BBT

CPP

Multithreads

BBT

Basic Multithreading Concepts

- Concurrency and Parallelism
- Looking at Multithreading Structure
- Scheduling
- Cancellation

BBT

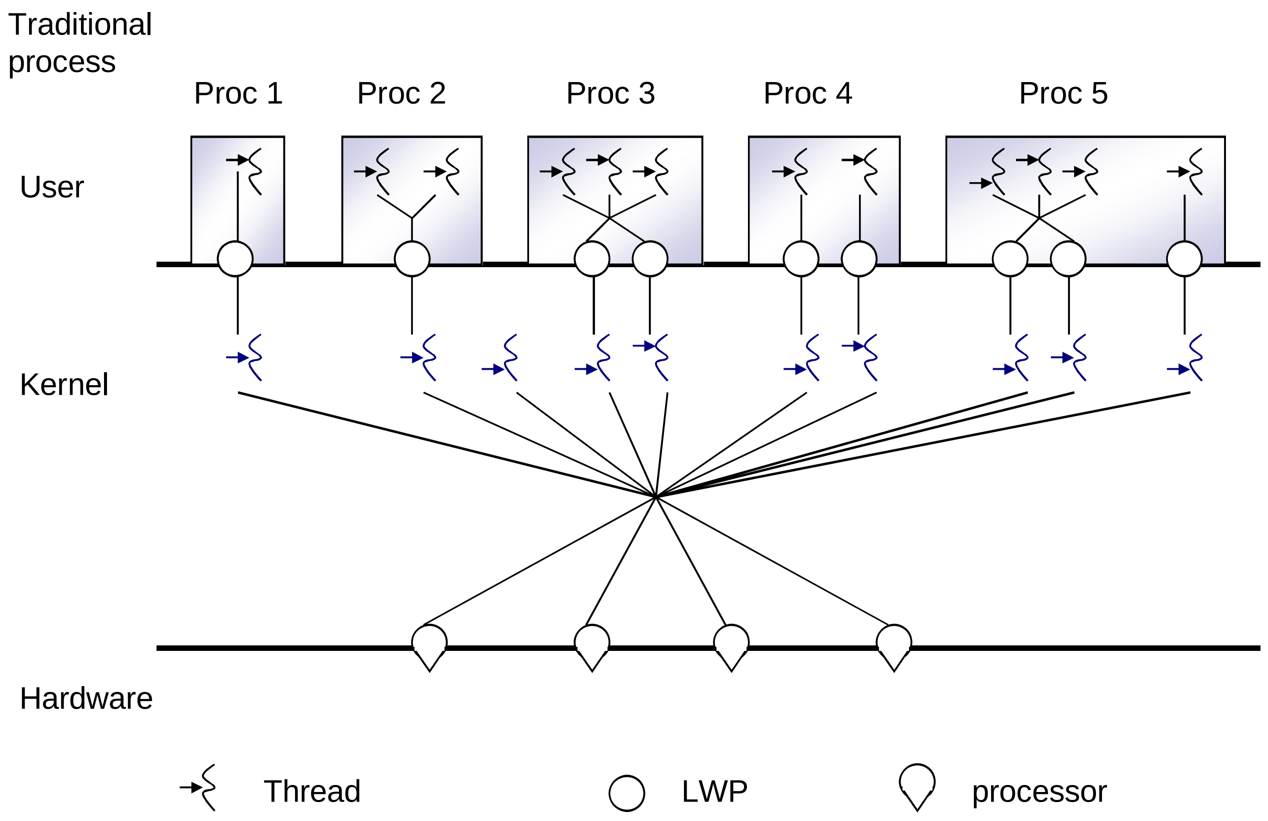
User-Level Threads

- Thread ID
- Register state (including PC and stack pointer)
- Stack
- Signal mask
- Priority
- Thread-private storage

