Operating Systems (coe628) Lecture Notes—Week 3

Table of Contents

Topics

- Review
- Process management
- Process creation with fork()
- Overlaying an existing process with exec
- Notes on Lab 3

Review

- An OS manages resources: CPU, memory, devices (and “files”).
- Managing the CPU in a multi-tasking OS involves *threads* and *processes.*
- A process has its own memory spaces; processes do not *usually* communicate (coordinate) with each other.
- A process may create threads that belong to the process.

- Threads share the same global memory spaces of its parent process (but each thread has its own stack.)
- Threads often communicate/coordinate with each other.
- The memory spaces managed by the OS for each process include:
  - The (read-only) *text* segment for the machine language of the process.
  - A *data* segment for global variables.
  - A *stack* segment for the process's stack.
  - A *heap* segment for dynamically allocated memory.
  - A *read-only data* segment for globals that cannot be modifies.

Process management

A core task of a multitasking operating system is to manage the *processes*.

(We assume here that the CPU has a single core: i.e. only one process can actually be running at any instant in time. A multitasking operating system creates the "illusion" that many processes are running simultaneously.)

1. The Process Table

The OS kernel maintains a table containing information about all of the processes that wish to use the CPU. The information associated with each processes includes:

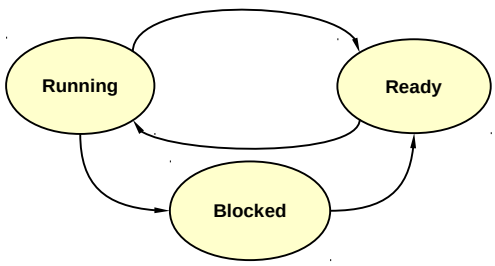
- The values of all the CPU registers especially:
  - The Program Counter (PC or IP) containing the address of the instruction that the process should execute when it runs.
  - The Stack Pointer (SP) containing the address of the top of the hardware maintained stack.
  - The Program Status Register (CC or CCR or PSW) containing bit flags used by conditional jump instructions and to enable/disable recognition of interrupts.
- Pointers to the memory areas used by the process including the "text" segment where the machine language instructions are stored, the Stack area, the global data area and the Heap area (used for dynamically allocated memory)>
- The Process ID (PID) number. (Each process has a unique PID.)
- The PID of the process's parent (PPID). (Each process—except for "init", the very first process—is created by another process: its parent.)
- The process's priority. (When more than one process is ready to use the CPU the kernel *scheduler* selects the one with the highest priority. In general purpose operating systems (such as Windows, OSX, Unix/Linux), the priorities are adjusted dynamically to ensure "fairness". In real-time operating systems, the priorities are usually fixed.

- Statistics such as the amount of CPU time actually used; when the process was created.
- The *state* of the process (to be discussed next).

2. Process States

- Only one process can run (use the CPU) at any instant in time (assuming a single-core CPU).
- That process is in the *Running* state.
- Other processes may be completely ready to run but not currently be in the Running state. These processes are in the *Ready* state.
- Other processes may not be able to run because they are waiting for some event occur. These processes are in the *Blocked* state.
- The type of event a process may be waiting for include:
    - User activity (keyboard or mouse or touch screen).
    - Input/output completion (such as reading or writing the disk drive)
    - Local area network or Internet activity.
    - A timer event.
    - Other kinds of "abstract" events (such as waiting on a *semaphore* or other inter-process communication activity that we will examine in detail later on in the course.)
- The possible state transitions are:
    - From **Ready→Running**: When another ready process replaces the one that was running.
    - From **Running→Ready**: When the running process is replaced by another but is still ready to run
    - From **Running→Blocked**: When the current running process has to wait for some external event.
    - From **Blocked→Ready**: When a process that is blocked waiting for an event is notified of the event.

The diagram below summarizes the kind of state transitions that can occur.



3. Process creation and termination

Different OSes created processes in various ways.

- **Unix (and Posix)**: To create a new process, the *fork()* system call is used to create a copy of the parent process. The child process then replaces itself with another process with the *exec()* system call.
- **Windows**: A process is created in a single step with *CreateProcess(...)* system call.
- **Linux (Android)**: While Linux supports the fork/exec model, it also has a more general form of "fork()" called "clone(....)" which takes numerous parameters identifying how much of the parent is copied to the child. At one extreme, it creates a new process (like "fork"); at the other extreme, it creates a *thread* or "anything in between". (We shall examine this later once we have experience with both threads and processes.)

