CST-321 Activity 2 Guide

Contents

Processes in Linux

Overview

In this activity, students will learn about processes using the Linux fork() and posix_spawn() functions.

Execution

Execute this assignment according to the following guidelines:

1. Access the following article, tutorials, and videos:
    a. Read the article "The fork() System Call," located in the topic Resources.
    b. View the video "Getting Started Creating Processes in Linux using fork."
    c. View the video playlist "Learning C."
    d. Download the Intro to C Programming training deck located in the topic Resources and study Modules 3 and 4.
2. Per guidance from your instructor create a small C program with the following capabilities:
    a. From the main() function using the fork() function to create a parent and child process.
    b. The child process should print 10 messages to the console and sleep for 1 second between message then exit the process with a return code of 0.
    c. The parent process should print 10 messages to the console and sleep for 2 seconds between message then exit the process with a return code of 0.
    d. Take a screenshot of the Terminal and Console window output.
    e. After reading the article ""The fork() System Call," write a theory of operation explaining how the code worked.
3. Per guidance from your instructor create a small C program with the following capabilities:
    a. From the main() function, accept a command line argument that will be used to spawn off the passed in application.

b. Use the POSIX posix_spawn() function to spawn off the application.
c. Print the process ID to the console.
d. Wait for the process to end by calling the waitpid() function.
e. Take a screenshot of the Terminal and Console window output.
f. Research the documentation on the posix_spawn() and waitpid() functions and write a theory of operation for how the code worked.

Signals in Linux

Overview

In this activity, students will learn about signals using the Linux kill and signal functions.

Execution

Execute this assignment according to the following guidelines:

1. View the following video:
    a. "Signals in Linux."
2. Per guidance from your instructor, create a small C program with the following capabilities:
    a. From the main() function, use the fork() function to create a parent producer and child consumer process.
    b. The child consumer process should create a signal called WAKEUP, sleep until that signal is received, then once received run a loop for 20 iterations printing a message to the console and sleeping for 1 second for each iteration, then exit the process with a return code of 1.
    c. The parent producer process should run a loop for 30 iterations printing a message to the console and sleeping for 1 second for each iteration and once a count of 5 has been reached send the WAKEUP signal, then exit the process with a return code of 1.
    d. Take a screenshot of the Terminal and Console window output.
    e. After reading the article "The fork() System Call," and reading the documentation on signals, write a theory of operation explaining how the code worked.

Threads in Linux

Overview

In this activity, students will learn about Threads using the Linux POSIX thread functions.

Execution

Execute this assignment according to the following guidelines:

1. View the following videos:
    a. "Getting Started with Pointers in C."
    b. "Understanding the Stack vs. the Heap in C."
    c. "Getting Started Creating Thread in Linux using pthreads."

d. Download the Intro to C Programming training deck located in the topic Resources and study Modules 3 and 4.

2. Per guidance from your instructor, create a small C program with the following capabilities:
    a. From the main() function, use the pthread_create() function to create thread 1 and thread 2.
    b. The thread 1 function should print 10 messages to the console and sleep for 1 second between message then exit the thread with a return code of null.
    c. The thread 2 function should print 10 messages to the console and sleep for 2 seconds between message then exit the thread with a return code of null.
    d. The main() function should use the pthread_join() function to wait for each thread to exit before exiting the main program.
        i. **Note:** Make sure to use the -pthread flag as an option in the gcc compiler!
    e. Take a screenshot of the Terminal and Console window output.
    f. Research the documentation on the POSIX thread functions and write a theory of operation for how the code worked.

Mutexes and Semaphores in Linux

Overview

In this activity, students will learn about Mutexes using the Linux POSIX mutex and semaphore functions.

