EE3233 Systems Programming for Engrs

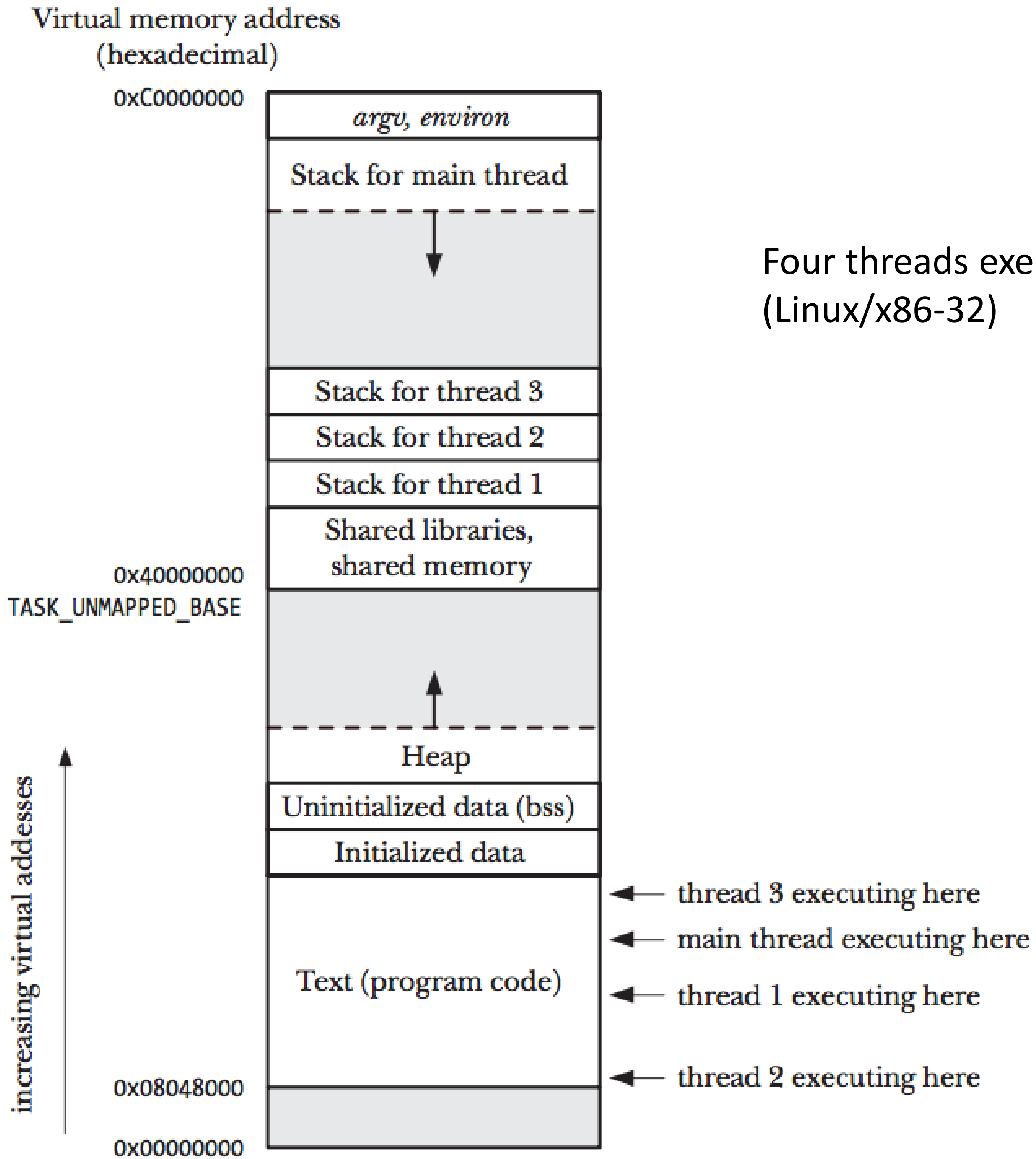
Reference: M. Kerrisk, The Linux Programming Interface

Lecture 13
Threads

ELECTRICAL & COMPUTER ENGINEERING

Thread

- A mechanism that permits an application to perform multiple tasks concurrently
- A single process can contain multiple threads
  - All of theses threads are independently executing the same program, and share the same global memory (initialized data, uninitialized data, and heap segments)
  - In a traditional UNIX, a process contains just one thread
  - Threads in a process can execute concurrently
- On a multiprocessor system, multiple threads can execute in parallel
  - If one thread is blocked on I/O, other threads are still eligible to execute