4. Inter-process Communication

Process creation with Fork()

- In POSIX compliant operating systems, a new processes can be created with the `fork()` system call.
- The process that invoked fork is called the *parent* and the newly created "forked" process is called the *child.*
- `fork()` returns to both parent and child. However, its return value is 0 (zero) to the child and the new process ID number (PID) is returned to the parent. Consequently, looking at the return value allows the child to identify itself.
- Other than that, the child is "born" with all the knowledge of its parent. New data sections for global variables and a heap are created and a new stack is created for the child. However, the initial contents of these new memory spaces contain the identical content for both parent and child.
- The following illustrates this:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int global = 5;

int main(int argc, char** argv) {
    int local = 5;
    int i;

    if (fork() == 0) {
        for (i = 0; i < 10; i++) {
            local++;
```

```
            global++;
            printf("CHILD local: %d, global: %d\n", local, global);
            usleep(500);
        }
        exit(EXIT_SUCCESS);
    }
    for (i = 0; i < 10; i++) {
            local--;
            global--;
            printf("PARENT local: %d, global: %d\n", local, global);
            usleep(500);
        }

    return (EXIT_SUCCESS);
}
```

1. Replacing a process with exec

- A running process can replace itself with another executable stored in an executable file. This does *not* change the process ID (i.e. the process will do something else but its position in the process table is not changed).
- To "exec" a different program, the kernel reads the executable file from desk placing the mahine language instructions into a "text" memory segment, allocating and initializing a data segment as well as memory for the stack and heap.
- There are several variants of "exec" defined in the POSIX standard. Some common variants are:
    - `execl(`*`path, name, arg1, arg2...,`* `(char *) 0)`: "path" must be the absolute path name of the file.
    - `execlp(commandName,` *`name, arg1, arg2...,`* `(char *) 0)`: With the "p" suffix search for the command from the list of directories in the *PATH* environment variable.
    - `execvp(`*`commandName, (char **) argv`*`)`: While the "l" suffix stands for "list" and the arguments to the command are given as separate parameters, the "v" suffix indicates that the command arguments are in an array of strings (i.e. an array of character pointers). The size of the array is not given; rather, the last argument must be *(char *) 0* (NULL).

2. The child "forks"; what does the parent do?

- The parent may continue concurrently with the child or, alternatively, the parent may wait until the child has completed.
- The parent can wait by invoking the *wait* system call when fork() returns.
- The examples below show how to do this.

Fork/Exec examples

Example using fork/execl

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

int main(int argc, char** argv) {
    pid_t pid;
    int status;
    if ((pid = fork()) == 0) {
        //Child runs this
        execl("/usr/bin/date","date",0);
        fprintf(stderr, "Should NEVER get here!\n");
        exit(1);
    }

   //Parent continues here
    /*Comment out the following line for the parent
     * to run concurrently with the child.  (i.e.
     * to make the parent run in the "background") */
    wait(&status);
    printf("Hello from parent with child %d\n", pid);
    return (EXIT_SUCCESS);
}
```

1. Example using fork/execlp

```
#include <stdio.h>
```

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

int main(int argc, char** argv) {
    pid_t pid;
    int status;
    if ((pid = fork()) == 0) {
        //Child runs this
        execlp("ls","ls","-l", "-t", 0);
        fprintf(stderr, "Should NEVER get here!\n");
        exit(1);
    }

   //Parent continues here
    /*Comment out the following line for the parent
     * to run concurrently with the child.  (i.e.
     * to make the parent run in the "background") */
    wait(&status);
    printf("Hello from parent with child %d\n", pid);
    return (EXIT_SUCCESS);
}
```

2. Example using fork/execvp

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

int main(int argc, char** argv) {
    pid_t pid;
    int status;
    char * my_argv[4];
    my_argv[0] = "ls";
    my_argv[1] = "-l";
    my_argv[2] = "-t";
    my_argv[3] = NULL;
    if ((pid = fork()) == 0) {
        //Child runs this
        execvp(my_argv[0], my_argv);
        fprintf(stderr, "Should NEVER get here!\n");
        exit(1);
    }

   //Parent continues here
    /*Comment out the following line for the parent
     * to run concurrently with the child.  (i.e.
```

```
     * to make the parent run in the "background") */
    wait(&status);
    printf("Hello from parent with child %d\n", pid);
    return (EXIT_SUCCESS);
}
```

I/O Redirection

(Note: a more complete examination of this topic will have to wait until we discuss file systems and device drivers later in the course.)

1. stdin, stdout and stderr

- Any OS that allows C programming opens 3 "files", *stdin, stdout, stderr* before the program starts.
- The OS maintains a table with an entry for each open file.
- Entry 0 is *stdin.*
- Entry 1 is *stdout*
- Entry 2 is *stderr.*
- Note that a "file" may be a *device* (such as a window on the screen or a keyboard or a printer) as well as (of course) an "ordinary file".
- By default, *stdin* is the keyboard and both *stdout* and *stderr* are the terminal (window).

