Mohit Jain

CS 1550

Lab Assignment #2

Part 1 Processes (Intro)

1. The CPU utilization should be 100% because we give 5 tasks and assign the value 100 which refers to the amount of work the CPU should do.
2. It takes a total time of 10.
3. When I switched the order of the processes, I noticed that the IO instruction was first, then for the rest of the 4 instructions, the CPU ran throughout the rest of the time. The order clearly does matter because, in the first parameter we specify the number of instructions to run. This seems like it is the primary parameter as it tells the system what to do. In the second parameter, we simply tell the system how it wants to end, so by telling the CPU to run in the first parameter in itself, this will take more time than if we told the IO to run. By telling the CPU to run at the end, it will print out the last number of instructions as needed. I think it takes more time to stop the CPU from running than it does to stop the IO from running.
4. The system waits for 4 extra time intervals with a “READY” message before running the CPU and after running the IO.
5. It seems like all the PID: 0 that were initially “DONE” when we had the SWITCH\_ON\_END turned on were turned into “WAITING.”
6. First, the IO was switched on, which caused the RUN:cpu to wait. Then, Once that process was done, IO was switched off, and the CPU ran without interruption until it was done. Once that was done, then the IO completed the rest of its processes. System resources are not being effectively utilized.
7. Unlike RUN\_LATER, RUN\_IMMEDIATE completes the process a lot faster and take 18 time instead of 27 time. It helps to complete processes a lot faster if IO worked alongside CPU instead of separately.
8. When you use –I IO\_RUN\_IMMEDIATE, the IO and CPU processes will be clustered together by running on the same time interval at the same time and thus it less time than IO\_RUN\_LATER, in which the CPU and IO processes will run on separate time intervals. When you use –S SWITCH\_ON\_IO, then the IO processes will be prioritized and the CPU processes will be put on WAITING, whereas if you use the –S SWITCH\_ON\_END, the CPU processes will be prioritized.

Part 2 Process API

**Problem #1**

//

//  Problem1.c

//  Lab2

//

//  Created by Mohit Jain on 1/18/18.

//  Copyright © 2018 Mohit Jain. All rights reserved.

//

#include "Problem1.h"

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, const char \* argv[]) {

    int x = 100;

    int rc = fork();

    if (rc < 0) {         // fork failed; exit

        fprintf(stderr, "fork failed\n");

        exit(1);

    }

    else if (rc == 0) { // child (new process)

        x = 12;

        printf("hello, I am child\n");

        printf("x = %d\n", x);

    }

    else {              // parent goes down this path (main)

        x = 11;

        printf("hello, I am parent of %d\n", rc);

        printf("x = %d\n", x);

    }

    return 0;

}

// To answer Problem #1 question, the value of the child is 12.

// When both the parent and the child process and the parent process

// change the value of x, then the child's process value is the only value read.

**Problem #2**

//

//  Problem2.c

//  Lab2

//

//  Created by Mohit Jain on 1/18/18.

//  Copyright © 2018 Mohit Jain. All rights reserved.

//

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, const char \* argv[]) {

    FILE\* fp;

    fp = fopen("q2.txt", "w"); // change this file name to whatever you want

    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\n");

    }

    else {              // parent goes down this path (main)

        printf("hello, I am parent of %d\n", rc);

    }

    // The problem did not state to read or write to the file,

    // so I did not include any read or write functions.

    fclose(fp);

    return 0;

}

// To answer Problem #2 question, both the child and the parent

// processes can access the file descriptor. They can both

// write to the file at the same time, but it is up to the

// OS to decide whether the child processor should run first or the

// parent.

**Problem #3**

//

//  Problem3.c

//  Lab2

//

//  Created by Mohit Jain on 1/18/18.

//  Copyright © 2018 Mohit Jain. All rights reserved.

//

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, const char \* argv[]) {

    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\n");

    }

    else {              // parent goes down this path (main)

        wait(NULL);

        printf("goodbye\n");

    }

    return 0;

}

// To answer Problem #3 question, you can not

// accomplish this without a wait() call.

**Problem #4**

//

//  Problem4.c

//  Lab2

//

//  Created by Mohit Jain on 1/18/18.

//  Copyright © 2018 Mohit Jain. All rights reserved.

//

#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");

    }

    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;

}

// To answer Problem #4 question, I think there are many variants

// because there are different methods to execute different processes.

**Problem #5**

The code in this problem is the exact same as Problem #3 as problem #3 implements the wait() function.

To answer the questions in this problem, wait() returns an integer. If you try to put the wait() function in the child process instead of the parent, the program will simply execute the parent process first, then it will execute the child’s process.

**Problem #6**

//

//  Problem6.c

//  Lab2

//

//  Created by Mohit Jain on 1/18/18.

//  Copyright © 2018 Mohit Jain. All rights reserved.

//

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, const char \* argv[]) {

    int x;

   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\n");

    }

    else {// parent goes down this path (main)

        while(waitpid(rc, &x, 0) == 0) {

            sleep(1);

        }

        printf("goodbye\n");

    }

    return 0;

}

// To answer Problem #6 question, waitpid() is more useful

// because it seems more flexible. There are more parameters to

// make use of.

**Problem #7**

If a child calls printf() to print some output after closing the descriptor, nothing will be printed out.