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 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)
  - Can manage the entire system
- OS loads processes into 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

- Textbook uses the terms job and process almost interchangeably

- Process – a program in execution
  - Process execution can result in more processes being created

- Multiple parts
  - Program code
  - Current activity (program counter, processor registers, …)
  - Stack containing temporary data
    - function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time
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 be assigned to a processor
- **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 included here (we will talk about later)

- Resources are physical support necessary to run the process
  - Memory, disk, files, …
Process Address Space

- Process address is all locations that are addressable by process
- Every running program has its own private address space
- Can restrict use of addresses so as to isolate different area
  - 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 Address Space

- Program (Text)
- Global Data (Data)
- Dynamic Data (Heap)
- Thread-local Data (Stack)
- Each thread has its own stack

Diagram:
- 0xFFFFFFFF
- 0x00000000
- OS kernel [protected]
- stack
- shared libraries
- heap (malloc/free)
- read/write segment .data, .bss
- read-only segment .text, .rodata
Process Address Space

```c
int value = 5;

int main()
{
    int *p;

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

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

    *p = value;
    printf("%d\n", *p);
    free(p);
    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;
}
### 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;
}
```
### 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;
}
```

<table>
<thead>
<tr>
<th>OS kernel [protected]</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
</tr>
<tr>
<td>main</td>
</tr>
<tr>
<td>argc, argv</td>
</tr>
<tr>
<td>nums</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>malloc</td>
</tr>
<tr>
<td>heap (malloc/free)</td>
</tr>
<tr>
<td>read/write segment</td>
</tr>
<tr>
<td>globals</td>
</tr>
<tr>
<td>read-only segment</td>
</tr>
<tr>
<td>(main, f, g)</td>
</tr>
</tbody>
</table>
#include <stdlib.h>

```c
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(...)
```
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;
}
```

OS kernel [protected]

stack

main

argc, argv

nums

4 6 8

ncopy

read/write segment

globals

heap (malloc/free)

read-only segment

(main, f, g)
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;
}
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;
}
```

OS kernel [protected]

- stack
- main
- argc, argv
- nums
- ncopy
- heap (malloc/free)
- read/write segment
- globals
- 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;
}
```
#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;
}
Process Creation

What happens?

- New process object in 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
  (also called 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()
- exec()
- exit()
- wait
- parent
- resumes

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 */
        execvp("/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);
    }
}
```
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 overwritten with new code & data
5. Child added to ready queue,
   all set to go now!
Relocatable Memory

- Mechanism that enables the OS to place a program in an arbitrary location in memory
  - Gives the programmer the impression that they own the processor
- Program is loaded into memory at program-specific locations
  - Need virtual memory to do this
- Also, may need to share program code across processes
A tree of processes on a typical system

- init (pid = 1)
  - inetd (pid = 140)
    - telnetdaemon (pid = 7776)
      - Csh (pid = 7778)
        - Netscape (pid = 7785)
        - emacs (pid = 8105)
  - dtlogin (pid = 251)
    - Xsession (pid = 294)
      - sdт_shel (pid = 340)
      - Csh (pid = 1400)
        - ls (pid = 2123)
        - cat (pid = 2536)
- pageout (pid = 2)
- fsflush (pid = 3)
Graphically
Graphically

client

server
Graphically

client

server
Graphically

- Client
- Server
- Child
- Fork()
Graphically

client → server

child exit()’s / parent wait()’s

server
Graphically

client -> server

parent closes its client connection
Graphically

client -> server

fork() child

server -> server

client
Graphically
Graphically
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 SYSENTER / SYSEXIT x86 instructions
Details on x86 / Linux

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

- Your program
  - C standard library
  - POSIX
    - glibc

- Linux kernel
  - architecture-independent code
  - architecture-dependent code

- CPU

- 0xFFFFF000
- 0x00000000

Lecture 3 – Processes
Details on x86 / Linux

- Processes
  - CPU
  - Linux
    - Linux kernel
      - Stack
      - Shared libraries
      - Heap (malloc/free)
      - Read/write segment: `.data`, `.bss`
      - Read-only segment: `.text`, `.rodata`
    - Unpriv
  - IP
  - SP

- Your program
  - C standard library: glibc
  - POSIX architecture-dependent code
    - Architecture-independent code

- Unpriv
  - CPU
**Details on x86 / Linux**

- 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