Execution

Execute this assignment according to the following guidelines:

1. View the following videos:
    a. View the video "Getting Started Using a Mutex in Linux."
    b. View the video "Getting Started Using a Semaphore in Linux."
    c. View the video "Difference between a Mutex and Semaphore."
2. Per guidance from your instructor, create a small C program with the following capabilities:
    a. Write a bad bank program that simulates bank deposit transactions.
    b. From the main() function, use the pthread_create() function to create 2 threads.
    c. Each thread will call the same function, which simulates a bank deposit transaction, then should sit in a loop for each least 1,000,000 transactions and adding 1 (simulating $1 deposit) to a global variable representing a bank balance that starts out with a value of 0. The thread should return null after all deposits have been made.
    d. Run the program. The expected bank balance should be $2,000,000 (2 threads each depositing $1,000,000. Take a screenshot of the Terminal and Console window output. Write a theory of operation explaining why the program did not behave as expected and address the offending line or lines of code that are causing an issue.
        i. **Note**: Make sure to use the -pthread flag as an option in the gcc compiler!
3. Per guidance from your instructor create a small C program with the following capabilities:
    a. Fix the bad bank program by using POSIX mutexes.

b. Run the program. The expected bank balance should be $2,000,000 (2 threads each depositing $1,000,000). Take a screenshot of the Terminal and Console window output. Write a theory of operation explaining why the program now behaves properly with the mutexes.
    i. **Note**: Make sure to use the -pthread flag as an option in the gcc compiler!

4. Per guidance from your instructor, create a small C program with the following capabilities:
    a. Fix the bad bank program by using POSIX semaphores.
    b. Run the program. The expected bank balance should be $2,000,000 (2 threads each depositing $1,000,000). Take a screenshot of the Terminal and Console window output. Write a theory of operation explaining why the program now behaves properly with the semaphores.
        i. **Note**: Make sure to use the -pthread flag as an option in the gcc compiler!

Research Questions

Research Questions: For traditional ground students, answer the following questions in a Microsoft Word document:

a. Consider the following code for a process P0 (assume turn has been initialized to zero):

```
--- Etc. ---
while (turn != 0) {}
Critical Section /* … */
turn = 0;
--- Etc. ---
```

For process P1, the code is:

```
--- Etc. ---
while (turn != 1) {}
Critical Section /* … */
turn = 1;
--- Etc. ---
```

Does the above code meet all the required conditions for a correct mutual-exclusion solution? Explain and justify your answer.

b. Consider the following C code:

```
void main()
{
    fork();
    fork();
```

```
    exit();
}
```

How many child processes are created upon execution of this program? Run the code on your computer and support your answers with relevant screenshots.

Submission

1. In a Microsoft Word document, complete the following for the activity report:
    a. Cover sheet with your name, the name of this assignment, and the date.
    b. Section with a title that contains all theory of operation write-ups, answers to questions asked in the activity, and any screenshots taken during the activity.
    c. Section with a title that contains the answers to the research questions (traditional ground students only).

Submit the activity report to the digital classroom.

Appendix A – Sample Programs

The following can be used as guidance to program the C programs in the activity.

Linux Process Examples

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

/**
 * Logic to run to simulare a Parent Process
 **/
void childProcess()
{
    // Child process: Simple example where we count up to 10 and sleep for a second between each count
    for (int x=0;x < 10;++x)
    {
        printf("Child Process: %d\n", x);
         sleep(1);
    }

    // Exit the Child Process (use _exit() and NOT exit()
    _exit(0);
}

/**
 * Logic to run to simulare a Parent Process
 **/
void parentProcess()
{
    // Parent process: Simple example where we count up to 10 and sleep for 2 seconds between each count
    for (int y=0;y < 10;++y)
    {
        printf("Parent Process: %d\n", y);
        sleep(2);
    }

    // Exit the Child Process (use _exit() and NOT exit()
    _exit(0);
}

/**
 * Main application entry point to demonstrate forking off a child process that will run concurrently with this process.
 *
 * @return 1 if error or 0 if OK returned to code the caller.
 */
int main(int argc, char* argv[])
{
    pid_t  pid;

