Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 4:

- Multithreaded Programming with Pthreads

Creating a Thread

Initially, a program consists of a ***single thread*** of execution, often referred to as the ***main thread***.

If we want to spawn ***additional threads*** from the ***main thread***, we need to explicitly create each new thread using the $pthread\_create()$ function:

```
#include <pthread.h>

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

Returns 0 on success, or a positive error number on error

Creating a Thread

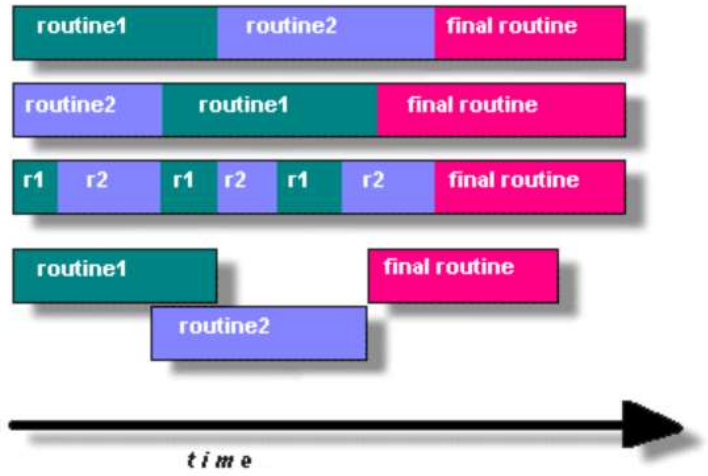
Parameters:

- ***thread*** : pointer to an ***opaque*** object of type $pthread\_t$ into which the unique identifier for the newly created thread is copied before the $pthread\_create()$ function returns
  - On many Unix variants including Linux, $pthread\_t$ is defined as $unsigned\ long$.
  - However, to make our code portable across all Unix variants, we should not assume so.
- ***attr*** : pointer to an ***attributes*** object of type $pthread\_attr\_t$ that specified a number of attributes for the newly created thread
  - With $NULL$, the new thread is created with the default attributes.
- ***start*** : function pointer to the routine to be executed by the new thread once its execution starts
  - The routine must have the following ***signature*** : $void\ *start(void\ *)$.
- ***arg*** : pointer to an object to be used as an argument to the function pointed to by ***start***
  - We may ***cast arg*** into any type in C/C++.

Creating a Thread

After a call to $pthread\_create()$, a program has no guarantees about which thread will be scheduled next on the CPU and, on a multiprocessor system, both threads may simultaneously execute on different CPUs.

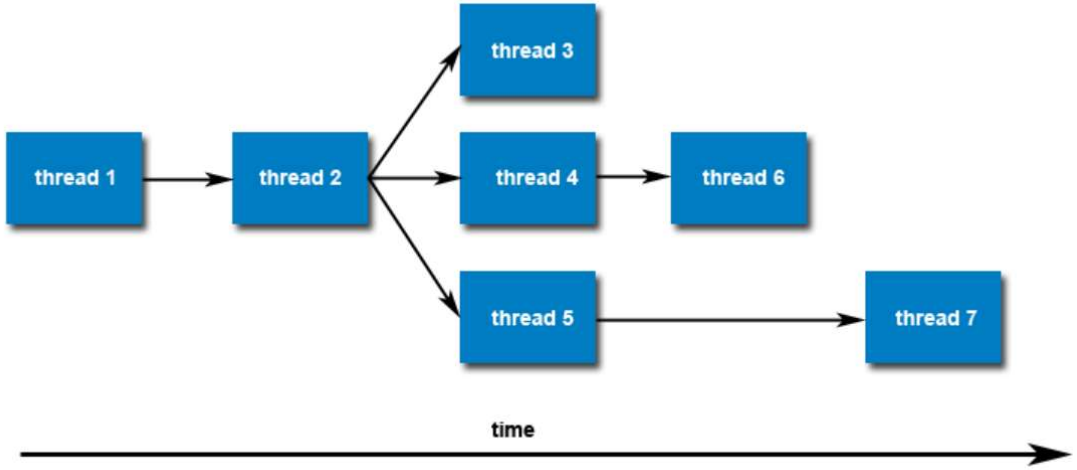
- The ***OS scheduler*** decides which thread will be scheduled next.



Creating a Thread

Threads do not form a parent-child hierarchy as in the case of ***processes***, which are created via the ***fork*** system call.

- In other words, once created, threads are ***peers*** and can also creates other threads.



Creating a Thread

Live Demo:

Go to Pthreads/ThreadCreate

Creating a Thread

Live Demo:

Go to Pthreads/ArgumentPassing

Terminating a Thread

The execution of a thread ***terminates*** in one of the following ways:

- The thread's ***start*** routine returns.
- The thread calls $pthread\_exit()$.
- The thread is cancelled via $pthread\_cancel()$.
- Any of the threads calls $exit()$ or the main function calls $return$.

Parameter:

- ***retval*** : pointer to the return value, which can be obtained in another thread by calling $pthread\_join()$.

