Threads

Fall 2016

What is a thread ?

- Creating a new process is a very expensive operation, mainly because it has to copy the entire memory space.
- To allow faster parallel programming threads were invented.
- A process might run few threads.
- Every such thread is an instance of the program **but** all the threads share the same memory.
- More specifically, when we create a new thread, the system creates a new process structure copies almost all the attributes from the original process that created the thread, **but** instead of creating a new memory space and copying the content of the memory from the original process to the new "process", the new "process" shares the memory with the original process.

What is a thread ? (con’t)

- Sometimes used the word "process" to describe the new thread that was created. That was because the system regards a thread just like a process.
- However it is easiest to think of a process to be composed of several threads.
- In this abstraction we think of a process as all the attributes that are common to all the threads.
- The threads within a process only keep the "pointer" that tell the CPU what **opcode** to perform next.

Writing threads in Linux

- Creating a thread in Linux is done with the function pthread_create.
- int pthread_create(pthread_t *thread, pthread_attr_t *attr, void * (*start_routine)(void *), void * arg);
- The function returns through the first parameter a number which is a unique id given to each thread created in the process.
- The second parameter sets the attributes of the thread to be different than the default. We will set this parameter to NULL. see the man page for more information
- The third parameter is a pointer to the function the thread will start to execute. Unlike fork that duplicates a process, that starts to run from the same place as the original, pthread_create creates a thread that starts running a given function.
- The forth parameter is the argument that will be passed to the function that the thread will run

Waiting for threads

- What happens when a multi-threaded program finishes the function main()?
- The program terminates.
- Unfortunately, when a program terminates its memory is destroyed.
- What happens to threads when the program terminates? Nothing.
- They keep running until they realize they don't have memory anymore (which happens very fast), then they crash.
- That is why main has to wait until all the threads it created finish before it finishes itself.
- Waiting for a thread to terminate is done with the function :

  int pthread_join(pthread_t *th*, void **thread_return);

  pthread_join waits until the thread whose id is *th* terminates. The second parameter is set by the os to be the value the thread return from the function it executes

Example

```
/*
        This is the Linux version of multi_thread
        The program demonstrates the use of threads and mutexex in Linux.
        We use the pthread library and must compile the program with
        lpthread.
*/

#include <pthread.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>

void* PrintFunc(void*);

pthread_t pids[2];
int pdata[2];
```

Example (con’t)

```
int main() {
                    /* preparing the data to pass to the threads */
pdata[0] = 0;
      pdata[1] = 1;

        /* creating two threads */
        pthread_create(& pids[0], NULL, & PrintFunc, & (pdata[1]));
        pthread_create(& pids[1], NULL, & PrintFunc, & (pdata[0]));

                    /* waiting for one thread to terminate */
        pthread_join(pids[1], NULL);
                    /* waiting for the other thread to terminate */
        pthread_join(pids[0], NULL);

        return 0;
}
```

Example (con’t)

```
	/* the thread receives as input the forth parameter passed to
	   pthread_create. In this case its the pointer to pdata[0] or
	   the pointer to pdata[1] */
void* PrintFunc(void* data) {
	int my_thread_num = *((int *)data);

	printf("%d\n", my_thread_num);

	return NULL;
}
```

What is mutual and what is not?

- Before answering the question
- First have to have a better understanding of the memory.
- We can generally say that the memory has two parts
- **Heap:** This is what you used to think of as memory. This part contains the code, memory we allocate and some variables.

What is mutual and what is not? (con't)

- **Stack:** This part of the memory keeps data about the execution of the code. Suppose we execute function a(), and a() calls b(). We have to keep somewhere the place in the code that called b(), because when we return from b() we have to know where to return to.
- If we called c() from b(), the place in b() has to be kept in addition to previous data we keep.
- This data is kept in a LIFO way. The last function that was called is the first function that will return. And that's why it is called **stack**.

