Lecture 2: The Process Abstraction

Operating Systems

Content taken from: https://www.cse.iitb.ac.in/~mythili/os/

What does an OS do?

- Manages resources
  - Manages CPU
  - Manages memory
  - Manages I/O devices
- Provides support for building correct concurrent programs
- Manages persistent data

Design goals of an operating system

- Convenience, abstraction of hardware resources for user programs

- Efficiency of usage of CPU, memory, etc.

- Isolation between multiple processes

History of Operating Systems

- Started out as a library to provide common functionality across programs
- Later, evolved from procedure call to system call: what’s the difference?
- When a system call is made to run OS code, the CPU executes at a higher privilege level
- Evolved from running a single program to multiple processes concurrently

OS provides process abstraction

- When you run an exe file, the OS creates a process = a running program
- OS timeshares CPU across multiple processes: virtualizes CPU
- OS has a CPU scheduler that picks one of the many active processes to execute on a CPU
  - Policy: which process to run
  - Mechanism: how to “context switch” between processes

What constitutes a process?

- A unique identifier (PID)
- Memory image
  - Code & data (static)
  - Stack and heap (dynamic)
- CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer
- File descriptors
  - Pointers to open files and devices

How does OS create a process?

- Allocates memory and creates memory image
  - Loads code, data from disk exe
  - Creates runtime stack, heap
- Opens basic files
  - STD IN, OUT, ERR
- Initializes CPU registers
  - PC points to first instruction

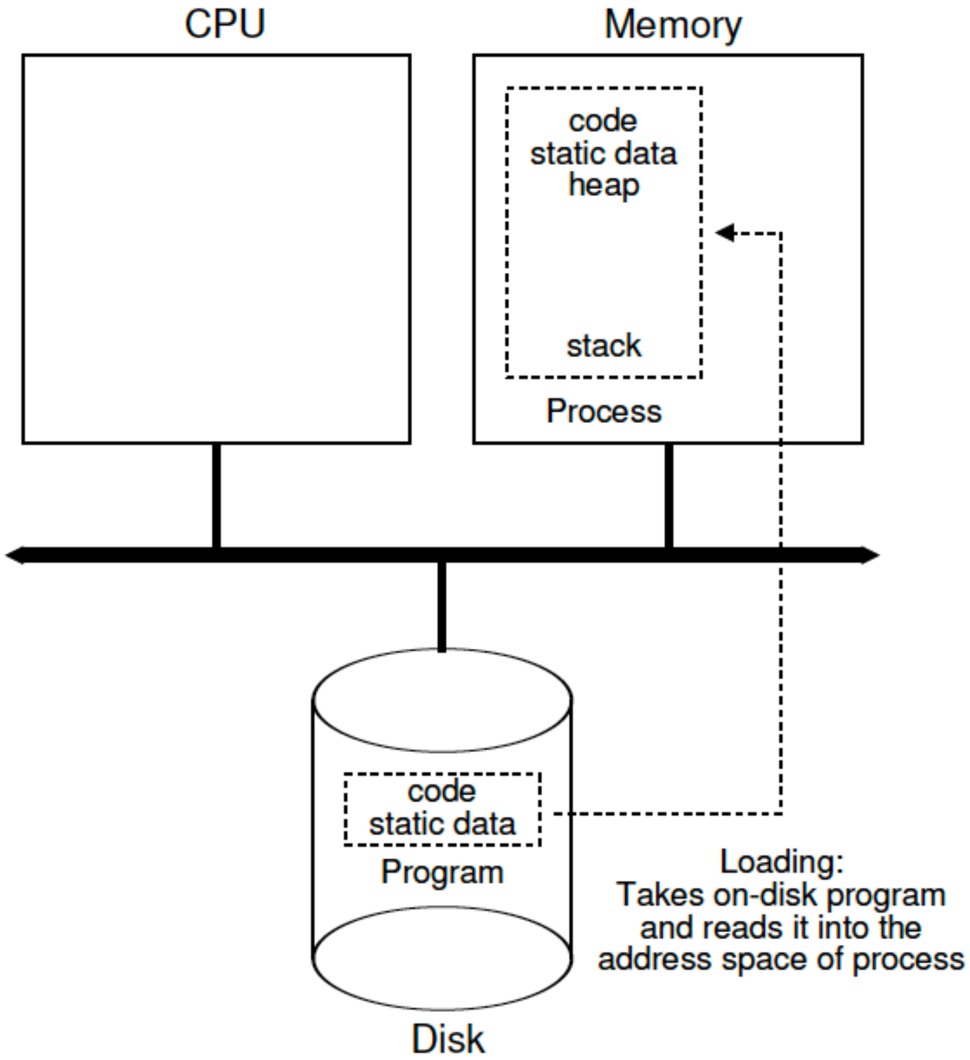

Figure 4.1: **Loading: From Program To Process**

States of a process

- Running: currently executing on CPU
- Ready: waiting to be scheduled
- Blocked: suspended, not ready to run
  - Why? Waiting for some event, e.g., process issues a read from disk
  - When is it unblocked? Disk issues an interrupt when data is ready
- New: being created, yet to run
- Dead: terminated