2. A note about C functions that are OS calls versus "user friendly" functions

- An operating system is basically defined by a set of functions that implement OS services.
- These functions usually operate in supervisor mode whereas the "user friendly" functions work in user mode.
- Any programmer can add new user mode functions; only the OS kernel writers can write functions that operate in supervisor mode.
- OS (or kernel) functions tend to be very low level and sometimes difficult to use.
- For example, "printf" is a user-mode function that is "easy" to use.
- But the actual output is done by the much lower-level "write" system call where the bytes to write and an index into the file descriptor table are required. (Of course, the user-level writer of the printf function does invoke "write" but all the lower-level details are hidden from the user.)

- By the way, although *stderr* and *stdout* are both by default connected to the terminal, they do ***not*** behave identically. In particular, stdout is *buffered* whereas stderr is *unbuffered.* (i.e. writing a character to stderr appears immediately whereas a byte to stdout is only written once it is "flushed" either because the buffer is full, a newline is output, the program exits or the programmer explicitly flushes it with "fflush(stdout);"
- In Unix, the "manual" is divided into 8 sections. Section 1 describes user commands (ls, mkdir, or whatever you type to the shell), section 2 describes operating system calls (such as open, close, dup, read, write, fork, exec, etc.) and section 3 describes the functions most programmers use (such as getchar, printf, fopen). Note: it is unlikely that you have ever written a C program before this course that used any Section 2 functions!

Here's a simple example illustrating that stderr and stdout do not behave the same way:

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

int main(int argc, char** argv) {
    int i;
    char ch = 'a';
    char ch2 = 'A';
    for(i = 0; i < 20; i++) {
        putc(ch + i, stdout);
        if(i == 10) {
            fflush(stdout);
        }
        putc(ch2 + i, stderr);
    }
    return (EXIT_SUCCESS);
}
```

3. Redirection and piping

Some useful functions (not supervisor mode):

- `FILE * fopen(const char * filename, char * mode)`: opens a file returning a pointer to its "handle"; mode can be "r" (read), "w" (write), "rw" (red/write), etc
- `int fileno(FILE * f)`: Get the file descriptor (index into file table) for this file. (OS calls deal with file descriptors, not handles.)

Some useful functions (OS calls or supervisor mode):

- `int open(const char path, int flags, int mode)`:
- `close(int fd)`: Close the file.
- `dup(int fd)`: Duplicates the file descriptor at the lowest numbered unused slot in the table of open files.
- `dup2(int oldfd, int newfd)`: Duplicates the oldfd to newfd (closing newfd first

if that slot was in use.)

Here's a simple example of redirecting stderr (file descriptor number 2):

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

int main(int argc, char** argv) {
    int i;
    char ch = 'a';
    char ch2 = 'A';
    FILE * f = fopen("junk", "w");
    dup2(fileno(f), 2);
    for(i = 0; i < 20; i++) {
        putc(ch + i, stdout);
        if(i == 10) {
            fflush(stdout);
        }
        putc(ch2 + i, stderr);
    }
    return (EXIT_SUCCESS);
}
```

Here's an example of piping two commands:

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

char *cmd1[] = {"ls", 0}; //Output goes to pipe input
char *cmd2[] = {"/usr/bin/tr", "a-z", "A-Z", 0};//Input comes from pipe
output

int
main(int argc, char **argv) {
    int pid, status;
    int fd[2];

    pipe(fd); //Create a pipe; fd[0] is input, fd[1] is output

    switch (pid = fork()) {

        case 0: /* child */

            /* Using close/dup */
            close(0);
            dup(fd[0]);
            /* Preferred: use dup2*/
//            dup2(fd[0], 0); //pipe input is now stdin for child
            close(fd[1]);   //child does not use pipe output
            execvp(cmd2[0], cmd2); //child executes "tr" command
            perror(cmd2[0]); //SHOULD NOT GET HERE!
            exit(1);
```

```
            default: /* parent */
                dup2(fd[1], 1);
                close(fd[0]); /* the parent does not need this end of the pipe
*/
                execvp(cmd1[0], cmd1);
                perror(cmd1[0]);  //SHOULD NOT GET HERE (exec failure)
                break;

            case -1://SHOULD NOT GET HERE (indicates fork failure)
                perror("fork");
                exit(1);
        }
        exit(0);
    }
```

You can download this file here.

Notes on Lab 2

- In class discussion.

Notes on Lab 3

- In class discussion.