and all threads
- `forkl()`
  - Copy address space of parent and only calling thread
- `vfork()`
  - Do not copy address space
  - Share address space between parent and child
- `exec()`
  - Load new program and replace address space
  - Some resources may be transferred (open file descriptors)
  - Specified by arguments
Process Creation

- `fork()`
- `wait()`
- `exec()`
- `exit()`

Flow:
- Parent calls `fork()` to create a child process.
- Child process calls `exec()` to replace its program with a new one.
- Parent calls `wait()` to suspend its execution and wait for the child to exit.
- Child process calls `exit()` to terminate.
- Parent resumes after waiting.

CIS 415: Operating Systems, University of Oregon, Fall 2015

Lecture 3 – Processes
C Program Forking Separate Process

```c
int main( )
{
    pid_t pid;
    /* fork another process */
    pid = fork( );
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {
        /* child process */
        execlp("/bin/ls", "ls", NULL); /* exec a file */
    }
    else {
        /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```

all execute a file and are frontends to `execve`
### Effects in memory after parent calls `fork()`

1. PCB with new Id created
2. Memory allocated for child initialized by copying over from the parent
3. If parent had called `wait()`, it is moved to a waiting queue
4. If child had called `exec()`, its memory is overwritten with new code and data
5. Child added to ready queue and is all set to go now!
Relocatable Memory

- Program instructions generate addresses that logically start at 0
- Cannot place all programs in memory at physical address 0
- Relocation is the mechanism needed that enables the OS to place a program in an arbitrary location in memory
  - Gives the programmer the impression that they own the processor and the memory
- Program is loaded into memory at program-specific locations
  - Need some form of address translation (relocation) to do this
    - base-limit, segmentation, paging, virtual memory
  - We will talk about this later
- Also, may need to share program code across processes
A tree of processes on a typical system
Process Actions in Client-Server (1)
Process Actions in Client-Server (3)
Process Actions in Client-Server (4)

- Client
- Server
- Server (forked child)

Diagram showing the actions in a client-server interaction with an example of a child process being forked.
Process Actions in Client-Server (5)

Client → Server

- child exit()’s
- parent wait()’s
Process Actions in Client-Server (6)

client → server

server

parent closes its client connection
Process Actions in Client-Server (7)
Process Actions in Client-Server (8)

client → server

server → fork(child)

server → client
Process Actions in Client-Server (9)
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 system do not allow child to continue if parent terminates
    - all children terminated - cascading termination
Executing a Process

- What to execute?
  - Register that stores the program counter
    - next instruction to be executed

- Registers store state of execution in CPU
  - Stack pointer
  - Data registers

- Thread of execution
  - Has its own stack
Executing a Process

- Thread executes over the process’s address space
  - Usually the text segment

- Until a trap or interrupt...
  - Time slice expires (timer interrupt)
  - Another event (e.g., interrupt from other device)
  - Exception (oops)
  - System call (switch to kernel mode)
Let’s walk through how a Linux system call actually works

- Assume 32-bit x86 using the modern \texttt{SYSENTER} / \texttt{SYSEXIT} x86 instructions
Details on x86 / Linux

- Remember our process address space picture
  - Let’s add some details

- C standard library
- POSIX
- glibc
- Linux-gate.so
- Linux kernel
- stack
- shared libraries
- heap (malloc/free)
- read/write segment .data, .bss
- read-only segment .text, .rodata

0xFFFFFFFF

0x00000000
Details on x86 / Linux

0xFFFFFFFF

linux-gate.so

Linux kernel

kernel stack

0xFFFFFFFF

SP

stack

stack

shared libraries

shared libraries

heap (malloc/free)

heap (malloc/free)

read/write segment

read/write segment

.text, .bss

.text, .bss

.read-only segment

.read-only segment

.data, .bss

.data, .bss

IP

your program

your program

C standard library

C standard library

glibc

glibc

POSIX

POSIX

architecture-dependent code

architecture-dependent code

architecture-independent code

architecture-independent code

Linux kernel

unpriv

CPU

process is executing your program code

process is executing your program code

process is executing your program code

process is executing your program code

process is executing your program code

process is executing your program code
glibc begins the process of invoking a Linux system call:

- glibc’s `fopen()` likely invokes Linux’s `open()` system call
- Puts the system call # and arguments into registers
- Uses the call x86 instruction to call into the routine `__kernel_vsyscall` located in `linux-gate.so`
Details on x86 / Linux

- **linux-gate.so** is a *vdso*
  - A virtual dynamically linked shared object
  - Is a kernel-provided shared library
  - It is not associated with a .so file, but rather is conjured up by the kernel and plunked into a process’s address space
  - Provides the intricate machine code needed to trigger a system call

- **IP**
  - 0xFFFFFFFF

- **Stack**
  - Kernel stack

- **Shared libraries**
  - Read/write segment
    - data, bss
