• Last class:
  ‣ Operating system structure and basics

• Today:
  ‣ Process Management
Why Processes?

- We have programs, so why do we need processes?
Overview

- 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
Supervisor 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

- Address space + threads + resources
- Address space contains code and data of a process
- Threads are individual execution contexts
- Resources are physical support necessary to run the process (memory, disk, …)
Process Address Space

- Program (Text)
- Global Data (Data)
- Dynamic Data (Heap)
- Thread-local Data (Stack)
- Each thread has its own stack
```c
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;
}
```
Process Creation

- Parent process create children processes,
  - which, in turn create other processes, forming a 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
Process Creation

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

- UNIX examples
  - `fork` system call creates new process
  - `exec` system call used after a fork to replace the process’s memory space with a new program
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!
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

fork() ➔ wait

parent ➔ resumes

child ➔ exec()

exec() ➔ exit()
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);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
Program Creation

- Design Choices
  - Resource Sharing
    - What resources of parent should the child share?
    - What about after exec?
  - Execution
    - Should parent wait for child?
  - What is the relationship between parent and child?
    - Hierarchical or grouped or …?
Program Creation

- **fork** -- copy address space and all threads
- **fork1** -- copy address space and only calling thread
- **vfork** -- do not copy address space; shared between parent and child
- **exec** -- load new program; replace address space
  - Some resources may be transferred (open file descriptors)
  - Specified by arguments
A tree of processes on a typical system
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?
  - Program status word
  - 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)
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
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</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
Summary

- **Process**
  - Execution state of a program

- **Process Structure**
  - Address Space

- **Process Creation**
  - From binary representation
  - Dynamic Linking

- **Process Creation**
  - From other processes
  - Issues
Next time: Process Scheduling & IPC