Unit 2 – CPU Virtualization

The *fork* system call

The following program shows how to use the `fork()` system call to create a child process as a replica of its parent. You can compile it using command `gcc fork.c -o fork`.

File `fork.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 ret = fork();
    if (ret < 0)
    {
         fprintf(stderr, "fork failed\n");
         exit(1);
    }
    else if (ret == 0)
    {
         printf("I am child (pid = %d)\n", (int) getpid());
    }
    else
    {
         printf("I am parent of %d (pid = %d)\n", ret, (int) getpid());
    }
    return 0;
}
```

Sample program output

```
$ ./fork
hello world (pid = 28725)
I am the parent of 28726 (pid = 28725)
I am the child (pid = 28726)
```

The *wait* system call

The program below shows how a parent process can wait for completion of its child by running the `wait()` system call. You can compile the program by running `gcc fork-wait.c -o fork-wait`.

File `fork-wait.c`

```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 ret = fork();
    if (ret < 0)
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (ret == 0)
    {
        printf("I am child %d\n", (int) getpid());
    }
    else
    {
        int w = wait(NULL);
        printf("I am parent of %d (w = %d)\n", ret, w);
    }
    return 0;
}
```

Sample program output

```
$ ./fork-wait
hello world (pid = 28732)
I am child 28733
I am parent of 28733 (w = 28733)
```

The *exec* system call

The next program shows how a child process can start running a different program by transforming its process image with system call `exec()`. You can compile this code by running `gcc exec.c -o exec`.

File `exec.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 ret = fork();
    if (ret < 0)
    {
         fprintf(stderr, "fork failed\n");
         exit(1);
    }
    else if (ret == 0)
    {
         printf("I am the child with pid %d\n", (int) getpid());

         // Launch command "wc exec.c"
         char *myargs[3];
         myargs[0] = "wc";
         myargs[1] = "exec.c";
         myargs[2] = NULL;
         execvp(myargs[0], myargs);
         printf("this is unreachable code");
    }
    else
    {
         int w = wait(NULL);
         printf("I am the parent of %d (w = %d)\n", ret, w);
    }
    return 0;
}
```

Sample program output

```
$ ./exec
hello world (pid = 28734)
I am the child with pid 28735
 36  96 661 exec.c
I am the parent of 28735 (w = 28735)
```

Redirecting the standard output

This program illustrates how to redirect the standard output of a child process that executes command `ls -l`. After the program finishes, the output of the child process will be available in file `list.txt`. Running this program is equivalent to invoking shell command `ls -l > list.txt`.

File `redirect-out.c`

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

int main(int argc, char **argv)
{
    int ret = fork();
    if (ret < 0)
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (ret == 0)
    {
        // Close default standard output
        close(1);

        // Redirect output to "list.txt"
        int fd = open("list.txt", O_WRONLY | O_CREAT, 0660);
        if (fd < 0)
        {
            fprintf(stderr, "cannot open list.txt\n");
            exit(1);
        }

        // Launch command "ls -l"
        char *myargs[3];
        myargs[0] = "ls";
        myargs[1] = "-l";
        myargs[2] = NULL;
        execvp(myargs[0], myargs);
        printf("this is unreachable code");
    }
    else
    {
        int w = wait(NULL);
        printf("I am the parent of %d (w = %d)\n", ret, w);
    }
    return 0;
}
```

Redirecting the standard input

This program runs a child process that executes command `wc` without any arguments. This makes it read its input data from the standard input, which is redirected to file `list.txt`. Executing this program is equivalent to invoking shell command `wc < list.txt`.

File `redirect-in.c`

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv)
{
    int ret = fork();
    if (ret < 0)
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (ret == 0)
    {
        // Close default standard output
        close(0);

        // Redirect output to "list.txt"
        int fd = open("list.txt", O_RDONLY);
        if (fd < 0)
        {
            fprintf(stderr, "cannot open list.txt\n");
            exit(1);
        }

        // Launch command "wc"
        char *myargs[2];
        myargs[0] = "wc";
        myargs[1] = NULL;
        execvp(myargs[0], myargs);
        printf("this is unreachable code");
    }
    else
    {
        int w = wait(NULL);
        printf("I am the parent of %d (w = %d)\n", ret, w);
    }
    return 0;
}
```

Using pipes to synchronize and communicate processes

In the following program, a parent and a child process share a pipe. The child process writes string "Hello, world!" on the pipe, and the parent process reads it and prints it into the standard output.

