CST-321 Activity 3 Guide

Contents

More Signals in Linux

Overview

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

Execution

Execute this assignment according to the following guidelines:

1. Read the following tutorials found in the Course Padlet:
    a. Read the article "The fork() System Call," located in the topic Resources.
    b. Download the Intro to C Programming training deck from the Course Materials and study Module 5.
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 that shares a buffer between them.
    b. Create a shared circular buffer that the processes can use.
    c. Have the parent process:
        i. Create an array with a desired message (like your name).
        ii. Have the parent send each character separately via the shared buffer.
        iii. Have the parent signal to the child process there is a character available.
        iv. Have the parent signal the child when the entire string has been sent by putting a null character in the shared buffer.
        v. Exit the process.
    d. Have the child process handle the received message:
        i. Wait to be notified there is a character available in the shared buffer.
        ii. Remove the received character from the shared buffer.
        iii. Print the received character on the console.
        iv. Exit if the null character was received.

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.
f. Provide a 3- to 5-minute screencast demonstration of the code running.

More Signals and Mutexes in Linux

Overview

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

Execution

Execute this assignment according to the following guidelines:

1. Per guidance from your instructor, create a small C program with the following capabilities:
    a. Create a C program that creates 2 threads that access a global variable called counter and that each run for a specified amount of time.
    b. Have the "counter thread" lock the counter, increment the counter each second, then sleep for a second with the mutex locked to give the "monitor thread" a reasonable chance of running. The counter value should be printed to the screen when this thread exits.
    c. Every 3 seconds, have the "monitor thread" try to lock the mutex and read the counter. If the try lock fails, increment a "misses" count, else print the value of the counter to the console. The "misses" count should be printed to the screen when this tread exits.
    d. The main program should specify how long (in minutes) each thread should run and also wait for both threads to finish their work before exiting.
    e. The main() function should use the pthread_join() function to wait for each thread to exit before exiting the main program.
    f. **Note:** Make sure to use the -pthread flag as an option in the gcc compiler!
    g. Write a theory of operation explaining why the program behaves properly with the mutexes.
    h. Provide a 3- to 5-minute screencast demonstration of the code running.

More Signals and Semaphores in Linux

Overview

In this activity, students will learn more about signals and semaphore using the Linux kill, signal, and semaphore functions to simulate a stuck process and terminate it.

Execution

Execute this assignment according to the following guidelines:

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 process and child process. The main() function should create a shared semaphore using the sem_init() function with an initial count of 1 before creating the parent process and child process. The semaphore can be saved in the program's global variables.
    b. Have the child process simulate a long-running or stuck process by obtaining a lock on a semaphore then sitting in a loop for at least 60 seconds before releasing the semaphore.
    c. Have the parent process try to get the semaphore that the child process has locked. This can be done by starting a thread that first sleeps for 10 seconds and then tries to see if it can get the semaphore and if it cannot, exits the thread. The thread can return a value of 1 if it was not able to get the semaphore. The parent process can then wait for the thread to exit and check if it was unable to get the semaphore; if it was not, then kill the child process. Once the parent process has killed the child process, logic can be written to determine if the semaphore can now be obtained.
    d. The main() function should use the sem_destroy() function to destroy the semaphore before exiting the main program.
    e. Write a theory of operation explaining why the program behaves properly with the mutexes.
    f. Provide a 3- to 5-minute screencast demonstration of the code running.

Research Questions

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

a. Can a system be in a state that is neither deadlocked nor safe? If so, give an example. If not, prove that all states are either deadlocked or safe.
b. Consider the following idea for deadlock prevention: When a process requests a resource, it specifies a time limit. If the process blocks because the resource is not available, a timer is started. If the time limit is exceeded, the process is released and allowed to run again. Is it a good solution?

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 or screencasts 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.

Signals Examples

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

// Constants
int MAX = 100;
int WAKEUP = SIGUSR1;
int SLEEP = SIGUSR2;

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

// A Signal Set
sigset_t sigSet;

// Shared Circular Buffer
struct CIRCULAR_BUFFER
{
    int count;          // Number of items in the buffer
    int lower;          // Next slot to read in the buffer
    int upper;          // Next slot to write in the buffer
    int buffer[100];
};
struct CIRCULAR_BUFFER *buffer = NULL;
```

```c
/**************************************************************************************************/

// This method will put the current Process to sleep forever until it is awoken by the WAKEUP signal
void sleepAndWait()
{
    int nSig;

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

// This method will signal the Other Process to WAKEUP
void wakeupOther()
{
    kill(otherPid, WAKEUP);
}

// Gets a value from the shared buffer
int getValue()
{
    // Get a value from the Buffer and adjust where to read from next
    int value = buffer->buffer[buffer->lower];
    printf("     Consumer read value of %c\n", value);
    ++buffer->lower;
    if(buffer->lower == MAX)
        buffer->lower = 0;
    --buffer->count;
    return value;
}