```
0xFFFFF000
IP
```

```
linux-gate.so
0x00000000
kernel stack
```

- Shared libraries
- Heap (malloc/free)
- Read/write segment .data, .bss
- Read-only segment .text, .rodata

```
your program
C standard library
POSIX
```

```
glibc
```

```
architecture-independent code
```

```
architecture-dependent code
Linux kernel
```

```
unpriv
CPU
```

```
0x00000000
```

CIS 415: Operating Systems, University of Oregon, Fall 2014
Details on x86 / Linux

- linux-gate.so is a **vdso**
  - A virtual dynamically linked shared object
  - Is a kernel-provided shared library, i.e., 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
Details on x86 / Linux

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

```
0xffffffff

<table>
<thead>
<tr>
<th>IP</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux kernel</td>
<td>kernel stack</td>
</tr>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>shared libraries</td>
<td></td>
</tr>
<tr>
<td>heap (malloc/free)</td>
<td></td>
</tr>
<tr>
<td>read/write segment</td>
<td>.data, .bss</td>
</tr>
<tr>
<td>read-only segment</td>
<td>.text, .rodata</td>
</tr>
</tbody>
</table>

0x00000000
```
Details on x86 / Linux

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
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
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

- Linux kernel stack
- Shared libraries
- Stack
- Heap (malloc/free)
- Read/write segment
  - .data, .bss
- Read-only segment
  - .text, .rodata

- Linux-gate.so

- CPU
- Privileged
- Unprivileged

- glibc
- C standard library
- POSIX architecture-dependent code
- Architecture-independent code

- Your program
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
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

Diagram:
- States: new, admitted, interrupt, exit, terminated, ready, running, waiting
- Transitions: 
  - I/O or event completion
  - Scheduler dispatch
  - I/O or event wait
Process States

- Running
  - Running == in processor and in memory with all resources

- Ready
  - Ready == in memory with all resources, waiting for dispatch

- Waiting
  - Waiting == waiting for some event to occur

![Process States Diagram]

CIS 415: Operating Systems, University of Oregon, Fall2014

Lecture 3 – Processes
State Transitions

- New Process $\Rightarrow$ Ready
  - Allocate resources
  - End of process queue

- Ready $\Rightarrow$ Running
  - Head of process queue
  - Scheduled

- Running $\Rightarrow$ Ready
  - Interrupt (Timer)
  - Back to end of process queue
State Transitions: Page Fault Handling

- Running ==> Waiting
  - Example: Page fault exception
  - Similar for syscall or I/O interrupt
  - Wait for event

- Waiting ==> Ready
  - Event has occurred (page fault serviced)
  - End of process queue (or head?)

- 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
  - 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`
  - 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

process $P_0$  operating system  process $P_1$

executing  interrupt or system call

save state into PCB$_0$

...  idle  executing

reload state from PCB$_1$

executing  interrupt or system call

save state into PCB$_1$

...  idle  idle

reload state from PCB$_0$
Context SwitchPerformance

- 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, etc.
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</td>
<td>E</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`]

<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>LOAD</td>
<td>0x000000</td>
<td>0x08048000</td>
<td>0x08048000</td>
<td>0x016b8</td>
<td>0x016b8</td>
<td>R</td>
<td>E</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x0016b8</td>
<td>0x0804a6b8</td>
<td>0x0804a6b8</td>
<td>0x00120</td>
<td>0x00120</td>
<td>RW</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

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</td>
<td>(NEEDED)</td>
<td>Shared library: [libm.so.6]</td>
</tr>
<tr>
<td>0x00000001</td>
<td>(NEEDED)</td>
<td>Shared library: [libc.so.6]</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)