Experiment #5
Multi-Threading Environment under Unix/Linux
Thread Management for Real-Time Applications

1. Prerequisite

ENCS 538, C programming language, basics of inter-process communications under Unix/Linux. Basic concepts about threads and pthreads.

2. Objectives

- How to create and terminate threads.
- How to establish communication between threads.
- How to synchronize, manage and schedule threads.
- How to protect shared resources and ensure data integrity is maintained.

3. Background

Code is often written in a serialized (or sequential) fashion. Ignoring instruction level parallelism (ILP), code is executed sequentially, one after the next in a monolithic fashion, without regard to possibly more available processors the program could exploit. Often, there are potential parts of a program where performance can be improved through the use of threads.

A computer program becomes a **process** when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a processor or a set of processors. A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth.

3.1What is thread?

A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel). Figure 1 shows that threads are within the same process address space, thus, much of the information present in the memory description of the process can be shared across threads.

3.2What are pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library, which you link with your program.

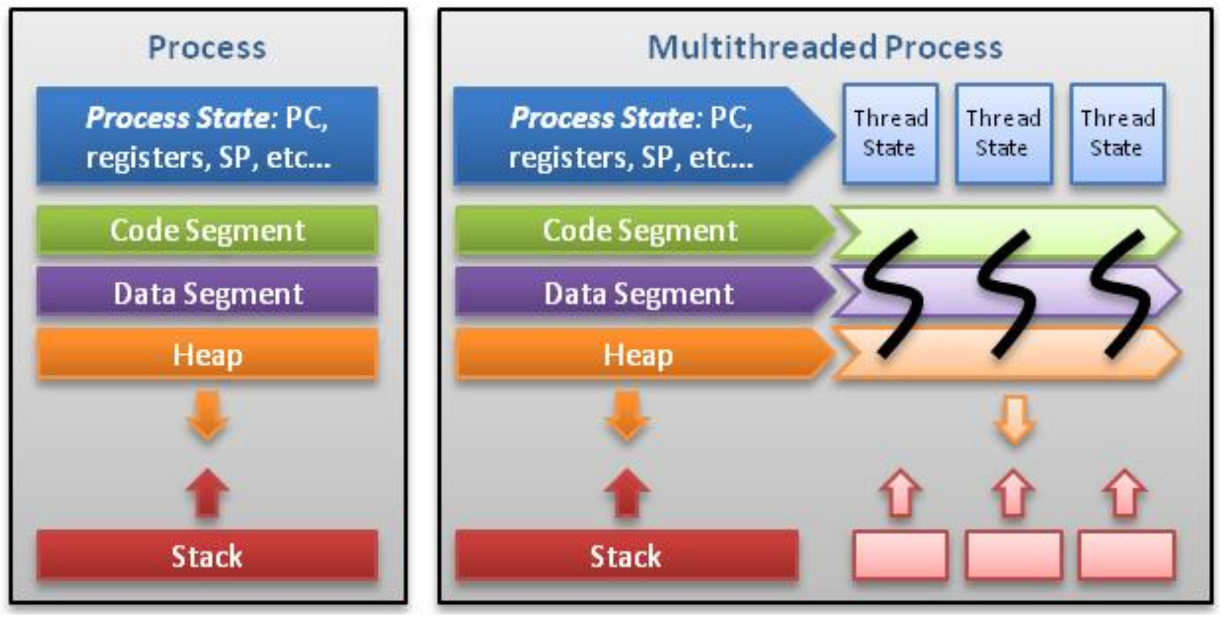

Figure 1 Process and Multithread Process Abstract View

3.3 pthreads APIs

The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

Thread management: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes and Race Conditions: The second class of functions deal with a coarse type of synchronization, called a "**mutex**", which is an abbreviation for "mutual exclusion". **Mutex** functions provide for creating, destroying, locking and unlocking **mutexes**. They are also supplemented by **mutex** attribute functions that set or modify attributes associated with **mutexes**.

