The Process Abstraction

CMPU 334 – Operating Systems
Jason Waterman
How to Provide the Illusion of Many CPUs?

• Goal: run N processes at once even though there are M CPUs
  - N >> M

• CPU virtualizing
  - The OS can promote the illusion that many virtual CPUs exist
  - One isolated machine for each program

• Time sharing
  - Running one program, then stopping it and running another
  - The potential cost is performance

• What are the benefits?
  - Ease of use for the programmer
  - Protection – program runs on a restricted machine
A Process

• A process is OS’s abstraction of a **running program**

• What constitutes a process?
  • Memory (address space)
    • Instructions
    • Data
  • Registers (state of the processor)
    • General purpose registers
    • Program counter (PC)
    • Stack pointer (SP)
  • I/O Information
    • List of files process currently has open
Process API

- These APIs are available on any modern OS
  - Create
    - Create a new process to run a program
  - Destroy
    - Halt a runaway process
  - Wait
    - Wait for a process to stop running
  - Miscellaneous Control
    - Suspend
    - Resume
  - Status
    - Get some status information about a process
    - How long it has been running
    - What state is it in
Process Creation

1. Load a program code into memory, the address space of the process
   • Programs reside on a disk in an **executable format**
   • OS performs the loading process **lazily**
     • Loads pieces of code or data only as they are needed during program execution (demand paging)

2. The program’s **run-time stack** is allocated
   • Stack is used for local variables, function parameters, return address
   • Initialize the stack with arguments
     • argc and argv array of main() function
Process Creation (Cont.)

3. The program’s **heap** is created
   - Used for explicitly requested dynamically allocated data
   - `malloc(); free()`

4. The OS does some other **initialization**
   - I/O setup (stdin, stdout, stderr)

5. **Start** the program running at the entry point `main()`
   - The OS transfers control of the CPU to the newly-created process
Process States (simplified)

• A process can be in one of three states
  • **Running**
    • A process is running on the CPU
  • **Ready**
    • A process is ready to run but for some reason the OS has chosen not to run it at this given moment
  • **Blocked**
    • A process has performed some kind of operation that it is waiting on
    • E.g., an disk request
Process Data Structures

• The OS has some key data structures that track various pieces of information
  • Process list
    • Ready processes
    • Blocked processes
    • Current running process
  • Register context
    • A copy of all the registers for a process
• The Process Control Block (PCB)
  • A C-structure that contains information about each process
The xv6 Kernel Process Structures

// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;       // Index pointer register
    int esp;       // Stack pointer register
    int ebx;       // Called the base register
    int ecx;       // Called the counter register
    int edx;       // Called the data register
    int esi;       // Source index register
    int edi;       // Destination index register
    int ebp;       // Stack base pointer register
};

// the different states a process can be in
enum procstate { UNUSED, EMBRYO, SLEEPING,
    Runnable, Running, Zombie };
The xv6 Kernel Process Structures (Cont.)

// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
};
Process Creation

• We talked about process creation in general terms
• Now let’s discuss process creation in UNIX systems
  • fork() – Makes a copy of the currently running process
  • exec() – Replaces a process with a different program
  • wait() – Wait for a child process to finish

• Questions to think about
  • What interfaces should the OS present for process creation and control?
  • How should these interfaces be designed to enable ease of use as well as utility?
The fork() System Call

• Create a new process
  • The newly-created process has its own copy of the address space, registers, and PC.

```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;
}
```
Calling fork() example (Cont.)

Result (Not deterministic)

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

or

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

- This system call won’t return until the child has run and exited

```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;
}
```
The wait() System Call (Cont.)

Result (Deterministic)

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```
The exec() System Call

• Run a program that is different from the calling program

```c
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;
}
```

Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```
Motivating the API

• Why the odd interface for the simple act of creating a new process?
• Why is fork() and exec() a separate functions?
• Necessary for building a UNIX shell
  • Let’s the shell run code *after* the call to fork() but *before* the call to exec()
  • Can alter the environment of the about to be run program
  • Can easily support things like redirection and pipes
All of the above with redirection

p4.c

```c
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;
}
```

Result

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