    // Use fork()
    // Note: The output from both the child and the parent process will be written to standard out since they both run at
    pid = fork();
    if (pid == -1)
    {
        // Error: If fork() returns -1 then an error happened (for example, number of processes reached the limit).
        fprintf(stderr, "can't fork, error %d\n", errno);
        exit(EXIT_FAILURE);
    }
    // OK: If fork() returns 0 then the child process is running
    if (pid == 0)
    {
        // Run Child Process logic
        childProcess();
    }
    else
    {
        // Run Parent Process logic
        parentProcess();
    }

   // Return OK
   return 0;
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>

extern char **environ;

/**
 * Spawn a child process and wait for it to complete.
 *
 * @param a Not used.
 * @return Always null.
 */
void run_cmd(char *cmd)
{
    pid_t pid;
    char *argv[] = {"sh", "-c", cmd, NULL};
    int status;

    // Spawn off the provided command run
    printf("Running command..... %s\n", cmd);
    status = posix_spawn(&pid, "/bin/sh", NULL, NULL, argv, environ);
    if (status == 0)
    {
        // If all OK then print the PID and wait for this process to finish
        printf("Child Process ID (PID) is: %i\n", pid);
        if (waitpid(pid, &status, 0) != -1)
        {
            printf("Child Process exited with status of %i\n", status);
        }
        else
        {
            perror("Failed to wait for Child Process");
        }
    }
    else
    {
        // Else print error
        printf("Child Process failed to spawn with error of '%s'\n", strerror(status));
    }
}

/**
 * Main application entry point to spawn a child process provided argv[1] program argument.
 *
 * @return 1 if error or 0 if OK returned to code the caller.
 */
int main(int argc, char* argv[])
{
    // Run a desired command by spawning off a Child Process provided argv[1] program argument
    run_cmd(argv[1]);
    return 0;
}
```

Linux Thread Example

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

/**
 * Thread 1 function.
 *
 * @param a Not used.
 * @return Always null.
 */
void *thread1(void *a)
{
    int x;
    for(x = 0;x < 10;++x)
    {
        // Print a message to the console and sleep for awhile
        printf("This is Thread 1 ..... %d\n", x);
        sleep(1);
    }
    return NULL;
}

/**
 * Thread 2 function.
 *
 * @param a Not used.
 * @return Always null.
 */
void *thread2(void *a)
{
    int x;
    for(x = 0;x < 10;++x)
    {
        // Print a message to the console and sleep for awhile
        printf("This is Thread 2 ..... %d\n", x);
        sleep(2);
    }
    return NULL;
}

/**
 * Main application entry point to create some threads.
 *
 * @return 1 if error or 0 if OK returned to code the caller.
 */
int main()
{
    pthread_t tid1, tid2;

    // Create 2 threads
    if(pthread_create(&tid1, NULL, thread1, NULL))
    {
        printf("\n ERROR creating thread 1");
        exit(1);
    }
    if(pthread_create(&tid2, NULL, thread2, NULL))
    {
        printf("\n ERROR creating thread 2");
        exit(1);
    }

    // Wait for both threads to finish
    if(pthread_join(tid1, NULL))
    {
        printf("\n ERROR joining thread 1");
        exit(1);
    }
    if(pthread_join(tid2, NULL))
    {
        printf("\n ERROR joining thread 2");
        exit(1);
    }

    // Thread creation cleanup
    pthread_exit(NULL);
}
```

Linux Mutex and Semaphore Examples

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

#define MAX_DEPOSITS 1000000

int balance = 0;

/**
 * Thread function to deposit money into the bank.
 *
 * @param a Not used.
 * @return Always null.
 */
void *deposit(void *a)
{
    int x, tmp;
    for(x = 0;x < MAX_DEPOSITS;++x)
    {
        // Copy the balance to a local variable, add $1 to the balance, and save the balance back in the global variable
        tmp = balance;
        tmp = tmp + 1;
        balance = tmp;
    }
    return NULL;
}

/**
 * Main application entry point to deposit money into a bank.
 *
 * @return 1 if error or 0 if OK returned to code the caller.
 */
int main()
{
    pthread_t tid1, tid2;