// Puts a value in the shared buffer
void putValue(int value)
{
    // Write to next available position in the Buffer and adjust where to write next
    buffer->buffer[buffer->upper] = value;
    printf("Producer stored value of %c\n", buffer->buffer[buffer->upper]);
    ++buffer->upper;
    if(buffer->upper == MAX)
        buffer->upper = 0;
    ++buffer->count;
}

/**************************************************************************************************/
```

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

    // Run the Child Consumer Logic
    printf("Running the Child Consumer Process.....\n");

    // Reads characters from shared buffer until a 0 is received
    int character = 0;
    do
    {
        // Wait to be notified there is a character in the shared buffer
        sleepAndWait();

        // Read the character from the shared buffer until 0 has been received
        character = getValue();
    }while(character != 0);

    // Exit cleanly from the Consumer Process
    printf("Exiting the Child Consumer Process.....\n");
    _exit(1);
}

/**
 * Logic to run to the Producer Process
 **/
void producer()
{
    // Buffer value to write
    int value = 0;

    // Run the Parent Producer Logic
    printf("Running the Parent Producer Process.....\n");

    // Send a desired message to the child consumer process
    char message[10] = "Mark Reha";
    for(int x = 0;x < strlen(message);++x)
    {
        // Put the next character in the shared buffer, notify the consume, sleep a bit
        putValue(message[x]);
        wakeupOther();
        sleep(1);
    }
    // Tell the child consumer process that all characters have been sent
    putValue(0);
    wakeupOther();

    // Exit cleanly from the Producer Process
    printf("Exiting the Parent Producer Process.....\n");
    _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;

    // Create shared memory for the Circular Buffer to be shared between the Parent and Child  Processes
    buffer = (struct CIRCULAR_BUFFER*)mmap(0,sizeof(buffer), PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS, -1, 0);
    buffer->count = 0;
    buffer->lower = 0;
    buffer->upper = 0;

    // Use fork()
    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;
}
```

Signals and Mutexes Examples

```c
void *counter_thread (void *arg)
{
    int status;

    // For the specificed time duration increment the counter, sleep for second with the mutext locked (so monitor can run),
    while (time (NULL) < end_time)
    {
        // ***** ENTER CRITICAL REGION *****
        status = pthread_mutex_lock (&mutex);
        if (status == 0)
            printf ("Counter Thread: Locked mutex for Counter so it can be incremented\n");
        ++counter;

        sleep (1);  // Sleep to hog the mutex for a greater chance of trylock failing

        status = pthread_mutex_unlock (&mutex);
        if (status == 0)
            printf ("Counter Thread: Unlocked mutex for Counter since we are done with the Counter\n");
        // ***** EXIT CRITICAL REGION *****
     }

    // Print the final value of the Counter
    printf ("Final Counter is %lu\n", counter);
    return NULL;
}

/*
 * The Monitor thread simply tries to get the mutex and if locked keeps track of the Misses count else accesses the Counter
 */
void *monitor_thread (void *arg)
{
    int status;
    int misses = 0;

    // For the specificed time duration try to get the mutext and if locked count it else access the counter and print it
    while (time (NULL) < end_time)
    {
        // Try to get the mutex and if not busy access the Counter to print it else keep track of misses
        status = pthread_mutex_trylock (&mutex);
        if (status != EBUSY)
        {
            printf ("     Monitor Thread: the trylock will lock the mutex so we can safely read the Counter\n");
            printf ("     Monitor Thread: The Counter from Monitor is %ld\n", counter);
            status = pthread_mutex_unlock (&mutex);     // !!! Remember to unlock the mutex !!!
            if (status == 0)
                printf ("     Monitor Thread: will now unlock the mutex since we are done with the Counter\n");
        }
        else
        {
            // Count Misses on the lock
            ++misses;
        }
    }

    // Print number of Misses out
    printf ("Final Monitor Thread missed was %d times.\n", misses);
    return NULL;
}
```

```c
/*
 * Application entry point to demonstrate the use of mutex trywait API.
 */
int main (int argc, char *argv[])
{
    int status;
    pthread_t counter_thread_id;
    pthread_t monitor_thread_id;

    // Initialize the mutex
    pthread_mutex_init(&mutex, 0);

    // Set the end time for 60 seconds from now
    end_time = time (NULL) + 60;

    // Create the Counter and Monitor Threads
    if(pthread_create (&counter_thread_id, NULL, counter_thread, NULL))
        printf ("Create counter thread failed");
    if(pthread_create (&monitor_thread_id, NULL, monitor_thread, NULL))
        printf ("Created monitor thread failed");

    // Wait for Counter and Monitor Threads to both finish
    if(pthread_join (counter_thread_id, NULL))
        printf ("Joined counter thread failed");
    if (pthread_join (monitor_thread_id, NULL))
        printf ("Joined monitor thread failed");

    // Exit OK
    return 0;
}
```