Four threads executing in a process
(Linux/x86-32)

Advantage of Threads over Processes

- In traditional UNIX achieving concurrency by creating multiple processes
  - Difficult to share information between processes: Since the parent and child don’t share memory (other than read-only text segment), we must use some form of interprocess communication
  - Process creation with *fork()* is relatively expensive: Even with *copy-on-write* technique, duplicating process attributes such as page table and file descriptor tables is time-consuming

Advantage of Threads over Processes

- Threads address both of problems:
  - Sharing information between threads is easy and fast : In order to avoid updating problem to the same information from multiple threads, synchronization techniques are employed
  - Threads creation is faster than process creation (> x10) : On Linux, threads are implemented using *clone()*. *copy-on-write* or page table duplication is not required

Thread

- Distinct for each thread
  - Thread ID
  - Signal mask
  - Real-time scheduling policy and priority
  - Capabilities (Linux-specific)
  - Stack (local variables and function call linkage info)

POSIX Thread (Pthread)

- Pthreads and types

  pthread_t : Thread identifier

  pthread_cond_t : Condition variable

  pthread_attr_t : Thread attributes object

POSIX Thread (Pthread)

- Return value from Pthread functions
  - return 0 on success and positive value on failure
  - Example program

```
pthread_t *thread;
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

Thread Creation

```
#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

- calls the function identified by *start* with argument *arg* (i.e., *start(arg)*)
- If we need to pass multiple arguments to *start*, then *arg* can be specified as a pointer to a structure containing the arguments as separate fields
- Thread points to a buffer of type *pthread_t* into which the unique identifier for this thread is copied before *pthread_create* returns

Thread Termination

- Thread's start function performs a return specifying a return value for the thread
- The thread calls *pthread_exit()*
- A thread is canceled using *pthread_cancel()*
- Any of the threads calls *exit()*, or the main thread performs a return, which causes all threads in the process to terminate immediately

pthread_exit()

- terminates calling thread

```
#include <pthread.h>

void pthread_exit (void *retval);
```

- If the main thread calls *pthread_exit()* instead of calling *exit()* or performing a return, then the other threads continue to execute

Thread IDs

- ID is returned to the caller of *pthread_create()*, and a thread can obtain its own ID using *pthread_self()*

```
#include <pthread.h>

pthread_t  pthread_self (void);
```

- useful for following reasons:
  - Various pthreads functions use thread ID to identify the thread on which they are to act
  - to tag dynamic data structures with the ID of a particular thread

Thread IDs

- to check whether two thread IDs are the same

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

- returns nonzero value if *t1* and *t2* are equal, otherwise 0
- Example: check if the ID of the calling thread matches a thread ID saved in the variable *tid*:

```
if ( pthread_equal (tid, pthread_self() )
      printf("tid matches self\n");
```

Joining with a Terminated Thread

- *pthread_join()* waits for the thread identified by thread to terminate

```
#include <pthread.h>

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

Joining

threads/simple_thread.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}
```

```
$ ./simple_thread
Message from main()
Hello world
Thread returned 12
```

```
int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

Thread Synchronization

- Two tools for thread synchronization: **mutexes**, **condition variables**
- mutexes
  - one thread does not try to access a shared variable at the same time as another thread is modifying it
- condition variables
  - inform threads that a shared variable has changed state

Mutexes

- Threads share information via global variables
  - However, multiple threads do not attempt to modify the same variable at the same time
  - One thread should not try to read the value of a variable while another thread is modifying it
- *critical section*
  - is a section of code that accesses a shared resource and whose execution should be *atomic*
  - *atomic*: its execution should not be interrupted by another thread that simultaneously accesses the same shared resource

Following program shows a problem when shared resources are not accessed atomically
threads/thread_incr.c

threads/thread_incr.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;

static void *                    /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;

    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }

    return NULL;
}
```