    // Create 2 threads (users) to deposit funds into bank
    if(pthread_create(&tid1, NULL, deposit, NULL))
    {
        printf("\n ERROR creating deposit thread 1");
        exit(1);
    }
    if(pthread_create(&tid2, NULL, deposit, NULL))
    {
        printf("\n ERROR creating deposit thread 2");
        exit(1);
    }

    // Wait for threads (users) to finish depositing funds into bank
    if(pthread_join(tid1, NULL))
    {
        printf("\n ERROR joining deposit thread 1");
        exit(1);
    }
    if(pthread_join(tid2, NULL))
    {
        printf("\n ERROR joining deposit thread 2");
        exit(1);
    }

    // Check balance
    if (balance < 2 * MAX_DEPOSITS)
        printf("\n BAD Balance: bank balance is $%d and should be $%d\n", balance, 2 * MAX_DEPOSITS);
    else
        printf("\n GOOD Balance: bank balance is $%d\n", balance);

    // Thread creation cleanup
    pthread_exit(NULL);
}
```

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

// Global program variables
#define MAX_DEPOSITS 1000000
int balance = 0;
int depositAmount = 1;
pthread_mutex_t mutex;

/**
 * Thread function to deposit money into the bank.
 *
 * @param a Not used.
 * @return Always null.
 */
void *deposit(void *a)
{
    int x, tmp;

    for(x = 0;x < MAX_DEPOSITS;++x)
    {
        // *** Start of Critical Region ***
        pthread_mutex_lock(&mutex);

        // NOT THEAD SAFE!!
        //  Copy the balance to a local variable, add $1 to the balance, and save the balance back in the global variable
        tmp = balance;
        tmp = tmp + depositAmount;
        balance = tmp;

        // *** End of Critical Region ***
       pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

/**
 * Main application entry point to deposit money into a bank.
 *
 * @return 1 if error or 0 if OK returned to code the caller.
 */
int main()
{
    pthread_t tid1, tid2;

    // Create a mutex to be used in a critical region of our code
    pthread_mutex_init(&mutex, 0);

    // Create 2 threads (users) to deposit funds into bank
    if(pthread_create(&tid1, NULL, deposit, NULL))
    {
        printf("\n ERROR creating deposit thread 1");
        exit(1);
    }
    if(pthread_create(&tid2, NULL, deposit, NULL))
    {
        printf("\n ERROR creating deposit thread 2");
        exit(1);
    }

    // Wait for threads (users) to finish depositing funds into bank
    if(pthread_join(tid1, NULL))
    {
        printf("\n ERROR joining deposit thread 1");
        exit(1);
    }
    if(pthread_join(tid2, NULL))
    {
        printf("\n ERROR joining deposit thread 2");
        exit(1);
    }

    // Check balance
    if (balance < 2 * MAX_DEPOSITS)
        printf("\n BAD Balance: bank balance is $%d and should be $%d\n", balance, 2 * MAX_DEPOSITS);
    else
        printf("\n GOOD Balance: bank balance is $%d\n", balance);

    // Thread and mutex creation cleanup
    pthread_mutex_destroy(&mutex);
    pthread_exit(NULL);
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>

// Global program variables
#define MAX_DEPOSITS 1000000
int balance = 0;
int depositAmount = 1;
sem_t* mutex;

/**
 * Thread function to deposit money into the bank.
 *
 * @param a Not used.
 * @return Always null.
 */
void *deposit(void *a)
{
    int x, tmp;

    for(x = 0;x < MAX_DEPOSITS;++x)
    {
        // *** Start of Critical Region ***
        sem_wait(mutex);

        // NOT THEAD SAFE!!
        //  Copy the balance to a local variable, add $1 to the balance, and save the balance back in the global variable
        tmp = balance;
        tmp = tmp + depositAmount;
        balance = tmp;

        // *** End of Critical Region ***
       sem_post(mutex);
    }
    return NULL;
}

/**
 * Main application entry point to deposit money into a bank.
 *
 * @return 1 if error or 0 if OK returned to code the caller.
 */
int main()
{
    pthread_t tid1, tid2;

    // Create a mutex to be used in a critical region of our code
    mutex = sem_open("Mutex", O_CREAT, 00644, 1);
    if(mutex == SEM_FAILED)
    {
        printf("\n ERROR creating mutex");
        exit(1);
    }

    // Create 2 threads (users) to deposit funds into bank
    if(pthread_create(&tid1, NULL, deposit, NULL))
    {
        printf("\n ERROR creating deposit thread 1");
        exit(1);
    }
    if(pthread_create(&tid2, NULL, deposit, NULL))
    {
        printf("\n ERROR creating deposit thread 2");
        exit(1);
    }

    // Wait for threads (users) to finish depositing funds into bank
    if(pthread_join(tid1, NULL))
    {
        printf("\n ERROR joining deposit thread 1");
        exit(1);
    }
    if(pthread_join(tid2, NULL))
    {
        printf("\n ERROR joining deposit thread 2");
        exit(1);
    }

    // Check balance
    if (balance < 2 * MAX_DEPOSITS)
        printf("\n BAD Balance: bank balance is $%d and should be $%d\n", balance, 2 * MAX_DEPOSITS);
    else
        printf("\n GOOD Balance: bank balance is $%d\n", balance);

    // Thread and mutex creation cleanup
    sem_close(mutex);
    pthread_exit(NULL);
}
```