```
include <pthread.h>

void pthread_exit(void *retval);
```

Terminating a Thread

The value pointed to by ***retval*** should not be located on the thread's stack since the contents of the stack become ***undefined*** on thread termination.

If the ***main thread*** calls $return$ or $exit()$, the ***entire process*** terminates.

- This means that if the main threads finishes before the threads it has created, the other threads will be ***automatically*** terminated.

If the ***main thread*** terminates via $pthread\_exit()$, the other threads can continue executing.

Terminating a Thread

Live Demo:

Go to Pthreads/ThreadExit

Joining a Thread

The $pthread\_join()$ function ***waits*** for the thread with the specified ***thread*** identifier to terminate:

- If the thread has already terminated, $pthread\_join()$ returns immediately.

Parameters:

- ***thread*** : thread identifier object of the thread to be joined
- ***retval*** : double pointer for retrieving the return value of the specified thread

```
include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
                                Returns 0 on success, or a positive error number on error
```

Joining a Thread

A call to $pthread\_join()$ ***blocks*** the calling thread until either one of the following circumstances:

- The specified thread returns from its ***start*** routine.
- The specified thread calls $pthread\_exit()$.
- The specified thread is ***cancelled*** via $pthread\_cancel()$.
  - If the thread was ***cancelled***, the memory location specified by ***retval*** is set to $PTHREAD\_CANCELED$.

By calling $pthread\_join()$, we automatically place the thread with which we are joining in the ***detached state*** so that its resources can be released.

- By default, all threads are created ***joinable***.
- If the thread was already in the ***detached state***, $pthread\_join()$ can ***fail***, returning $EINVAL$.

If we are not interested in the thread's return value, we can set ***retval*** to $NULL$ in $pthread\_join()$.

Joining a Thread

Live Demo:

Go to Pthreads/ThreadJoin

Estimate Pi using Monte Carlo

Live Demo:

Go to Pthreads/Pi

Cancelling a Thread

One thread in a process can request that another thread ***within the same process*** be canceled by calling $pthread\_cancel()$:

- A call to $pthread\_cancel()$ does not wait for the specified thread to terminate; it only makes the request.
- In the ***default circumstances***, $pthread\_cancel()$ will cause the thread specified by ***tid*** to behave as if it had called $pthread\_exit()$ with an argument of $PTHREAD\_CANCELED$.

Parameter:

- ***tid*** : thread identifier object of the thread specified to be canceled

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Returns: 0 if OK, error number on failure

Detaching a Thread

By default, a thread's termination status is retained until the thread is joined via a call to $pthread\_join()$.

- On the contrary, the underlying resources of a thread in the ***detached state*** can be released immediately on termination.

- When a thread is detached, we cannot call $pthread\_join()$ as we will risk encountering ***undefined behavior***.

Parameter:

- ***tid*** : thread identifier object of the thread specified to be detached

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, error number on failure

Detaching a Thread

Live Demo:

Go to Pthreads/ThreadDetach

Mutex : Concept

Shared data in a critical region can be protected using Pthreads' ***mutual exclusion interfaces***:

- A mutex of type $pthread\_mutex\_t$ is a lock that a thread must ***acquire*** before it can access shared data and ***release*** when the thread is done accessing the data.
- While a lock's value is set, any thread that attempts to acquire the mutex will be blocked until the ***owner thread*** releases it.
- If more than one thread is blocked when the owner thread releases its mutex, then all threads blocked on the mutex will wake up, and one of them will successfully acquire the mutex while the others go back to sleep.
- An attempt to release a lock owned by a different thread can result in ***undefined behavior***.

Mutex: Initialization

Before we can use a mutex, we must initialize it first.

There are ***two ways*** to initialize a mutex, depending on whether the mutex is a ***statically allocated*** object or a ***dynamically allocated*** object:

- For a ***statically allocated*** mutex, we can use the $PTHREAD\_MUTEX\_INITIALIZER$ macro to initialize the mutex with the ***default behavior*** as follows:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

- For a ***dynamically allocated*** mutex, we can use the $pthread\_mutex\_init()$ function.

An attempt to initialize an ***already initialized mutex*** can result in ***undefined behavior***.

After a successful initialization, the state of a mutex becomes ***initialized*** and ***unlocked***.

Mutex: Initialization

Parameters:

- ***mutex*** : pointer to a mutex of the type $pthread\_mutex\_t$ to be initialized

- ***attr*** : pointer to a Pthreads attributes object of the type $pthread\_mutexattr\_t$ that has been initialized

  - Pass $NULL$ to initialize the mutex with ***default attributes***.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
                                    Returns 0 on success, or a positive error number on error
```