threads/thread_incr.c

```c
int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

Execution Results

$ ./thread_incr 1000

glob = 2000

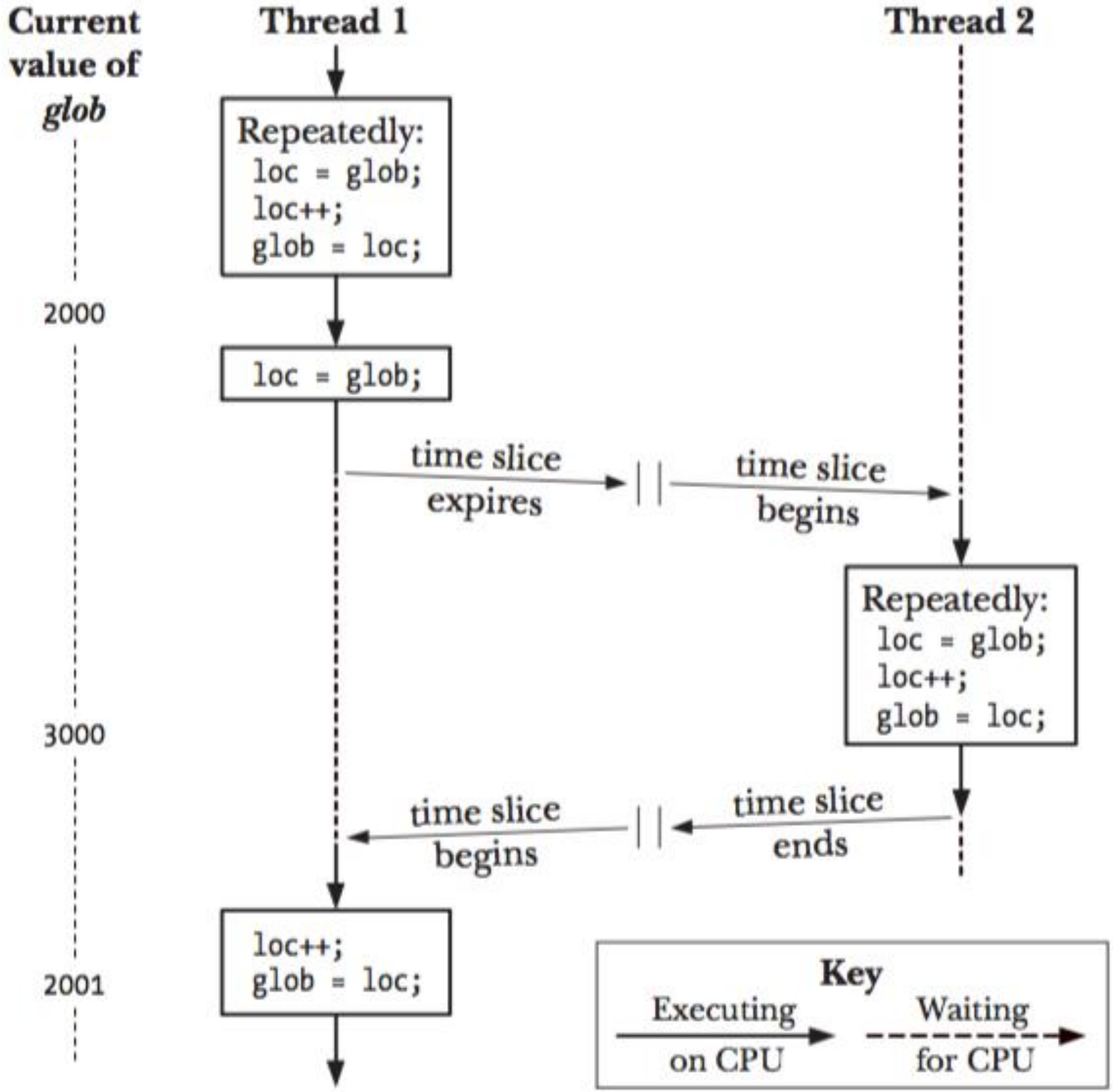
The first thread completed all of its work and terminated before the second thread even started

Execution Results

$ ./thread_incr 10000000

glob = 12039302

Why not 20000000?


Current value of glob
Thread 1
Thread 2
Repeatedly:
loc = glob;
loc++;
glob = loc;
2000
loc = glob;
time slice expires
time slice begins
Repeatedly:
loc = glob;
loc++;
glob = loc;
3000
time slice begins
time slice ends
loc++;
glob = loc;
2001
Key
Executing on CPU
Waiting for CPU

Execution Results

- Thread 1 (*t1*) fetches the current value of *glob* into *loc* (assume *glob* = 2000)
- Scheduler time slice for *t1* is expired, and *t2* commences execution
- *t2* performs multiple loops in which it fetches the current value of *glob* into *loc* (starts from *glob*=2000, when time slice terminates for *t2*, assume *glob*=3000)
- *t1* receives another time slice and resumes. It now increases *loc* and assigns to *glob* (2001)

➔ effect of increase by *t2* is lost (3000 → 2001)

Eliminating the problem

- Three statements inside the for loop → a single statement:

```
loc = glob;
loc++;
glob = loc;
```

→

```
glob++;
```

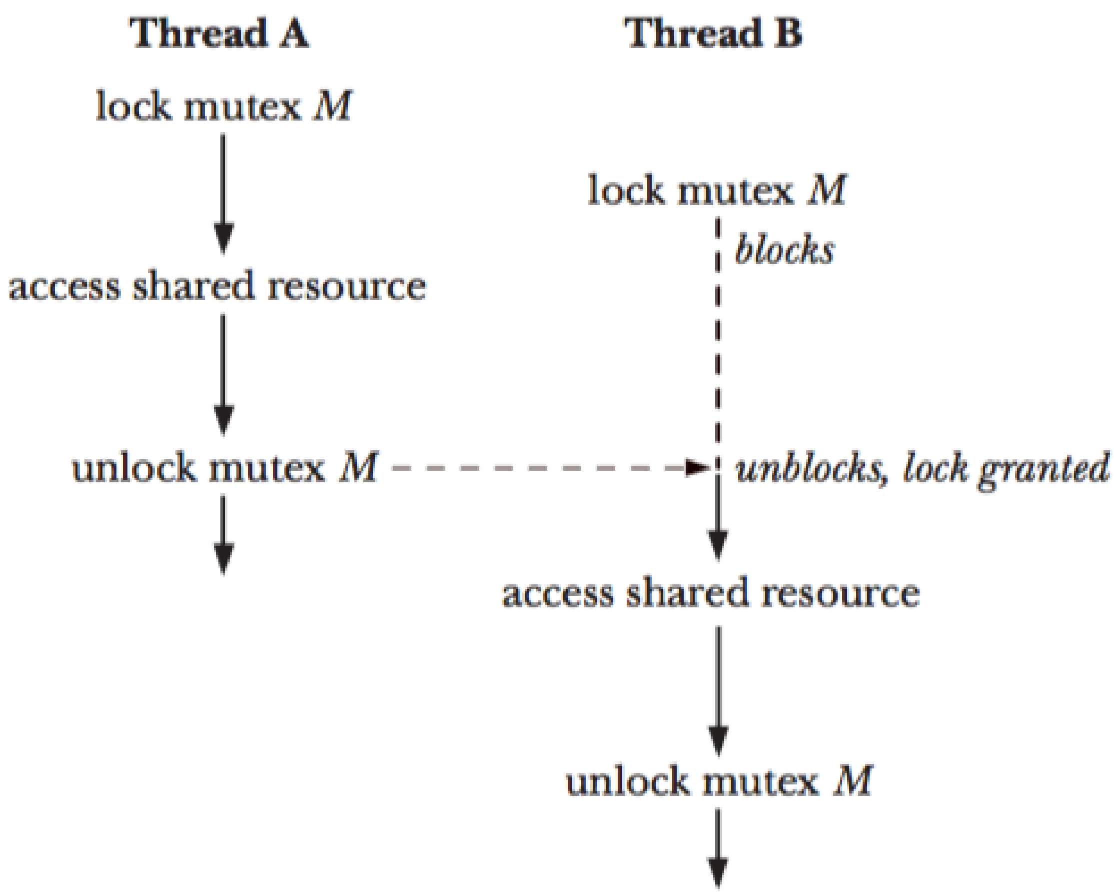
- However, on many H/W architecture, the compiler would still need to convert this single statement into machine code whose steps are equivalent to the three statements
  - To avoid the problems, use a *mutex* (mutual exclusion)

Mutex

- has two states
  - locked
  - unlocked
- At any moment, at most one thread may hold the lock on a mutex
  - Attempting to lock a mutex that is already locked either blocks or fails with an error
- When a thread locks a mutex, it becomes owner of that mutex
  - Only the owner can unlock

Mutex

- Protocol for accessing a resource
  - lock the mutex for the shared resource
  - access the shared resource
  - unlock the mutex



Statically Allocated Mutexes

- A mutex can either be
  - allocated as a static variable
  - created dynamically at run time
- variable type of mutex: pthread_mutex_t
- **Before using, a mutex must always be initialized**