What is mutual and what is not? (con't)

- It is obvious that threads cannot share their stacks, but they can share a heap.
- **How about local variables**
- Where are local variables kept?
- They have to be kept in the stack, otherwise, recursion would be impossible.
- Since local variables are kept in the stack, and the stacks are not mutual, it follows that local variables cannot be shared between threads.
- On the other hand: **static variables, allocated memory, global variables** are all kept on the heap, and therefore shared between all threads.

Threads Scenarios

- Consider these four pseudo code programs below in the next few slides.
- These programs use these two (non-existing) functions:
- createWindow() - A function that create a new window in which the process or thread prints all its messages.
- GetHrTime() - A function that returns a very fast timer. I.e. The function never returns the same value twice.
- For each of the programs, what is the output?

Pseudo code program #1

```
int time;

main() {
        int k = 0;
        id = fork();

        createWindow();
        time = GetHrTime();
        while (k < 100) {
                k++;
                printf("%d\n", time);
        }
```

- Answer: Two windows will be created. In each window one number (different in each window) will be printed 100 times

Pseudo code program #2

```
int time;

main() {
        createThread(,,PrintTime(),)
        createThread(,,PrintTime(),)
}

PrintTime() {
        int k = 0;

        createWindow();
        time = GetHrTime();
        while (k < 100) {
                k++;
                printf("%d\n", time);
        }
}
```

- Answer: Two windows will be created. two number will be printed t1, t2. In one window t1 will be printed 100 times. In the other window t2 will be printed then (maybe) t1 will be printed. In both window 100 numbers will be printed

Pseudo code program #3

```
main() {
        createThread(,,PrintTime(),)
        createThread(,,PrintTime(),)
}

PrintTime() {
        int time;
        int k = 0;

        createWindow();
        time = GetHrTime();
        while (k < 100) {
                k++;
                printf("%d\n", time);
        }
}
```

- Answer: Two windows will be created. In each window one number (different in each window) will be printed 100 times.

Pseudo code program #4

```
int k = 0;

main() {
        createThread(,,PrintTime(),)
        createThread(,,PrintTime(),)
}

PrintTime() {
        int time;

        createWindow();
        time = GetHrTime();
        while (k < 100) {
                k++;
                printf("%d\n", time);
        }
}
```

Answer: Two windows will be created. In every window one number will be printed. However the number of lines printed might be very high (anything between 100 and about 100*99/2). This happens because k is a global variable and the two threads access it with a non-atomic operation

Synchronizing threads

- As we have seen in the previous threads scenarios, when two threads try to increment the same variable at the same time, they may leave the variable in an incorrect condition.
- This is true for every resource the threads share (memory variables, screen, modem etc.).
- The parts of the code that access the shared resource are called **critical sections**.
- To avoid using the same resource at the same time, a thread should perform some check to see whether there is another thread in the critical section.
- If there is no thread in the critical the thread may enter the critical section. Otherwise, it has to wait until the critical section is free again.

Synchronizing threads (con’t)

- The mechanism that performs this check is called a mutex.
- A mutex has two states, that are usually referred to as unlocked and locked.
- An unlocked mutex indicates that the critical section is empty, and locked mutex indicates that there is some thread inside the critical section.
- A thread that wishes to access a resource checks the mutex associated with that resource.
- If the mutex is unlocked, it means there is no thread in the critical section, so the thread locks the mutex and enters the critical section. When the thread leaves the critical section it should unlock the mutex.
- If the mutex is locked, it means that there is another thread in the critical section, so the thread (that is trying to lock the mutex and enter) waits until the mutex becomes unlocked.

How do we make use of mutex

- There are two operations defined on a mutex (beside initializing and destroying).
- The first is an operation that checks the state of the mutex, and then it locks the mutex if it is unlocked, or waits until it becomes unlocked. This operation is usually called **lock**.
- The second operation unlocks the mutex, and it usually is called **unlock**.