Mutex: Initialization

Among the cases where we must use $pthread\_mutex\_init()$ rather than the static initializer macro are the following:

- The mutex was ***dynamically allocated*** on the ***heap*** via the $malloc()$ routine or the new operator.
- The mutex is an ***automatic variable*** on the ***stack***.
- We want to initialize a ***statically allocated mutex*** with attributes other than the default ones.

Mutex: Uninitialization

When an automatically or dynamically allocated mutex is no longer required, it should be destroyed using the $pthread\_mutex\_destroy()$ function:

- An attempt to destroy an ***unlocked mutex*** can result in ***undefined behavior***.
- A destroyed mutex can be subsequently ***reinitialized*** using the $pthread\_mutex\_init()$ function.
- If a mutex is located on the ***heap***, it should be destroyed before ***freeing*** that memory region with the $free()$ routine or the $delete$ operator.
- It is ***unnecessary*** to destroy a statically allocated mutex initialized with the macro.

Mutex: Uninitialization

Parameters:

- ***mutex*** : pointer to a mutex of the type $pthread\_mutex\_t$ to be destroyed

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Returns 0 on success, or a positive error number on error

Mutex: Init & Destroy

Live Demo:

Go to Pthreads/MutexInitDestroy

Mutex : Types

There are ***three types*** of mutexes available in Pthreads:

- $PTHREAD\_MUTEX\_NORMAL$
  - If a thread tries to lock a mutex that it has already locked, then a deadlock occurs.
- $PTHREAD\_MUTEX\_RECURSIVE$
  - A recursive mutex maintains the concept of a ***lock count***.
  - When a thread acquires a mutex, its lock count is set to 1.
  - Each subsequent locking operation by the same thread increments the lock count.
  - Each unlocking operation decrements the lock count by 1.
- $PTHREAD\_MUTEX\_ERRORCHECK$
  - It checks for deadlock conditions that occur when a thread reacquires a mutex that it has already locked.
  - On a deadlock, the locking operation fails with the $EDEADLK$ error.

Mutex : Types

```
pthread_mutex_t mtx;
pthread_mutexattr_t mtxAttr;
int s, type;

s = pthread_mutexattr_init(&mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_init");

s = pthread_mutexattr_settype(&mtxAttr, PTHREAD_MUTEX_ERRORCHECK);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_settype");

s = pthread_mutex_init(mtx, &mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutex_init");

s = pthread_mutexattr_destroy(&mtxAttr);        /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_mutexattr_destroy");
```

Mutex: Locking & Unlocking

After ***initialization***, a mutex is ***unlocked***.

- To lock and unlock a mutex, we use the $pthread\_mutex\_lock()$ and $pthread\_mutex\_unlock()$ functions, respectively.

