Lecture 3: Process API

Mythili Vutukuru
IIT Bombay
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 commandline 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\n");
    } else { // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", wc, rc, (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 *margs[3];
        margs[0] = strdup("wc"); // program: "wc" (word count)
        margs[1] = strdup("p4.c"); // argument: file to count
        margs[2] = NULL; // marks end of array
        execvp(margs[0], margs); // 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>
```