How do we make use of mutex (con’t)

- The first question that should arise is what happens when two threads call lock at the same time.
- Seemingly, they could ruin the mutex.
- However, notice that the operating system performs the lock (it is a function of the OS), and we know that the operating system is the one that decides when to schedule a new thread.
- Combining this information we get an elegant solution:
- When executing the lock function the operating system does not allow rescheduling. More specifically, the code of lock disables interrupts. This way two lock functions cannot ruin the mutex.

How do we make use of mutex (con’t)

- The second question that should arise is how the OS checks that a locked mutex was unlocked. After all, checking every some period of time is very not efficient.
- The answer is, it doesn't!
- **The OS does not check whether a mutex is unlocked every some period of time!**
- Every mutex has a list of threads that are waiting for it to unlock.
- When a thread discovers a mutex is locked, it adds itself to the waiting list.
- When the mutex is unlocked, the OS chooses one of the threads in the waiting list of that mutex, and wakes this thread. The thread then will try to lock the mutex again.

Using MUTEX in Lunix

- **Defining and initializing a mutex**

  A mutex is defined with the type pthread_mutex_t, and it needs to be assigned the initial value: "PTHREAD_MUTEX_INITIALIZER";
- **Locking a mutex**

  A mutex is locked with the function: pthread_mutex_lock(pthread_mutex_t *mutex)).

  This function gets a pointer to the mutex it is trying to lock. The function returns when the mutex is locked, or if an error occurred. (a locked mutex is not an error, if a mutex is locked the function waits until it is unlocked).
- **Unlocking a mutex**

  A mutex is unlocked with the function: pthread_mutex_unlock(pthread_mutex_t *mutex)).

Example

```
/*
        This is the Linux version of multi_thread
The program demonstrates the use of threads and mutex in Linux.
        We use the pthread library and must compile the program with
-lpthread.
*/

#include <pthread.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>

#define MAX_THREADS 32
```

Example (con’t)

```
typedef struct {
        int thread_num;
} thread_data;

void* PrintFunc(void*);

pthread_mutex_t print_mutex = PTHREAD_MUTEX_INITIALIZER;

int thread_num = 0;

pthread_t pids[MAX_THREADS];
thread_data pdata[MAX_THREADS];
```

Example (con’t)

```
int main() {
        int i;

        thread_num = 5;

        for(i = 0; i < thread_num; ++i) {

                /* this line might not work
                pthread_create(&pids[i], NULL, &PrintFunc, &i);
                        because i might change until the thread starts to run */

                /* this line might not work
                pthread_create(&pids[i], NULL, &PrintFunc, &pids[i]);
                        because pids[i] might not be initialized before the thread
                        starts to tun */

                pdata[i].thread_num = i;
                pthread_create(&pids[i], NULL, &PrintFunc, &pdata[i]);
        }

        for(i = 0; i < thread_num; ++i)
                pthread_join(pids[i], NULL);

        pthread_mutex_destroy(&print_mutex);

        return 0;
}
```

Example (con’t)

```
void* PrintFunc(void* data) {
        int i, j;
        int my_thread_num = ((thread_data *)data)->thread_num;

        for (i = 0; i < 5; ++i) {

                pthread_mutex_lock(&print_mutex);

                for (j = 0; j < 40; ++j)
                        printf("%d", my_thread_num);
                printf("\n");

                pthread_mutex_unlock(&print_mutex);

        }

        return NULL;
}
```

Semaphore

- A semaphore is an extension of a mutex. A mutex allows one thread inside the critical section, a semaphore allows n threads in a critical section (when the number n is given as a parameter on the initialization).
- A semaphore is useful when a resource has more than one instance, and a mutex can be implemented by initializing a semaphore with the value 1.
- There are two operations defined on a semaphore (beside initializing and destroying).
- The first is an operation that checks the value of the semaphore, and then it decreases it by 1 if it is positive, or waits until it becomes positive. This operation is usually called **wait** or **decrease**.
- The second operation increases the value of the semaphore, and it usually is called **post** or **increase**.