Parameters:

- ***mutex*** : pointer to the mutex to be locked or unlocked

```
#include <pthread.h>

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

Both return 0 on success, or a positive error number on error

Mutex: Locking & Unlocking

Locking & Unlocking Semantics:

- If a thread attempts to lock a mutex via $pthread\_mutex\_lock()$ and the mutex is currently unlocked, the thread successfully acquires the lock and the call to $pthread\_mutex\_lock()$ returns immediately.

- If a thread attempts to lock a mutex via $pthread\_mutex\_lock()$ but the mutex is currently locked by another thread, the call to $pthread\_mutex\_lock()$ blocks until the mutex is unlocked.

- If the calling thread itself has already locked the mutex given to $pthread\_mutex\_lock()$, then, for the normal type of mutex, then one of two implementation-defined properties may result:
  a) The thread ***deadlocks***, blocked trying to acquire a mutex it already owns: On Linux, the thread deadlocks by default.
  b) The call fails, returning the $EDEADLK$ error.

Mutex: Locking & Unlocking

Locking & Unlocking Semantics:

- A call to $pthread\_mutex\_unlock()$ unlocks the specified mutex previously locked by the calling thread itself.
- It is an error to unlock a currently unlocked mutex.
- It is an error to unlock a mutex currently locked by another thread.
- If more than one thread is waiting to acquire a mutex unlocked by a call to $pthread\_mutex\_unlock()$, it is indeterminate which thread will succeed in acquiring it.

Mutex: Locking Variants

The Pthreads API provides ***two other variants*** of the $pthread\_mutex\_lock()$ function.

- The $pthread\_mutex\_trylock()$ function is the same as the $pthread\_mutex\_lock()$ function, except that if the mutex is currently locked, a call to $pthread\_mutex\_trylock()$ fails and ***immediately returns*** the $EBUSY$ error.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

All return: 0 if OK, error number on failure

Mutex: Locking Variants

- The $pthread\_mutex\_timedlock()$ function is the same as the $pthread\_mutex\_lock()$ function, except that the caller can specify ***one additional argument***, ***tsptr***, that places a limit on the amount of time that the thread will be blocked while waiting to acquire the mutex.
  - If the timeout interval specified by its ***tsptr*** argument expires without the caller becoming the owner of the mutex, the call returns $ETIMEDOUT$ the error.
  - The timeout interval specified how long we are willing to wait in terms of ***absolute time***.

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

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict tsptr);
```

Returns: 0 if OK, error number on failure

Condition Variable

Condition variables are another synchronization mechanism available in the Pthreads API, where they are represented by the $pthread\_cond\_t$ data type:

- They provide a place for threads to rendezvous.
- When used with a mutex, a condition variable allows one thread to inform other threads about changes in ***the condition state of shared variables*** and allows other threads to block and wait for such conditions to occur.
- The condition itself is protected by a mutex, that is, a thread must first acquire the mutex to change the condition state.

Condition Variable: Initialization

As with mutexes, before a condition variable is used, it must first be initialized and they can be initialized in two ways:

- A ***statically allocated*** condition variable can be initialized with the $PTHREAD\_COND\_INITIALIZER$ macro.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- A ***dynamically allocated*** condition variable can be initialized with the $pthread\_cond\_init()$ function.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

Returns 0 on success, or a positive error number on error

Condition Variable: Initialization

Parameters:

- ***cond*** : pointer to the condition variable to be initialized
- ***attr*** : pointer to to a Pthreads attributes object of the type $pthread\_condattr\_t$ that has been initialized
  - Pass $NULL$ to initialize the condition variable with ***default attributes***.

The circumstances where we need to use $pthread\_cond\_init()$ are analogous to those where $pthread\_mutex\_init()$ is needed to dynamically initialize a mutex:

- We need $pthread\_cond\_init()$ to initialize automatic condition variables (on the ***stack***).
- We need $pthread\_cond\_init()$ to initialize statically allocated condition variables (on the ***heap***).
- We need $pthread\_cond\_init()$ to initialize a statically allocated condition variable with attributes ***other than the defaults***.

Caveat: Initializing an already initialized condition variable results in ***undefined behavior***.