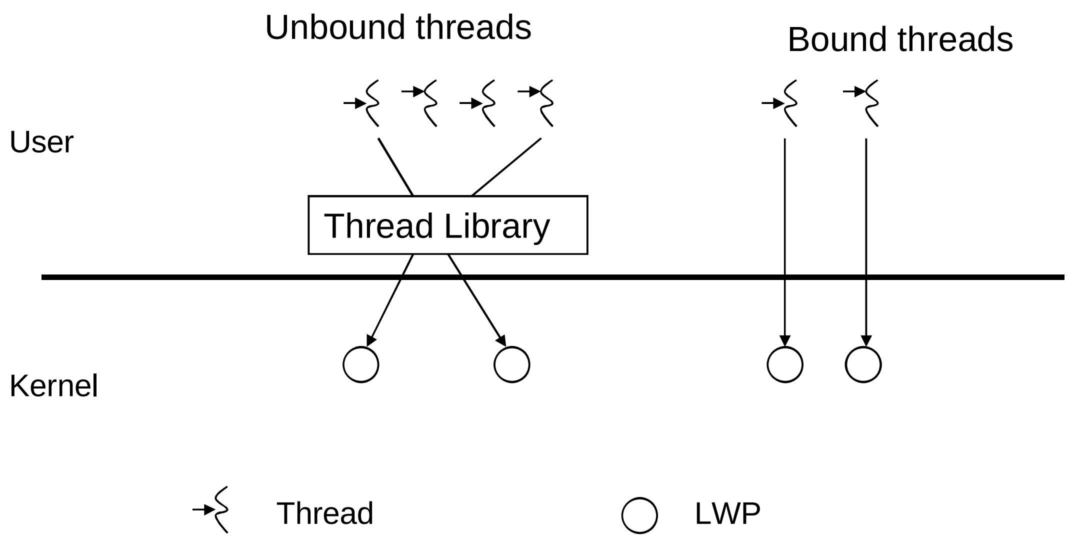
BBT

what is it good for

- Inexpensive to create because they do not need to create their own address space. They are bits of virtual memory that are allocated from your address space at run time.
- Fast to synchronize because synchronization is done at the application level, not at the kernel level.
- Easily managed by the threads library; either libpthread or libthread



Two-level thread model (M on N)

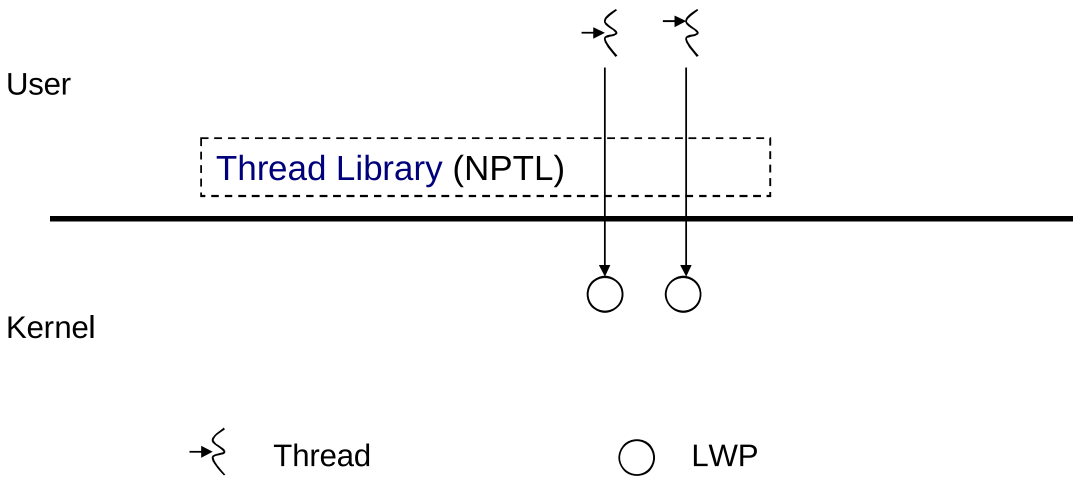


Lightweight Processes

- The threads library uses underlying threads of control called **lightweight processes (LWP)** that are supported by the kernel.
- every LWP has a corresponding kernel thread.
- Not every kernel thread has a corresponding LWP.
  - The kernel itself is multithreaded and will create kernel threads that perform various operating-system-specific tasks —memory management, STREAMS queue processing, scheduling, etc.— that execute as kernel threads not bound to a user process.