Condition variables: The third class of functions deal with a finer type of synchronization - based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Thread Management:

The function **pthread_create** is used to create a new thread, and a thread to terminate itself uses the functions **pthread_exit** and **pthread_cancel**. A thread to wait for termination of another thread uses the function **pthread_join**.

- **Create Thread**

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,void
*(*start_routine)(void *), void *arg);
```

The required arguments for `pthread_create()`:

1. `pthread_t *thread`: the actual thread object that contains pthread id.
2. `pthread_attr_t *attr`: attributes to apply to this thread.
3. `void *(*start_routine)(void *)`: the function this thread executes.
4. `void *arg`: arguments to pass to thread function above.

The return value is 0 on success. The return value is negative on failure.

- **Exit Thread**

```
void pthread_exit(void *value_ptr);
```

The required arguments for `pthread_exit()`:

1. `void *value_ptr`: the return value is passed as a pointer.

- **Join Thread**

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The required arguments for `pthread_join()`:

1. `pthread_t thread`: the actual thread object that contains pthread id.
2. `void **value_ptr`: Return value is returned by ref.

The return value is 0 on success. The returned value is a pointer returned by reference.

- **Cancel Thread**

```
int  pthread cancel   (pthread t    thread )
```

pthread_cancel sends a cancellation request to the thread denoted by the thread argument.

If there is no such thread, pthread_cancel fails. Otherwise it returns 0.

- **Thread Identifiers**

```
int  pthread cancel   (pthread t    thread )
```

Returns the unique thread ID of the calling thread. The returned data object is opaque cannot be easily inspected.

```
pthread equal ( thread1, thread2 )
```

Compares two thread IDs: If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

Mutexes and Race Conditions:

Mutual exclusion locks (**mutexes**) can prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

Mutex Overview

- **Mutex** is a shortened form of the words "**mutual exclusion**".
- **Mutex** variables are one of the primary means of implementing thread synchronization.
- A **mutex** variable acts like a "**lock**" protecting access to a shared data resource. The basic concept of a **mutex** as used in **Pthreads** is that only one thread can lock (or own) a **mutex** variable at any given time. Thus, even if several threads try to lock a **mutex** only one thread will be successful. No other thread can own that **mutex** until the owning thread unlocks that **mutex**. Threads must "take turns" accessing protected data.
- Very often the action performed by a thread owning a **mutex** is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "**critical section**".
- When several threads compete for a **mutex**, the losers block at that call an unblocking call is available with "**trylock**" instead of the "**lock**" call.

Creating and Destroying Mutexes APIs:

pthread mutexes are created and destroyed through the following functions:

```
• int pthread_mutex_init(pthread_mutex_t *mutex, const
  pthread_mutexattr_t *mutexattr);
• int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

The **pthread_mutex_init()** function requires a **pthread_mutex_t** variable to operate on as the first argument. Attributes for the **mutex** can be given through the second parameter. To specify default attributes, pass NULL as the second parameter. The **pthread_mutex_destroy()** function shall destroy the **mutex** object referenced by **mutex**; the **mutex** object becomes, in effect, uninitialized. An implementation may cause **pthread_mutex_destroy()** to set the object referenced by **mutex** to an invalid value. A destroyed **mutex** object can be reinitialized using **pthread_mutex_init()**; the results of otherwise referencing the object after it has been destroyed are undefined.

Mutexes can be initialized to default values through a convenient macro rather than a function call:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

A mutex object named **lock** is initialized to the default pthread mutex values.