Using Semaphore in Lunix

- **Defining and initializing a semaphore**

  A semaphore is defined with the type sem_t, and it need to be initialized with the function sem_init.
- **Waiting on a semaphore**

  The value of a semaphore is decreased with the function: sem_post. This function gets a pointer to the semaphore it is trying to decrease. The function returns when the semaphore's value is decreased, or if an error occurred. (a semaphore with value 0 is not an error, if a semaphore is 0 the function waits until it is positive).
- **Posting on a semaphore**

  A value of a semaphore is increased with the function: sem_post.

Notifying threads of events

- we already saw how to prevent two threads from entering the same critical section using a mutex.
- Now we are going to see another tool for synchronization.
- Many times we want to notify a thread that an event has occurred. For example, thread A decreases the value of an integer, and another thread B needs to wait until the integer reaches a certain value. In this case we want a mechanism to allow thread B to sleep, and to be notified by thread A that the value of the integer is small enough.
- There are three operations we want to perform:

  wait - wait until an event occurs.

  signal - notify one waiting thread that an even has occurred.

  broadcast - notify all waiting threads that an even has occurred.

Notifying threads of events (con’t)

- The **pthread** library supply a tool for this kind of synchronization. This tool is called condition-variable. Intuitively, we want to define a condition variable for every event that exists in our program.
- The three operations defined on condition-variables are:
- *wait* - blocks the thread.
- *signal* - wakes on thread that is waiting on the condition-variable
- *broadcast* - wakes all threads that are waiting on the condition-variable

Notifying threads of events (con’t)

- When we write code, one thread waits for an event to occur, and another thread signals that the event occurred.
- For example, let's look at the following pseudo-code program that defines an int.
- One thread is decrementing the integer i.
- The other thread is waiting until i reaches 0.

Pseudo code program

```
/* This code is not correct !!! */
/* details will be given soon */

int i = 4;
        /* We use c to signal when i reaches 0 */
condition_variable c;

void* waiting(void*) {
                /* wait until i = 0 */
        wait(&c);
        print("iii\n");
}

void* signaling(void*) {
        while (i > 0) {
                --i;
                if (i == 0) {
                                        /* signal that i reached 0 */
                        signal(&c);
                }
        }
}
```

Let’s see how threads wait for a signal

- Just like mutexes every condition variable has a list of threads that are waiting to be signaled.
- When a thread calls wait(& c) it adds itself to the waiting list and removes itself from the ready queue.
- When signal(& c) is called one thread is extracted from the waiting list and is returned to the ready queue

Basically the code for signal looks like the following :

```
signal(&c) {
    choose one thread from waiting list.
    remove the thread from the waiting list
(otherwise we might signal the same
thread twice)
}
```

The code for wait looks like the following:

```
/* This code is not correct !!! */
/* details will be given soon */

wait(&c) {
        add current thread to waiting list

        relinquish CPU (thus going to sleep)

        (the thread reaches this point after it is woken)
        check the waiting list, if the thread woke because of an error
        it should remove itself from the waiting list.
}
```

continue

- As you see both functions access the waiting list.
- We want to prevent two threads from accessing the same waiting at the same time.
- The solution chosen in pthread is to lock a mutex before calling signal or wait

So our code should look like the following

```
/* This code is not correct !!! */
/* details will be given soon */

int i = 4;
        /* We use c to signal when i reaches 0 */
condition_variable c;
mutex m;

void* waiting(void*) {
                /* wait until i = 0 */
        lock(&m);
        wait(&c);
        unlock(&m)
        print("iii\n");
}

void* signaling(void*) {
        while (i > 0) {
                --i;
                if (i == 0) {
                                        /* signal that i reached 0 */
                        lock(&m)
                        signal(&c);
                        unlock(&m)
                }
        }
}
```

continue

- However, this causes a new problem.
- The mutex m is locked when the thread calling wait goes to sleep, so how can any thread call signal, after all it cannot lock the mutex before that.
- To solve this problem we unlock the mutex inside the wait function.