Process State Transitions

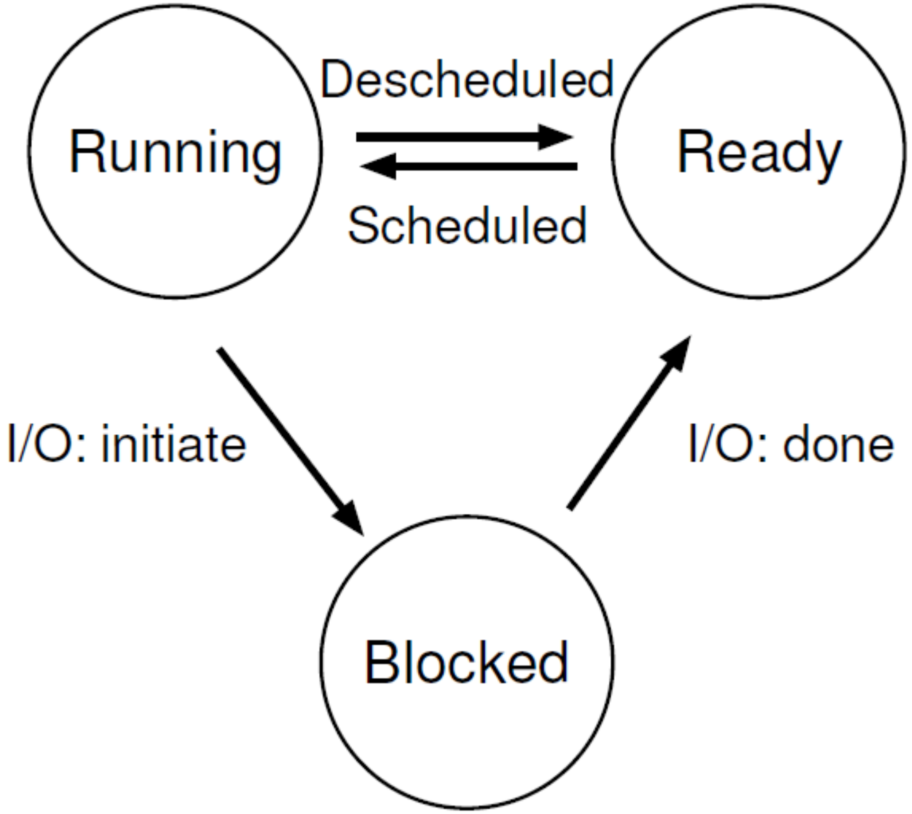

Figure 4.2: **Process: State Transitions**

Example: Process States

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

OS data structures

- OS maintains a data structure (e.g., list) of all active processes
- Information about each process is stored in a process control block (PCB)
  - Process identifier
  - Process state
  - Pointers to other related processes (parent)
  - CPU context of the process (saved when the process is suspended)
  - Pointers to memory locations
  - Pointers to open files

What API does the OS provide to user programs?

- API = Application Programming Interface

  = functions available to write user programs
- API provided by OS is a set of “system calls”
  - System call is a function call into OS code that runs at a higher privilege level of the CPU
  - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
  - Some “blocking” system calls cause the process to be blocked and descheduled (e.g., `read` from disk)

So, should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
  - Programs written to the POSIX API can run on any POSIX compliant OS
  - Most modern OSes are POSIX compliant
  - Ensures program portability
- Program language libraries hide the details of invoking system calls
  - The printf function in the C library calls the `write` system call to write to screen
  - User programs usually do not need to worry about invoking system calls

Process related system calls (in Unix)

- `fork()` creates a new child process
  - All processes are created by forking from a parent
  - The `init` process is ancestor of all processes
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

What happens during a fork?

- A new process is created by making a copy of parent’s memory image
- The new process is added to the OS process list and scheduled
- Parent and child start execution just after fork (with different return values)
- Parent and child execute and modify the memory data independently

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {         // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
                rc, (int) getpid());
    }
    return 0;
}
```

Figure 5.1: **Calling `fork()` (`p1.c`)**

When you run this program (called `p1.c`), you'll see the following:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

Waiting for children to die…

- Process termination scenarios
  - By calling `exit()` (exit is called automatically when end of main is reached)
  - OS terminates a misbehaving process
- Terminated process exists as a zombie
- When a parent calls `wait()`, zombie child is cleaned up or “reaped”
- `wait()` blocks in parent until child terminates (non-blocking ways to invoke wait exist)
- What if parent terminates before child? `init` process adopts orphans and reaps them

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

What happens during exec?

- After fork, parent and child are running same code
  - Not too useful!
- A process can run `exec()` to load another executable to its memory image
  - So, a child can run a different program from parent
- Variants of `exec()`, e.g., to pass command line arguments to new executable

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {         // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

Figure 5.3: Calling **fork()**, **wait()**, And **exec()** (**p3.c**)

Case study: How does a shell work?

- In a basic OS, the `init` process is created after initialization of hardware
- The `init` process spawns a shell like `bash`
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like `ls` are all executables that are simply exec'ed by the shell

```
prompt>ls
a.txt b.txt c.txt
```

More funky things about the shell

- Shell can manipulate the child in strange ways
- Suppose you want to redirect output from a command to a file
- `prompt>ls > foo.txt`
- Shell spawns a child, rewires its standard output to a file, then calls exec on the child

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {                // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (p4.c)**

Here is the output of running the p4.c program:

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
```