Linux NPTL (1 on 1)

NPTL: Native POSIX Thread Library



BBT

Scheduling

■ **SCHED_FIFO**	first-in-first-out	
■ **SCHED_RR**	round-robin	
■ **SCHED_OTHER**	Custom	$\leftarrow$ default

BBT

Schedule scope

- **PTHREAD_SCOPE_PROCESS**
  - Process Scope (Unbound Threads)
- **PTHREAD_SCOPE_SYETEM**
  - System Scope (Bound Threads)

BBT

Create a Default Thread

- Unbound
- Nondetached
- With a default stack and stack size
- With the parent's priority

BBT

pthread_create

int pthread_create(pthread_t ****tid*, const pthread_attr_t ****tattr***, void*(****start_routine***)(void *), void ****arg***);**

Example:

```
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
 int ret;
/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);
```

Wait for Thread Termination

int pthread_join (thread_t *tid*, void ***status*);

- suspends execution of the calling thread until the target thread terminates

Detaching a thread

int pthread_detach(thread_t *tid*);

- This guarantees that the memory resources consumed by th will be freed immediately when thread terminates

Terminate a thread

- **void pthread_exit(void *_status_);**

Synchronization

- Using Mutual Exclusion Locks
- Using Condition Variables
- Semaphores
- Using Read-Write Locks

BBT

Mutex attributes

- int pthread_mutexattr_init(pthread_mutexattr_t **mattr*);
- int pthread_mutexattr_destroy(pthread_mutexattr_t **mattr)*

- int pthread_mutexattr_setpshared(pthread_mutexattr_t **mattr*, int *pshared*);
- int pthread_mutexattr_getpshared(pthread_mutexattr_t **mattr*, int **pshared*);

- int pthread_mutexattr_settype(pthread_mutexattr_t *attr , int type);
- int pthread_mutexattr_gettype(pthread_mutexattr_t *attr , int *type);

BBT

Mutex scope

- PTHREAD_PROCESS_SHARED
- PTHREAD_PROCESS_PRIVATE

BBT

Mutex types

- **PTHREAD_MUTEX_NORMAL**
  - This type of mutex does not detect deadlock. An attempt to relock this mutex without first unlocking it deadlocks. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.
- **PTHREAD_MUTEX_ERRORCHECK**
  - An attempt to relock this mutex without first unlocking it returns with an error. An attempt to unlock a mutex that another thread has locked returns with an error. An attempt to unlock an unlocked mutex returns with an error.

BBT

Mutex types –cont.

- **PTHREAD_MUTEX_RECURSIVE**
  - A thread attempting to relock this mutex without first unlocking it succeeds in locking the mutex. The relocking deadlock that can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. An attempt to unlock a mutex that another thread has locked returns with an error. An attempt to unlock an unlocked mutex returns with an error.
- **PTHREAD_MUTEX_DEFAULT**

BBT

Routines for mutal Exclusion locks

- **pthread_mutex_init**
  - Initialize a Mutex
- **pthread_mutex_lock**
  - Lock a Mutex
- **pthread_mutex_unlock**
  - Unlock a Mutex
- **pthread_mutex_trylock**
  - Lock with a Nonblocking Mutex
- **pthread_mutex_destroy**
  - Destroy a Mutex