  - Read-only segment
    - text, rodata

- **Heap**
  - malloc/free

- **CPU**
  - Unpriv

- **Your program**
  - C standard library
    - POSIX
  - glibc

- **Linux kernel**
  - Architecture-independent code
  - Architecture-dependent code
linux-gate.so eventually invokes the **SYSENTER**
\[x86\] instruction

- **SYSENTER** is x86’s “fast system call” instruction
- It has several side-effects
  - Causes the CPU to raise its privilege level
  - Traps into the Linux kernel by changing the SP, IP to a previously determined location
  - Changes some segmentation related registers

---

**Diagram: Details on x86 / Linux**

- **0xFFFFFFFF**
- **IP**
- **SP**
  - **linux-gate.so**
  - **kernel stack**
  - **stack**
  - **shared libraries**
  - **heap (malloc/free)**
  - **read/write segment**
    - `.data, .bss`
  - **read-only segment**
    - `.text, .rodata`

---

**Architecture:**

- **POSIX**
- **C standard library**
- **glibc**
- **architecture-independent code**
- **architecture-dependent code**
The kernel begins executing code at the `SYSENTER` entry point

- Is in the architecture-dependent part of Linux
- Its job is to:
  - Look up the system call number in a system call dispatch table
  - Call into the address stored in that table entry; this is Linux’s system call handler
  - For `open()`, the handler is named `sys_open`, and is system call #5
Details on x86 / Linux

The system call handler executes:

- What it does is system-call specific, of course
- It may take a long time to execute, especially if it has to interact with hardware
  - Linux may choose to context switch the CPU to a different runnable process

CIS 415: Operating Systems, University of Oregon, Fall 2015
Lecture 3 – Processes
Eventually, the system call handler finishes

- Returns back to the system call entry point
  - Places the system call’s return value in the appropriate register
  - Calls SYSEXIT to return to the user-level code
SYSEXIT transitions the processor back to user-mode code

- Has several side-effects
  - Restores the IP, SP to user-land values
  - Sets the CPU back to unprivileged mode
  - Changes some segmentation related registers
- Returns the processor back to glibc
Details on x86 / Linux

glibc continues to execute

- Might execute more system calls
- Eventually returns back to your program code
Process State

- What do we need to track about a process?
  - How many processes?
  - What’s the state of each of them?

- Process table
  - Kernel data structure tracking processes on system

- Process control block
  - Structure for tracking process context
Scheduling Processes

- Processes transition among execution states
Process States

- Running
  - Process is executing in the processor and in memory with all resources to run

- Ready
  - Process in memory with all resources to run, but is waiting for dispatch onto a processor

- Waiting
  - Process is not running, instead waiting for some event to occur
State Transitions

- **New Process ==> Ready**
  - Allocate resources necessary to run
  - Place process on process queue (usually at end)

- **Ready ==> Running**
  - Process is at the head of process queue
  - Process is scheduled onto an available processor

- **Running ==> Ready**
  - Process is interrupted
    - usually by a timer interrupt
  - Process could still run, in that it is not waiting something
  - Placed back on the process queue
State Transitions: Page Fault Handling

- **Running ==> Waiting**
  - Either something exceptional happened that caused an interrupt to occur (e.g., page fault exception) …
  - … or the process needs to wait on some action (e.g., it made a system call or requested I/O)
  - Process must wait for whatever event happened to be serviced

- **Waiting ==> Ready**
  - Event has been satisfied so that the process can return to run
  - Put it on the process queue

- **Ready ==> Running**
  - As before…
State Transitions: Other Issues

- Priorities
  - Can provide policy indicating which process should run next
    - more when we discuss scheduling…

- Yield
  - System call to give up processor voluntarily
  - For a specific amount of time (sleep)

- Exit
  - Terminating signal (Ctrl-C)
Process Control Block

- State of running process
- Linked list of process control information
Per Process Control Info

- Process state
  - Ready, running, waiting (momentarily)
- Links to other processes
  - Children
- Memory Management
  - Segments and page tables
- Resources
  - Open files
- And much more…
/proc File System

- Linux and Solaris
  - `ls /proc`
  - Process information pseudo-file system
  - Does not contain “real” files, but runtime system information
    - System memory
    - Devices mounted
    - Hardware configuration
  - A directory for each process

- Various process information
  - `/proc/<pid>/io`
    I/O statistics
  - `/proc/<pid>/environ`
    Environment variables (in binary)
  - `/proc/<pid>/stat`
    Process status and info
Context Switch

- OS switches from one execution context to another
  - One process to another process
  - Interrupt handling
  - Process to kernel (mode transition, not context switch)

- Current process to new process
  - Save the state of the current process
    - process control block: describes the state of the process in the CPU
  - Load the saved context for the new process
    - load the new process’s process control block into OS and registers
  - Start the new process

- Does this differ if we are running an interrupt handler?