Locking and Unlocking Mutexes APIs:

To perform **mutex** locking and unlocking, the **pthreads** provides the following functions:

```
• int pthread_mutex_lock(pthread_mutex_t *mutex);
• int pthread_mutex_trylock(pthread_mutex_t *mutex);
• int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Each of these calls requires a reference to the **mutex** object. The difference between the lock and **trylock** calls is that lock is blocking and **trylock** is non-blocking and will return immediately even if gaining the **mutex** lock has failed due to it already being held/locked. It is absolutely essential to check the return value of the **trylock** call to determine if the **mutex** has been successfully acquired or not. If it has not, then the error code **EBUSY** will be returned.

Condition Variables:

Condition variables provide yet another way for threads to **synchronize**. While **mutexes** implement **synchronization** by controlling thread access to data, **condition variables** allow threads to **synchronize** based upon the actual value of data.

Without **condition variables**, the programmer would need to have threads continually **polling** (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

Creating and Destroying Condition Variables APIs:

pthread condition variables are created and destroyed through the following functions:

- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
- `int pthread_cond_destroy(pthread_cond_t *cond);`

Similar to the mutex initialization call, **condition variables** can be given non-default attributes through the second parameter. To specify defaults, either use the initializer macro or specify NULL in the second parameter to the call to pthread_cond_init(). The **pthread_cond_destroy()** function shall destroy the given condition variable specified by **cond**; the object becomes, in effect, uninitialized. An implementation may cause **pthread_cond_destroy()** to set the object referenced by **cond** to an invalid value. A destroyed condition variable object can be reinitialized using **pthread_cond_init();** the results of otherwise referencing the object after it has been destroyed are undefined.

Condition Variable can be initialized to default values through a convenient macro rather than a function call:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

A condition variable object named **cond** is initialized to the default pthread condition values.

Waiting and Destroying Condition Variables APIs:

Threads can act on condition variables in three ways: wait, signal or broadcast::

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- `int pthread_cond_signal(pthread_cond_t *cond);`
- `int pthread_cond_broadcast(pthread_cond_t *cond);`

pthread_cond_wait() puts the current thread to sleep. It requires a **mutex** of the associated shared resource value it is waiting on. **pthread_cond_signal()** signals one thread out of the possibly many sleeping threads to wakeup. **pthread_cond_broadcast()** signals all threads waiting on the **cond** condition variable to wakeup.

4. Procedure

4.1 Creating and joining pthreads

1. Type the following program that creates concurrent processes using the fork system call. Name the file **pfork.c**.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int shared_mem_id;
int *shared_mem_ptr;
int *r1p;
int *r2p;

extern int main(void)
{
    pid_t child1_pid, child2_pid;
    int status;

/* initialize shared memory segment */

    shared_mem_id = shmget(IPC_PRIVATE, 2*sizeof(int), 0660);
    shared_mem_ptr = (int *)shmat(shared_mem_id, (void *)0, 0);

    r1p = shared_mem_ptr;
    r2p = (shared_mem_ptr + 1);
    *r1p = 0;
    *r2p = 0;

    if ((child1_pid = fork()) == 0) {
            /* first child */

            do_one_thing(r1p);
            exit(0);
    }

/* parent */

    if ((child2_pid = fork()) == 0) {


    /* second child */
    do_another_thing(r2p);
          exit(0);
}

    /* parent */
    waitpid(child1_pid, &status, 0);
    waitpid(child2_pid, &status, 0);

    do_wrap_up(*r1p, *r2p);

    return 0;
    }

void do_one_thing(int *pnum_times)
{

    int i, j, x;

    for ( i = 0; i < 4; i++ )
    {
            printf("doing one thing\n");

    for ( j = 0; j < 10000; j++ )
            x = x + i;
            (*pnum_times)++;
    }
}

void do_another_thing(int *pnum_times)
{

    int i, j, x;

    for ( i = 0; i < 4; i++ ) {
            printf("doing another \n");

    for ( j = 0; j < 10000; j++ )

    x = x + i;
    (*pnum_times)++;
    }
}

void do_wrap_up(int one_times, int another_times)
{
    int total;
    total = one_times + another_times;

    printf("wrap up: one thing %d, another %d, total %d\n",one_times, another_times,
    total);
}
```

2. Compile **pfork.c** using the **gcc** compiler to create the executable **pfork** and run it. Notice how the processes run concurrently.