BBT

Example:

```
#include <pthread.h>
 pthread_mutex_t count_mutex;
 long count;
 void increment_count()
{
      pthread_mutex_lock(&count_mutex);
      count = count + 1;
      pthread_mutex_unlock(&count_mutex);
 }
long get_count()
 {
      long c;
      pthread_mutex_lock(&count_mutex);
      c = count;
      pthread_mutex_unlock(&count_mutex);
      return (c);
 }
```

Condition Variable attributes

- **pthread_condattr_init**
  - Initialize a Condition Variable Attribute
- **pthread_condattr_destroy**
  - Remove a Condition Variable Attribute
- **pthread_condattr_setpshared**
  - Set the Scope of a Condition Variable
- **pthread_condattr_getpshared**
  - Get the Scope of a Condition Variable

Race conditions

- While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they are created. They may also execute at different speeds. When threads are executing (racing to complete) they may give unexpected results (race condition). Mutexes and joins must be utilized to acheive a predictable execution order and outcome.

BBT

Mutex Deadlock

- This condition occurs when a mutex is applied but then not "unlocked". This causes program execution to halt indefinitely. It can also be caused by poor application of mutexes or joins

BBT

Condition variable functions

- **pthread_cond_init**
  - Initialize a Condition Variable
- **pthread_cond_wait**
  - Block on a Condition Variable
- **pthread_cond_signal**
  - Unblock a Specific Thread
- **pthread_cond_timedwait**
  - Block Until a Specified Event
- **pthread_cond_broadcast**
  - Unblock All Threads
- **pthread_cond_destroy**
  - Destroy Condition Variable State

example

```
typedef struct {
    char buf[BSIZE];
    int occupied;
    int nextin;
    int nextout;
    pthread_mutex_t mutex;
    pthread_cond_t more;
    pthread_cond_t less;
} buffer_t;

buffer_t buffer;
```

Example –cont.

```
void producer(buffer_t *b, char item)
 {
    pthread_mutex_lock(&b->mutex);
    while (b->occupied >= BSIZE)
           pthread_cond_wait(&b->less, &b->mutex); assert(b->occupied <
                                                BSIZE);
    b->buf[b->nextin++] = item;
    b->nextin %= BSIZE; b->occupied++;
    /* now: either b->occupied < BSIZE and b->nextin is the index of the next
    empty slot in the buffer, or b->occupied == BSIZE and b->nextin is the
    index of the next (occupied) slot that will be emptied by a consumer (such
    as b->nextin == b->nextout) */
    pthread_cond_signal(&b->more);
    pthread_mutex_unlock(&b->mutex);
}
```

Example –cont.

```
char consumer(buffer_t *b)
{
    char item;
    pthread_mutex_lock(&b->mutex);
    while(b->occupied <= 0)
           pthread_cond_wait(&b->more, &b->mutex);
    assert(b->occupied > 0);
    item = b->buf[b->nextout++];
    b->nextout %= BSIZE;
    b->occupied--;
```

Example –cont.

```
    /* now: either b->occupied > 0 and b->nextout is the index
      of the next occupied slot in the buffer, or
      b->occupied == 0 and b->nextout is the index of the next
      (empty) slot that will be filled by a producer (such as
      b->nextout == b->nextin) */
    pthread_cond_signal(&b->less);
    pthread_mutex_unlock(&b->mutex);
    return(item);
}
```

BBT

Semaphore

- sem_init
- sem_post
- sem_wait
- sem_trywait
- sem_destroy

BBT

Read_write lock attributes

- pthread_rwlockattr_init
  - Initialize a Read-Write Lock Attribute
- pthread_rwlockattr_destroy
  - Destroy a Read-Write Lock Attribute
- pthread_rwlockattr_setpshared
  - Set a Read-Write Lock Attribute
- pthread_rwlockattr_getpshared
  - Get a Read-Write Lock Attribute

BBT

Read-Write lock functions

- **pthread_rwlock_init**
  - Initialize a Read-Write Lock
- **pthread_rwlock_rdlock**
  - Read Lock on Read-Write Lock
- **pthread_rwlock_tryrdlock**
  - Read Lock with a Nonblocking Read-Write Lock
- **pthread_rwlock_wrlock**
  - Write Lock on Read-Write Lock

BBT

Read-Write lock functions –cont.