Linux Signal Example

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include  <signal.h>
#include <sys/mman.h>
#include <err.h>
#include <errno.h>

// Constants
int WAKEUP = SIGUSR1;

// The Child PID if the Parent else the Parent PID if the Child
pid_t otherPid;

// A Signal Set
sigset_t sigSet;

/*****************************************************************************************/

void sleepUntilWoken()
{
    int nSig;

    // Put to sleep until notified to wake up
    printf("Sleeping.......\n");
    sigwait(&sigSet, &nSig);
    printf("Awoken\n");
}

/*****************************************************************************************/

/**
 * Logic to run to the Consumer Process
 **/
void consumer()
{
    // Setup a Signal Set
    sigemptyset(&sigSet);
    sigaddset(&sigSet, WAKEUP);
    sigprocmask(SIG_BLOCK, &sigSet, NULL);

    // Put the Consumer to sleep forever
    printf("Putting consumer to sleep forever\n");
    sleepUntilWoken();

    // Run the Consumer Process
    int count = 0;
    printf("Running Consumer Process.....\n");
    while(count < 20)
    {
        printf("Consumer %d\n", count);
        sleep(1);
        ++count;
    }
    _exit(1);
}

/**
 * Logic to run to the Producer Process
 **/
void producer()
{
    // Run the Producer Process
    int count = 0;
    printf("Running Producer Process.....\n");
    while(count < 30)
    {
        printf("Producer %d\n", count);
        sleep(1);
        if(count == 5)
        {
            printf("Waking up the Consumer....\n");
            kill(otherPid, WAKEUP);
        }
        ++count;
    }
    _exit(1);
}
```

```c
/**
 * Main application entry point to demonstrate forking off a child process that will run concurrently with this process.
 *
 * @return 1 if error or 0 if OK returned to code the caller.
 */
int main(int argc, char* argv[])
{
    pid_t  pid;

    // Use fork()
    // Note: The output from both the child and the parent process will be written to standard out since they both run at the same
    pid = fork();
    if (pid == -1)
    {
        // Error: If fork() returns -1 then an error happened (for example, number of processes reached the limit).
        printf("Can't fork, error %d\n", errno);
        exit(EXIT_FAILURE);
    }
    // OK: If fork() returns non zero then the parent process is running else child process is running
    if (pid == 0)
    {
        // Run Producer Process logic as a Child Process
        otherPid = getppid();
        producer();
    }
    else
    {
        // Run Consumer Process logic as a Parent Process
        otherPid = pid;
        consumer();
    }

    // Return OK
    return 0;
}
```