3. We would like to run concurrent threads using the POSIX **pthreads** instead of **fork** system calls. Type the following program that uses the functions **pthread** create and **pthread** join calls. Name the file **pcreate.c**.

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

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;

extern int main(void)
{
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL,(void *) do_one_thing,(void *) &r1);

    pthread_create(&thread2, NULL,(void *) do_another_thing,(void *) &r2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    do_wrap_up(r1, r2);

    return 0;
}
```

4. We would Compile the file **pcreate.c** using the **gcc** compiler to create the executable **pcreate** as follows:

```
> gcc pcreate.c -o pcreate -lpthread
```

   where lpthread is the pthreads library.

4.2 Threads Synchronization and Sharing Resources

1. Type the following program that uses a mutex to synchronize the access of two threads to a shared resource (variable **r3** in the example below). Name the file **pmutex1.c** and compile it using the **gcc** compiler.

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

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0, r3 = 0;

pthread_mutex_t r3_mutex=PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char **argv)
{
	pthread_t thread1, thread2;

	if ( argc != 2 )
	{
		fprintf(stderr, "You must supply an integer as argument!\n");
		exit(-1);
	}

	r3 = atoi(argv[1]);

	pthread_create(&thread1,NULL,(void *) do_one_thing,(void *) &r1);
	pthread_create(&thread2,NULL,(void *) do_another_thing,(void *) &r2);

	pthread_join(thread1, NULL);
	pthread_join(thread2, NULL);

	do_wrap_up(r1, r2);

	return 0;
}

void do_one_thing(int *pnum_times)
{
	int i, j, k, x;

	pthread_mutex_lock(&r3_mutex);

	if ( r3 > 0 )
	{
		x = r3;
		r3++;
	}
	else
	x = 1;

	pthread_mutex_unlock(&r3_mutex);

	for ( i = 0; i < 4; i++ )
	{
		printf("doing one thing\n");

		for ( j = 0; j < 10000; j++ )
		x = x + i;

		(*pnum_times)++;

		for ( k = 0; k < 1000000; k++ );
	}
}
void do_another_thing(int *pnum_times)
{
	int i, j, k, x;
```

```
        pthread_mutex_lock(&r3_mutex);

        if ( r3 > 0 )
        {
                x = r3;
                r3--;
        }
        else
        x = 1;

        pthread_mutex_unlock(&r3_mutex);

        for ( i = 0; i < 4; i++ )
        {
                printf("doing another \n");

                for ( j = 0; j < 10000; j++ )
                x = x + i;

                (*pnum_times)++;

                for ( k = 0; k < 1000000; k++ );
        }
}

void do_wrap_up(int one_times, int another_times)
{
        int total;

        total = one_times + another_times;

        printf("wrap up: one thing %d, another %d, total %d\n", one_times,
        another_times, total);
}
```

2. Notice how the functions **pthread_mutex_lock** and **pthread_mutex_unlock** have been used to limit the access to variable r3 to only one thread. Note also that if a thread cannot get a mutex since it is unavailable, that specific thread will suspend its execution until the mutex becomes available. Attention to deadlocks!!

4.3 Threads Identity

1. Type the following program that creates two threads and make use of the function call **pthread_self** and **pthread** equal. Name the file **pself.c** and compile it using the **gcc** compiler.

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

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0, r3 = 0;

pthread_mutex_t r3_mutex=PTHREAD_MUTEX_INITIALIZER;

pthread_t thread1, thread2;

int main(int argc, char **argv)
{
        pthread_create(&thread1,NULL,(void *) do_one_thing,(void *) &r1);
        pthread_create(&thread2,NULL,(void *) do_one_thing,(void *) &r2);