So the wait function is carefully implemented like in the following

```
wait(&c, &m) {
    add current thread to waiting list

    unlock(&m);
    relinquish CPU (thus going to sleep)
    lock(&m);

    (the thread reaches this point after it is woken)
    check the waiting list, if the thread woke because of an error
it should remove itself from the waiting list.
}
```

continue

- **What happens when a thread signals on a conditional variable and there is no thread currently waiting?**
- A signal is not preserved. If one thread signals on a condition variable and no thread is waiting at that moment, the signal "goes away" and when a thread waits on the same condition variable it does not catch the previous signal, and has to wait for a new signal.

Example

```
#include <pthread.h>

int i = 9999999;
        /* We use c to signal when i reaches 0 */
pthread_cond_t c;
pthread_mutex_t m;

void* waiting(void*a) {
                    /* wait until i = 0 */
        pthread_mutex_lock(&m);
        pthread_cond_wait(&c, &m);
        pthread_mutex_unlock(&m);
        printf("iii\n");
}
```

Example (con’t)

```
void* signaling(void*a) {
        while (i > 0) {
                --i;
                if (i == 0) {
                                /* signal that i reached 0 */
                        pthread_mutex_lock(&m);
                        pthread_cond_signal(&c);
                        pthread_mutex_unlock(&m);
                }
        }
}
```

Example (con’t)

```
main() {
        pthread_t tid[2];

        pthread_create(&tid[0], NULL, &waiting, NULL);
        pthread_create(&tid[1], NULL, &signaling, NULL);

        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);

        pthread_mutex_destroy(&m);
        pthread_cond_destroy(&c);
}
```

Example of implementing a semaphore using a mutex and a condition variable.

```
struct semaphore {
        mutex m;
        cond_var c;
        int n;
}

wait(semaphore &s) {
        lock(&(s->m));

                /* wait until n > 0 */
        while (n == 0)
                wait(&(s->c), &(s->m));

        --n;

        unlock(&(s->m));
}

post(semaphore &s) {
        lock(&(s->m));

        ++n;
        signal(&(s->c));
        unlock(&(s->m));
}
```

continue

- If the while loop in the previous example is replaced with an if statement (a very common mistake), the following scenario will lead to an error:
- suppose we start with n=1
- Thread A decreases n. Now n=0
- Thread B tries to decrease n, and is now waiting
- Thread A increases n, and sends a signal that wakes B
- Thread C decreases n (before B actually gets CPU)
- Thread B gets CPU, and since we don't have a while statement decreases n to be -1

Compiling with threads

- When programing with threads you need to link your program with libthread.
- Add "-lpthread" to your link command

Useful pthread functions

- pthread_create - create a new thread
- pthread_equal - compare two thread identifiers
- pthread_exit - terminate the calling thread
- pthread_cancel - thread cancellation
- pthread_join - wait for termination of another thread
- pthread_self - return identifier of current thread

Useful pthread functions (con’t)

- pthread_mutex_destroy - deletes a mutex
- pthread_mutex_init - initializes a mutex
- pthread_mutex_lock - lock
- pthread_mutex_trylock - try to lock. Does not block
- pthread_mutex_unlock - unlock

Useful pthread functions (con’t)

- pthread_cond_broadcast - signal all threads that are waiting
- pthread_cond_destroy - operations on conditions
- pthread_cond_init - operations on conditions
- pthread_cond_signal - signal one thread
- pthread_cond_timedwait - wait with timeout
- pthread_cond_wait - wait for signal

Useful advanced pthread functions

- pthread_cleanup_pop, pthread_cleanup_push - install and remove cleanup handlers
- pthread_condattr_destroy - condition creation attributes
- pthread_condattr_init - condition creation attributes
- pthread_key_create - management of thread-specific data
- pthread_key_delete - management of thread-specific data
- pthread_getspecific - management of thread-specific data
- pthread_kill - handling of signals in threads
- pthread_kill_other_threads_np - terminate all threads in program except calling thread
- pthread_once - once-only initialization
- Type **"man -k pthread"** to get all functions related to threads.