File `pipe.c`

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

int main(void)
{

    // Create the pipe
    int fd[2];
    pipe(fd);

    // Fork
    pid_t childpid = fork();
    if (childpid < 0)
    {
        // Function "perror" print last system call error
        perror("fork");
        return 1;
    }
    else if (childpid == 0)
    {
        // Child process closes up input side of pipe
        close(fd[0]);

        // Send "string" through the output side of pipe
        char *s = "Hello, world!\n";
        write(fd[1], s, (strlen(s) + 1));
        return 0;
    }
    else
    {
        // Parent process closes up output side of pipe
        close(fd[1]);

        // Read in a string from the pipe
        char buffer[80];
        int nbytes = read(fd[0], buffer, sizeof(buffer));
        printf("Received string: %s", buffer);
    }

    return 0;
}
```

External commands communicating through a pipe

The following program consists of two processes: parent and child. The parent process runs command `ls -l`, and the child runs command `wc`. The standard output of `ls` is forwarded into the standard input of `wc` through a pipe. The result is equivalent to running shell command `ls -l | wc`.

File `pipe-dup.c`

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

int main(int argc, char **argv)
{
	// Create pipe
	int fds[2];
	int err = pipe(fds);
	if (err == -1)
	{
		perror("pipe");
		return 1;
	}

	// Spawn child
	int ret = fork();
	if (ret < 0)
	{
		perror("fork");
		return 1;
	}
	else if (ret == 0)
	{
		// Close write end of pipe
		close(fds[1]);

		// Duplicate read end of pipe in standard input
		close(0);
		dup(fds[0]);

		// Child launches command "wc"
		char *argv[2];
		argv[0] = "wc";
		argv[1] = NULL;
		execvp(argv[0], argv);
	}
	else
	{
		// Close read end of pipe
		close(fds[0]);

		// Duplicate write end of pipe in standard output
		close(1);
		dup(fds[1]);
```

```
        // Parent launches command "ls -l"
        char *argv[3];
        argv[0] = "ls";
        argv[1] = "-l";
        argv[2] = NULL;
        execvp(argv[0], argv);
    }
    return 0;
}
```

Signal handlers

The program below is a basic example of the installation of a signal handler for signal `SIGHUP`. After the program installs the handler, it enters an infinite loop. The only way to terminate the program execution is sending it a signal whose default handler causes the program to end, such as `SIGINT`. If run in the foreground, this signal can be sent by hitting *Control+C*. If run in the background, shell command `kill` can be used to send this signal, with the following sequence of commands:

```
    $ ./signal &
    [1] 3560

    $ kill -SIGHUP 3560
    Signal 1 received

    $ kill -SIGINT 3560
    [1] Interrupted
```

File `signal.c`

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

void handler(int signum)
{
    printf("Signal %d received\n", signum);
}

int main()
{
    signal(SIGHUP, handler);
    while (1);
}
```

Sending signals

This program uses a parent and child process to illustrate process communication through signals. The child thread installs a signal handler for `SIGHUP` and enters an infinite loop. The parent thread suspends itself for 1 second (call `sleep`), sends the child a `SIGHUP` signal, and exits. The 1 second delay is used to guarantee that the child installs its signal handler before it receives the signal.

File `kill.c`

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

void handler(int signum)
{
    printf("Child process %d exits\n", getpid());
    exit(0);
}

int main()
{
    int pid = fork();
    if (pid < 0)
    {
        perror("fork");
        return 1;
    }
    else if (pid == 0)
    {
        // Child installs signal handler
        signal(SIGHUP, handler);

        // Infinite loop
        while (1);
    }
    else
    {
        // Parent waits 1 second
        sleep(1);

        // Parent sends signal SIGHUP to child
        kill(pid, SIGHUP);

        // Parent exits
        printf("Parent process %d exits\n", getpid());
        exit(0);
    }
}
```

Limited direct execution (privilege modes)

	OS (kernel mode)	Hardware	Program (user mode)
OS boot	Initialize trap table		
		Remember address of –Syscall handler	
Run	–Create entry in process list		
	–Allocate memory for program		
	–Load program into memory		
	–Setup user stack with `argv`		
	–Setup kernel stack with reg/PC		
	–return-from-trap (`iret`)		
		–Restore regs from kernel stack	
		–Move to user mode	
		–Jump to `main()`	
			–Run `main()`
			–Make system call
			–trap into OS (`int 0x80`)
		–Save regs to kernel stack	
		–Move to kernel mode	
		–Jump to trap handler	
	–Handle trap		
	–Do work of syscall		
	–return-from-trap (`iret`)		
		–Restore regs from kernel stack	
		–Move to user mode	
		–Jump to PC after trap	
			[…]
			–Return from main
			–trap via `exit()`
	–Free memory of processes		
	–Remove from process list		

Limited direct execution (timer interrupts)

	OS (kernel mode)	Hardware	Program (user mode)
	Initialize trap table		
		Remember address of –Syscall handler –Timer handler	
OS boot	Start interrupt timer		
		–Start timer –Interrupt CPU in `XXX` ms	
Run		...	
			Process A runs
		timer interrupt received –Save regs(A) to k-stack(A) –Move to kernel mode –Jump to trap handler	
	–Handle the trap –Call `switch()` –Save regs(A) to proc-struct(A) –Restore regs(B) from proc-struct(B) –Switch to k-stack(B) **return-from-trap (into B)**		
		–Restore regs(B) from k-stack(B) –Move to user mode –Jump into B's PC	
			Process B runs