Condition Variable: Uninitialization

When an automatically or dynamically allocated condition is no longer required, it should be destroyed using the $pthread\_cond\_destroy()$ function:

- A destroyed condition variable can be subsequently ***reinitialized*** using the $pthread\_cond\_init()$ function.
- An automatic condition variable (on the ***stack***) should be destroyed before its function returns.
- If a condition variable is located on the ***heap***, it should be destroyed before ***freeing*** that memory region with the $free()$ routine or the $delete$ operator.
- It is ***unnecessary*** to destroy a statically allocated condition initialized with the macro.
- It is safe to destroy a condition variable only when no thread is waiting on it.

Condition Variable: Uninitialization

Parameters:

- ***cond*** : pointer to a condition variable of the type $pthread\_cond\_t$ to be destroyed

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
                                    Returns 0 on success, or a positive error number on error
```

Condition Variable: Signaling & Waiting

The two basic operations on condition variables are ***signal*** and ***wait***.

- The ***signal*** operation is a notification to one or more waiting threads that the state of some shared variables has changed.
- The ***wait*** operation is the means of blocking until such a notification is received.

The Pthreads API provides ***three basic functions*** for the ***signal*** and ***wait*** operations:

- $pthread\_cond\_signal()$
- $pthread\_cond\_broadcast()$
- $pthread\_cond\_wait()$

```
#include <pthread.h>

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

All return 0 on success, or a positive error number on error

Condition Variable: Signaling & Waiting

Signal & Wait Semantics:

- $pthread\_cond\_signal()$ and $pthread\_cond\_broadcast$
  - Both signal the condition variable specified by ***cond***.
  - With $pthread\_cond\_signal()$, one of the threads waiting for the condition specified by ***cond*** is woken up.
  - With $pthread\_cond\_broadcast()$, all of the waiting threads waiting for the condition specified by ***cond*** are woken up.
- $pthread\_cond\_wait()$
  - The $pthread\_cond\_wait()$ function blocks the calling thread until the condition variable ***cond*** is signaled.

Condition Variable: Signaling & Waiting

A ***condition variable*** holds no state information.

- If no thread is waiting on the condition variable at the time that it is signaled, then the signal is ***lost***.
- A thread that later waits on the condition variable will unblock only when the variable is signaled once more.

The use of a ***condition variable*** must always be associated with a ***mutex***.

- The thread locks the mutex in preparation for checking the state of the shared variable.
- The state of the shared variable is checked.
- If the shared variable is not in the desired state, then the thread must unlock the mutex before it goes to sleep on the condition variable.
- When the thread is woken up again because the condition variable has been signaled, the thread must once more be locked since, typically, the thread then immediately accesses the shared variable.

Condition Variable: Signaling & Waiting

Live Demo:

Go to Pthreads/BBProblem

Condition Variable: Signaling & Waiting

Each condition variable has an associated ***predicate*** that involves one or more shared variables.

- In the example during the last live demon, the predicate $avail == 0$ is associated with the condition variable $cond$.
- One design principle is that $pthread\_cond\_wait()$ must be governed by a ***while loop*** rather than an ***if statement***.
  - This is because, on return, from $pthread\_cond\_wait()$, there are no guarantees about the state of the predicate.
  - Therefore, we should immediately recheck the predicate and resumes sleeping if the predicate is not in the desire state.

Condition Variable: Signaling & Waiting

We cannot make any assumptions about the state of the predicate upon return from $pthread\_cond\_wait()$ for the following reasons:

- Other threads may be woken up first:
  - More than one thread was probably waiting for the mutex associated with the condition variable.
  - Even if the thread that signaled the condition variable set the predicate to the desired state, it is still possible that another thread might acquire the mutex first and change the state of the associated shared variable(s), hence the state of the predicate.
- Designing for loose predicate may be simpler:
  - Sometimes it is easier to design applications based on condition variables that indicate "***possibility***" rather than "***certainty***".
  - In other words, signaling a condition variable would mean "***there might be something***" for the signaled thread to do, rather than "***there is something to do***".
- ***Spurious wake-ups*** can occur:
  - On some implementations, a thread waiting on a condition variable might be woken up even though no other thread actually signaled the condition variable.