Context Switch

Diagram showing the process of context switch between two processes, $P_0$ and $P_1$. The diagram illustrates the steps involved in saving the state of $P_0$ into its process control block (PCB), loading the state of $P_1$ from its PCB, and executing $P_1$. The process is reversed when the state of $P_1$ needs to be saved and $P_0$ needs to be executed.
Context Switch Performance

- No useful work is being done during a context switch
  - Speed it up and limit system calls to things that can’t be done in user mode

- Hardware support
  - Multiple hardware register sets (Sun UltraSPARC)
  - Be able to quickly set up the processor

- However, hardware optimization may conflict
  - TLB flush is necessary
  - Different virtual to physical mappings on different processes
Process Description Summary

- Serves two purposes
  - Track per process resources
  - Save CPU state on context switch

- Process control block
  - Represents both aspects
  - CPU state
    - program counter, registers
  - Resources
    - linked lists of pages, child processes, files, and so on
Program to Process

- Program is stored in a binary format
  - Executable and Linkable Format (ELF)
  - a.out

- Binary format describes
  - Program sections
    - Text, Data, … (many types of sections)
  - Program segments
    - what to load at execution time
    - one or more per section
ELF Files

- Source code
  - `test.c`

- Compile into an ELF relocatable file
  - `test.o` (object file)

- Compile into an ELF shared object file
  - `gcc -shared >> test.so` (from `.o` files)

- Compile into an ELF executable file
  - `gcc -o test test.c`
**ELF Files**

- ELF executable file contains segments
  - Describes how to load them in memory
- ELF executable file also references any shared object files used
  - Dynamically linked
Load and Run ELF Binaries

- Program Interpreter is loaded first
  - Guides the loading process by interpreting ELF binaries
  - Segment type PT_INTERP
  - Run by exec

- Interpreter loads Loadable Segments
  - Contains the program contents: text, global data
  - Segment type PT_LOAD
  - Mapped into the process address space at loadtime (you see these for libraries only)

- Others are loaded on demand, Dynamic Segment
  - Libraries
  - Segment type PT_DYNAMIC
  - Load of separate library files when needed (you see these in opening of lib files)
ELF Binary View

- **Commands**
  - Linux: `readelf`
  - Solaris: `elfdump`

Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>FileSiz</th>
<th>MemSiz</th>
<th>Flg</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHDR</td>
<td>0x000034</td>
<td>0x08048034</td>
<td>0x08048034</td>
<td>0x000e0</td>
<td>0x000e0</td>
<td>R E</td>
<td>0x4</td>
</tr>
<tr>
<td>INTERP</td>
<td>0x000114</td>
<td>0x08048114</td>
<td>0x08048114</td>
<td>0x00013</td>
<td>0x00013</td>
<td>R</td>
<td>0x1</td>
</tr>
</tbody>
</table>
  
  [Requesting program interpreter: `/lib/ld-linux.so.2`]

| LOAD      | 0x000000 | 0x08048000 | 0x08048000 | 0x016b8 | 0x016b8 | R E | 0x1000|
| LOAD      | 0x0016b8 | 0x0804a6b8 | 0x0804a6b8 | 0x00120 | 0x00120 | RW  | 0x1000|
| DYNAMIC   | 0x0016cc | 0x0804a6cc | 0x0804a6cc | 0x000d0 | 0x000d0 | RW  | 0x4   |
| NOTE      | 0x000128 | 0x08048128 | 0x08048128 | 0x00020 | 0x00020 | R   | 0x4   |
| GNU_STACK | 0x000000 | 0x00000000 | 0x00000000 | 0x00000 | 0x00000 | RW  | 0x4   |

... 

Dynamic section at offset 0x16cc contains 21 entries:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Name/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000001 (NEEDED)</td>
<td>Shared library: [libm.so.6]</td>
<td></td>
</tr>
<tr>
<td>0x00000001 (NEEDED)</td>
<td>Shared library: [libc.so.6]</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Linking

- Global Offset Table (GOT)
  - Access to symbol in GOT results in dynamic loading and linking of associated library

- Program calls printf in libc
  - Symbol points to dynamic linker at loadtime
  - Loads libc library
  - Fixes GOT pointer for printf to actual libc function

- Results in a level of indirection for calling library functions
  - Slight performance cost
Process Scheduling

- Maximize CPU use, 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
Ready Queue And Various I/O Device Queues
Representation of Process Scheduling

- Process scheduling queueing diagram represents:
  - queues, resources, flows
- Processes move through the queues
Next Class

- More process scheduling
- Interprocess communication (IPC)