```
pthread_mutex_t  mtx  =  PTHREAD_MUTEX_INITIALIZER;
```

Locking and Unlocking a Mutex

```
#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
```

- To lock a mutex, call *pthread_mutex_lock( )*
  - If the mutex is currently <u>unlocked</u>, this call locks the mutex and returns immediately
  - If the mutex is currently <u>locked</u> by another thread, then this blocks until the mutex is unlocked

Locking and Unlocking a Mutex

```
#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
```

- *pthread_mutex_unlock( )* unlocks a mutex previously locked by the calling thread
- It is error
  - to unlock a mutex that is not currently locked
  - to unlock a mutex that is locked by <u>another thread</u>

threads/thread_incr_mutex.c

```c
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *                    /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;

    for (j = 0; j < loops; j++) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        loc = glob;
        loc++;
        glob = loc;

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}
```

threads/thread_incr_mutex.c

```
int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

Execution Results

$ ./thread_incr_mutex 10000000

glob = **20000000**

Mutex Deadlocks

- Sometimes, a thread needs to simultaneously access two or more different shared resources
  - each of which is governed by a separate mutex
- When more than one thread is locking the same set of mutexes, deadlock situations can arise

Thread A	**Thread B**
1. *pthread_mutex_lock(mutex1);*	1. *pthread_mutex_lock(mutex2);*
2. *pthread_mutex_lock(mutex2);*	2. *pthread_mutex_lock(mutex1);*
blocks	blocks

- To avoid such deadlocks, define a mutex hierarchy
  - When threads can lock the same set of mutexes, they should always lock them in the same order
  - Two threads always lock the mutexes in the order of mutex1 followed by mutex2

Dynamic Initialization

```
#include < pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

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

- *mutex* : to be initialized
- *attr* : is a pointer to a *pthread_mutexattr_t* to define attributes for the mutex (If it is NULL, the mutex is assigned default attributes)
- is required in the following cases:
  - the mutex was dynamically allocated on the heap (linked list contains a *pthread_mutex_t* field)
  - the mutex is an automatic variable allocated on the stack
  - we want to initialize a statically allocated mutex with attributes other than the defaults

Condition Variables

- allows one thread to inform other threads about changes in the state of a shared variable and allows the other threads to wait (block) for such notification

Without Condition Variables

- check “threads/prod_no_condvar.c”
- Producer(main) wastes CPU time due to continuous loop checking the state of the variable *avail*
- condition variable remedies this problem
  - it allows a thread to sleep until another thread notifies (signals) it

Static Allocation of Condition Variables

```
pthread_cond_t  cond  =  PTHREAD_COND_INITIALIZER;
```

- As with a mutex, a condition variable must be initialized before use

Signaling and Waiting on Condition Variables

```
#include <pthread.h>

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

- condition variable always has an associated mutex
  - Both of these objects are passed as arguments to *pthread_cond_wait( )*
- *pthread_cond_wait( )* performs the following steps:
  - unlock the mutex specified by *mutex* (so that other threads can access the shared variable)
  - block the calling thread until another thread signals the condition variable *cond*; and
  - relock *mutex* (since the thread then immediately accesses the shared variable)

Signaling and Waiting on Condition Variables

- check “threads/prod_condvar.c”

Signaling and Waiting on Condition Variables

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static phtread_cond_t cond = PTHREAD_COND_INITIALIZER;

static void *
threadFunc(void *arg) {

int cnt = atoi((char *) arg);
int s, j;

   for (j=0; j < cnt; j++) {
         sleep (1);
         s = pthread_mutex_lock(&mtx);
         if (s != 0)
                  errExitEN(s, “pthread_mutex_lock”);
         avail++;
         s = pthread_mutex_unlock(&mtx);
         if (s != 0)
                  errExitEN(s, “pthread_mutex_unlock”);
         s = pthread_cond_signal(&cond);
         if (s != 0)
                  errExitEN(s, “pthread_cond_signal”);
   }

return NULL;
}
```

Signaling and Waiting on Condition Variables

/* main thread : consumer */

```
for (;;) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail == 0) {            /* Wait for something to consume */
        s = pthread_cond_wait(&cond, &mtx);
        if (s != 0)
            errExitEN(s, "pthread_cond_wait");
    }

    while (avail > 0) {             /* Consume all available units */
        /* Do something with produced unit */
        avail--;
    }

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

    /* Perhaps do other work here that doesn't require mutex lock */
}
```