Signals and Semaphores with Terminal Process Example

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

// Global Variables
sem_t* semaphore;
pid_t otherPid;
sigset_t sigSet;

void  signalHandler1(int signum)
{
    printf("Caught Signal: %d\n",signum);
    printf("     Exit Child Process.\n");
    sem_post(semaphore);
    _exit(0);
}

void  signalHandler2(int signum)
{
    printf("I am alive!\n");
}

/**
 * Logic to run to simulare a Parent Process
 **/
void childProcess()
{
    // Setup some Signal Handlers
    signal(SIGUSR1, signalHandler1);
    signal(SIGUSR2, signalHandler2);

    // Child process: Simulates a hung process waiting on a semaphore (comment sem_post out) or running too long (comment sleep out
    int value;
    sem_getvalue(semaphore, &value);
    printf("     Child Process semaphore count is %d.\n", value);
    printf("     Child Process is grabbing semaphore.\n");
    sem_wait(semaphore);
    sem_getvalue(semaphore, &value);
    printf("     Child Process semaphore count is %d.\n", value);
    /* START CRITICAL REGION */
    printf("     Starting very long Child Process.....\n");
    for(int x=0;x < 60;++x)
    {
        printf(".\n");
        sleep(1);
    }
    /* END CRITICAL REGION */
    sem_post(semaphore);

    // Exit the Child Process (use _exit() and NOT exit()
    printf("     Exit Child Process.\n");
    _exit(0);
}
```

```c
void *checkHungChild(void *a)
{
    // Simulate a timer of 10 seconds by going to sleep, then check if semaphore is locked indicating a hung Child Process
    int* status = a;
    printf("Checking for hung Child Process.....\n");
    sleep(10);
    if (sem_trywait(semaphore) != 0)
    {
        printf("Child Process appears to be hung.....\n");
        *status = 1;
    }
    else
    {
        printf("Child Process appears to runnning fine.....\n");
        *status = 0;
    }
    return NULL;
}
```

```c
/**
 * Logic to run to simulare a Parent Process
 **/
void parentProcess()
{
    // Parent process: Detect hung Child Process and kill it after a timeout
    sleep(2);
    if(getpgid(otherPid) >= 0)
    {
        printf("Child Process is running.....\n");
    }
    int value;
    sem_getvalue(semaphore, &value);
    printf("In the Parent Process with the semaphore count of %d.\n", value);

    // Try to get semaphore (or could have used sem_getvalue since sem_trywait would block) and if it is locked then start a timer
    if (sem_trywait(semaphore) != 0)
    {
        // Start Timer Thread and wait for it to return
        pthread_t tid1;
        int status = 0;
        printf("Detected Child is hung or running too long.....\n");
        if (pthread_create(&tid1, NULL, checkHungChild, &status))
        {
            printf("ERROR creating timer thread.\n");
            _exit(1);
        }
        if(pthread_join(tid1, NULL))
        {
            printf("\n ERROR joining timer thread.\n");
            exit(1);
        }

        // See if we need to kill the Child Process
        if(status == 1)
        {
            // Kill Child Process
            printf("Going to kill Child Process with ID of %d\n", otherPid);
//            kill(otherPid, SIGUSR1);
            kill(otherPid, SIGTERM);
            printf("Killed Child Process\n");

            // Prove that the Child Process is killed
            printf("Now Proving Child Process is killed (you should see no dots and no response from SIGUSR2 signal\n");
            sleep(5);
            kill(otherPid, SIGUSR2);
            sleep(1);
            printf("Done proving Child Process is killed\n");

            // Try to get semaphore
            sem_getvalue(semaphore, &value);
            printf("In the Parent Process with the semaphore count of %d.\n", value);
            if (sem_trywait(semaphore) != 0)
            {
                if(value == 0)
                    sem_post(semaphore);
                printf("Cleaned up and finally got the semaphore.\n");
                sem_getvalue(semaphore, &value);
                printf("In the Parent Process with the semaphore count of %d.\n", value);
           }
            else
            {
                printf("Finally got the semaphore.\n");
            }
        }
    }

    // Exit the Parent Process (use _exit() and NOT exit()
    printf("Exit Parent Process.\n");
    _exit(0);
}
```

```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;

    // Create shared semaphore
    semaphore = (sem_t*)mmap(0,sizeof(sem_t), PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS, -1, 0);
    if(sem_init(semaphore, 1, 1) != 0)
    {
        printf("Failed to create semaphore.\n");
        exit(EXIT_FAILURE);
    }

    // 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.\n");
        exit(EXIT_FAILURE);
    }
    // OK: If fork() returns 0 then the child process is running
    if (pid == 0)
    {
        // Run Child Process logic
        printf("     Started Child Process with Process ID of %d.....\n", getpid());
        otherPid = getppid();
        childProcess();
    }
    else
    {
        // Run Parent Process logic
        printf("Started Parent Process with Process ID of %d.....\n", getpid());
        otherPid = pid;
        parentProcess();
    }

    // Cleanup
    sem_destroy(semaphore);

    // Return OK
    return 0;
}
```