        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);

        return 0;
}

void do_one_thing(int *pnum_times)
{
        int k;
        pthread_t thread;

        while ( 1 ) {
            pthread_mutex_lock(&r3_mutex);

           thread = pthread_self();

          if ( pthread_equal(thread1, thread) )

          printf("Thread %d: %d\n", thread, ++r3);
          else

          printf("Thread %d: %d\n", thread, --r3);

          pthread_mutex_unlock(&r3_mutex);

          for ( k = 0; k < 10000000; k++ );
            }
}
```

2. Notice how the two threads access the same function **do_one_thing**. If the calling thread is **thread1**, the variable **r3** is incremented. Otherwise, the variable **r3** is decremented.

4.4 Exiting Threads and Return Status

1. Type the following program that makes use of the function **pthread** exit if it encounters a negative number supplied by the user. Name the file **pexit1.c** and compile it using the **gcc** compiler.

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

pthread_t thread;

static int arg;
static const int real_bad_error = -12;
static const int normal_error = -10;
static const int success = 1;

void * routine_x(void *arg_in)
{
        int *arg = (int *)arg_in;

        if ( *arg < 0 )

        pthread_exit((void *) &real_bad_error);

        else if ( *arg == 0 )
          return ((void *) &normal_error);

        else
          return ((void *) &success);
}

int main(int argc, char **argv)
{
        pthread_t thread;
        int r;
        void *statusp;

        if ( argc != 2 )
        {
                fprintf(stderr, "You must supply an integer as argument!\n");
                exit(-1);
        }

        r = atoi(argv[1]);

        pthread_create(&thread, NULL, routine_x, &r);

        pthread_join(thread, &statusp);

        if ( (int *) statusp == PTHREAD_CANCELED )
           printf("Thread was canceled.\n");
        else
           printf("Thread completed and exit status is %ld.\n", *(int *)statusp);

        return 0;
}
```

2. Run the program by supplying the number -10, 0 and 10 consecutively. Notice the exit status that you get each time.

4.5 Threads Synchronization - Condition Variables

1. Type the following program that uses a condition variable and waits on a signal. Name the file **pwait1.c** and compile it using the **gcc** compiler.

```
/*
* Using conditions variables for synchronization
*/
#include <stdio.h>
#include <pthread.h>

#define TCOUNT 10
#define WATCH_COUNT 12

int count = 0;

pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_threshold_cv = PTHREAD_COND_INITIALIZER;

int thread_ids[3] = {0,1,2};

int main(void)
{
    int i;

    pthread_t threads[3];

    void inc_count(int *idp);
    void watch_count(int *idp);

    pthread_create(&threads[2], NULL, watch_count, &thread_ids[2]);
    pthread_create(&threads[0], NULL, inc_count, &thread_ids[0]);
    pthread_create(&threads[1], NULL, inc_count, &thread_ids[1]);

    for ( i = 0; i < 3; i++ )
    {
            pthread_join(threads[i], NULL);
    }

    return 0;
}

void watch_count(int *idp)
{

    pthread_mutex_lock(&count_mutex);

    while ( count <= WATCH_COUNT )
     {
            pthread_cond_wait(&count_threshold_cv, &count_mutex);
            printf("watch_count(): Thread %d, Count is %d\n", *idp, count);
    }
```

```
        pthread_mutex_unlock(&count_mutex);
}

void inc_count(int *idp)
{
        int i;
        for ( i = 0; i < TCOUNT; i++ )
         {
                pthread_mutex_lock(&count_mutex);

                count++;
                printf("inc_count(): Thread %d, old count %d, new count %d\n",*idp,
        count - 1, count);

                if ( count == WATCH_COUNT )
                        pthread_cond_signal(&count_threshold_cv);

                pthread_mutex_unlock(&count_mutex);
        }

}
```

2. Type the Run the program and notice how the different threads behave. Notice mainly how thread 3 waits a signal to be delivered by either thread 1 or thread 2 in order to accomplish its duty.
3. Note that if multiple threads are waiting for a condition to become true, only one thread gets awakened by **pthread_cond_signal**. The order is usually dependent on the threads scheduling priority. If all waiting threads are of the same priority, they are released in a first-in first-out order for each **pthread_cond** signal call that's issued.