- **pthread_rwlock_trywrlock**
  - Write Lock with a Nonblocking Read-Write Lock
- **pthread_rwlock_unlock**
  - Unlock a Read-Write Lock
- **pthread_rwlock_destroy**
  - Destroy a Read-Write Lock

BBT

thread-specific data (TSD)

- **int pthread_key_create(pthread_key_t ****key***, void (*destructor) (void *));**
  - At thread exit, if a key value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the function pointed to is called with the current associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.
- **int pthread_key_delete(pthread_key_t** ***key*****);**
- **int pthread_setspecific(pthread_key_t** ***key*****, const void ****value***);**
- **void * pthread_getspecific(pthread_key_t** ***key*****);**

BBT

Additional functions

- **pthread_t pthread_self(void);**
- **int pthread_equal(pthread_t** ***tid1*****, pthread_t** ***tid2*****);**

$\uparrow$

non-zero(equal)

- **int pthread_once(pthread_once_t ****once_control***, void (****init_routine***)(void));**
- **int sched_yield(void);**

Cancellation

- Cancel a thread
  - **int pthread_cancel(pthread_t *thread*);**
- Enable or Disable Cancellation
  - **int pthread_setcancelstate(int *state*, int **oldstate*);**
  - **PTHREAD_CANCEL_ENABLE**
  - **PTHREAD_CANCEL_DISABLE**
- Set Cancellation Type
  - **int pthread_setcanceltype(int *type*, int **oldtype*);**
  - **PTHREAD_CANCEL_DEFERRED**
  - **PTHREAD_CANCEL_ASYNCHRONOUS**
- Cancellation point
  - **void pthread_testcancel(void);**

Pthread Attributes

- **pthread_attr_t**
- **int pthread_attr_init(pthread_attr_t **tattr*);**
- **int pthread_attr_destroy(pthread_attr_t **tattr*);**

Default attributes

Attr	Value	Result
scope	PTHREAD_SCOPE_PROCESS	New thread is unbound - not permanently attached to LWP
detachstate	PTHREAD_CREATE_JOINABLE	Exit status and thread are preserved after the thread terminates
stackaddr	NULL	New thread has system-allocated stack address
stacksize	1 megabyte	New thread has system-defined stack size

Default attributes –cont.

Attr.	Value	Result
priority		New thread inherits parent thread priority
inheritsched	PTHREAD_INHERIT_SCHED	New thread inherits parent thread scheduling priority
schedpolicy	SCHED_OTHER	

Attribute functions

- **int pthread_attr_setdetachstate(pthread_attr_t **tattr*,int *detachstate*);**
  - PTHREAD_CREATE_DETACHED
  - PTHREAD_CREATE_JOINABLE
- **int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);**
- **int pthread_attr_setscope(pthread_attr_t **tattr*,int *scope*);**

Attribute functions –cont.

- **int pthread_attr_setschedpolicy(pthread_attr_t **tattr*, int *policy*);**
- **int pthread_attr_setschedparam(pthread_attr_t **tattr*, const struct sched_param *param);**
- **int pthread_attr_setstacksize(pthread_attr_t **tattr*, int *size*);**
- **int pthread_attr_setstackaddr(pthread_attr_t **tattr*,void **stackaddr*);**

Thread Pitfalls:

- Race conditions
- Thread safe code
  - The threaded routines must call functions which are "thread safe". This means that there are no static or global variables which other threads may clobber or read assuming single threaded operation. If static or global variables are used then mutexes must be applied or the functions must be re-written to avoid the use of these variables.
- Mutex Deadlock