2013

Lebanese University
Faculty of Sciences1
CS Department

D. FAOUR

[INFO 324 E – LAB ON PROCESSES AND SIGNALS]

Dear students: try to implement the following codes and interpret the results. Feel free to make the changes you want and send feedback to ahmad.faour@ul.edu.lb

```c
/*
 * forks.c - Examples of Unix process control
 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

/*
 * fork0 - The simplest fork example
 * Call once, return twice
 * Creates child that is identical to parent
 * Returns 0 to child process
 * Returns child PID to parent process
 */
void fork0()
{
    if (fork() == 0) {
        printf("Hello from child\n");
    }
    else {
        printf("Hello from parent\n");
    }
}

/*
 * fork1 - Simple fork example
 * Parent and child both run same code
 * Child starts with identical private state
 */
void fork1()
{
    int x = 1;
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    }
    else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}

/*
 * fork2 - Two consecutive forks
 * Both parent and child can continue forking
 * Ordering undetermined
 */
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
```

```
    printf("Bye\n");
}


/*
 * fork3 - Three consective forks
 * Parent and child can continue forking
 */
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}

/*
 * fork4 - Nested forks in parents
 */
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}

/*
 * fork5 - Nested forks in children
 */
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}

void cleanup(void) {
    printf("Cleaning up\n");
}

/*
 * fork6 - Exit system call terminates process
```

```
 * call once, return never
 */
void fork6()
{
    atexit(cleanup);
    fork();
    exit(0);
}

/*
 * fork7 - Demonstration of zombies.
 * Run in background and then perform ps
 */
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}

/*
 * fork8 - Demonstration of nonterminating child.
 * Child still running even though parent terminated
 * Must kill explicitly
 */
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}

/*
 * fork9 - synchronizing with and reaping children (wait)
 */
void fork9()
{
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    } else {
        printf("HP: hello from parent\n");
```

```
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}

#define N 5
/*
 * fork10 - Synchronizing with multiple children (wait)
 * Reaps children in arbitrary order
 * WIFEXITED and WEXITSTATUS to get info about terminated children
 */
void fork10()
{
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}

/*
 * fork11 - Using waitpid to reap specific children
 * Reaps children in order
 */
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

```
/*********
 * Signals
 *********/

/*
 * fork12 - Sending signals with the kill() function
 */
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}

/*
 * int_handler - SIGINT handler
 */
void int_handler(int sig)
{
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

/*
 * fork13 - Simple signal handler example
 */
void fork13()
{
    pid_t pid[N];
    int i;
    int child_status;

    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
```

```
                ;
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}


/*
 * child_handler - SIGCHLD handler that reaps one terminated child
 */
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received SIGCHLD signal %d for process %d\n", sig, pid);
    fflush(stdout);
}

/*
 * fork14 - Signal funkiness: Pending signals are not queued
 */
void fork14()
{
    pid_t pid[N];
    int i;
    ccount = N;
    signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            exit(0);  /* Child: Exit */
        }
    }
    while (ccount > 0)
        pause();
}


/*
 * child_handler2 - SIGCHLD handler that reaps all terminated children
 */
void child_handler2(int sig)
```

```
{
    int child_status;
    pid_t pid;
    while ((pid = wait(&child_status)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
        fflush(stdout);
    }
}

/*
 * fork15 - Using a handler that reaps multiple children
 */
void fork15()
{
    pid_t pid[N];
    int i;
    ccount = N;

    signal(SIGCHLD, child_handler2);

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            exit(0); /* Child: Exit */

        }
    while (ccount > 0) {
        pause();
    }
}

/*
 * fork16 - Demonstration of using /bin/kill program
 */
void fork16()
{
    if (fork() == 0) {
        printf("Child1: pid=%d pgrp=%d\n",
               getpid(), getpgrp());
        if (fork() == 0)
            printf("Child2: pid=%d pgrp=%d\n",
                   getpid(), getpgrp());
        while(1);
    }
}

/*
 * Demonstration of using ctrl-c and ctrl-z
 */
void fork17()
{
    if (fork() == 0) {
        printf("Child: pid=%d pgrp=%d\n",
               getpid(), getpgrp());
    }
    else {
```

```
        printf("Parent: pid=%d pgrp=%d\n",
               getpid(), getpgrp());
    }
    while(1);
}


int main(int argc, char *argv[])
{
    int option = 0;
    if (argc > 1)
        option = atoi(argv[1]);
    switch(option) {
    case 0: fork0();
        break;
    case 1: fork1();
        break;
    case 2: fork2();
        break;
    case 3: fork3();
        break;
    case 4: fork4();
        break;
    case 5: fork5();
        break;
    case 6: fork6();
        break;
    case 7: fork7();
        break;
    case 8: fork8();
        break;
    case 9: fork9();
        break;
    case 10: fork10();
        break;
    case 11: fork11();
        break;
    case 12: fork12();
        break;
    case 13: fork13();
        break;
    case 14: fork14();
        break;
    case 15: fork15();
        break;
    case 16: fork16();
        break;
    case 17: fork17();
        break;
    default:
        printf("Unknown option %d\n", option);
        break;
    }
    return 0;
}
```