System Programming

Week 11: Threads

Seongjin Lee

Updated: 2018/07/06
08_thread

insight@gnu.ac.kr
http://open.gnu.ac.kr
Systems Research Lab.
Gyeongsang National University

Table of contents

1. Thread Concepts

2. Thread Creation and Termination

3. Thread Synchronization

3.1 Mutexes

3.2 Deadlock Avoidance

3.3 Reader-Writer Locks

3.4 Condition Variables

3.5 Spin Locks

3.6 Barriers

Introduction

Limited amount of sharing can occur between related processes

thread of control (or simply *threads* to perform multiple tasks within the environment of a signle process.

All threads within a single process have access to the same process components, such as file descriptor and memory

- Threads Concepts
- Creation, Termination
- Consistency and synchronization mechanism
- Mutex, Reader-Writer Lock, Condition variable, Lock

Thread Concepts

Thread Concepts

A typical UNIX process can be thought of as having a single thread of control:

- each process is doing only one thing at a time.

With multiple threads of control,

- we can design our programs to do more than one thing at a time within a single process,
- each thread handles a separate task.

Thread Concepts cnt'd

There are several benefits

- We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type.
- Threads automatically have access to the same memory address space and file descriptors.
- With multiple threads of control, the processing of independent tasks can be interleaved by assigning a separate thread per task.
- interactive programs can realize improved response time by using multiple threads to separate the portions of the program

Thread Concepts cnt'd

Some people associate multithreaded programming with multiprocessor or multicore systems.

- The benefits of a multithreaded programming model can be realized even if your program is running on a uniprocessor.
- Some threads might be able to run while others are blocked

Thread Concepts cnt'd

A thread consists of the information necessary to represent an execution context within a process.

- a thread ID that identifies the thread within a process,
- a set of register values,
- a stack,
- a scheduling priority and policy,
- a signal mask,
- an errno variable (recall Section 1.7),
- and thread-specific data (Section 12.6).

Everything within a process is sharable among the threads in a process, including the text of the executable program, the program's global and heap memory, the stacks, and the file descriptors

Thread Identification

every thread has a thread ID

- thread ID has significance only within the context of the process to which it belongs.

A thread ID is represented by the `pthread_t` data type. Some times it is useful to print thread ID during program debugging

A thread can obtain its own thread ID by calling the `pthread_self`

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
// Returns: nonzero if equal, 0 otherwise

pthread_t pthread_self(void);
Returns: the thread ID of the calling thre
```

Thread Identification cnt'd

The master thread controls job assignment by placing the ID of the thread that should process the job in each job structure. Each worker thread then removes only jobs that are tagged with its own thread ID.

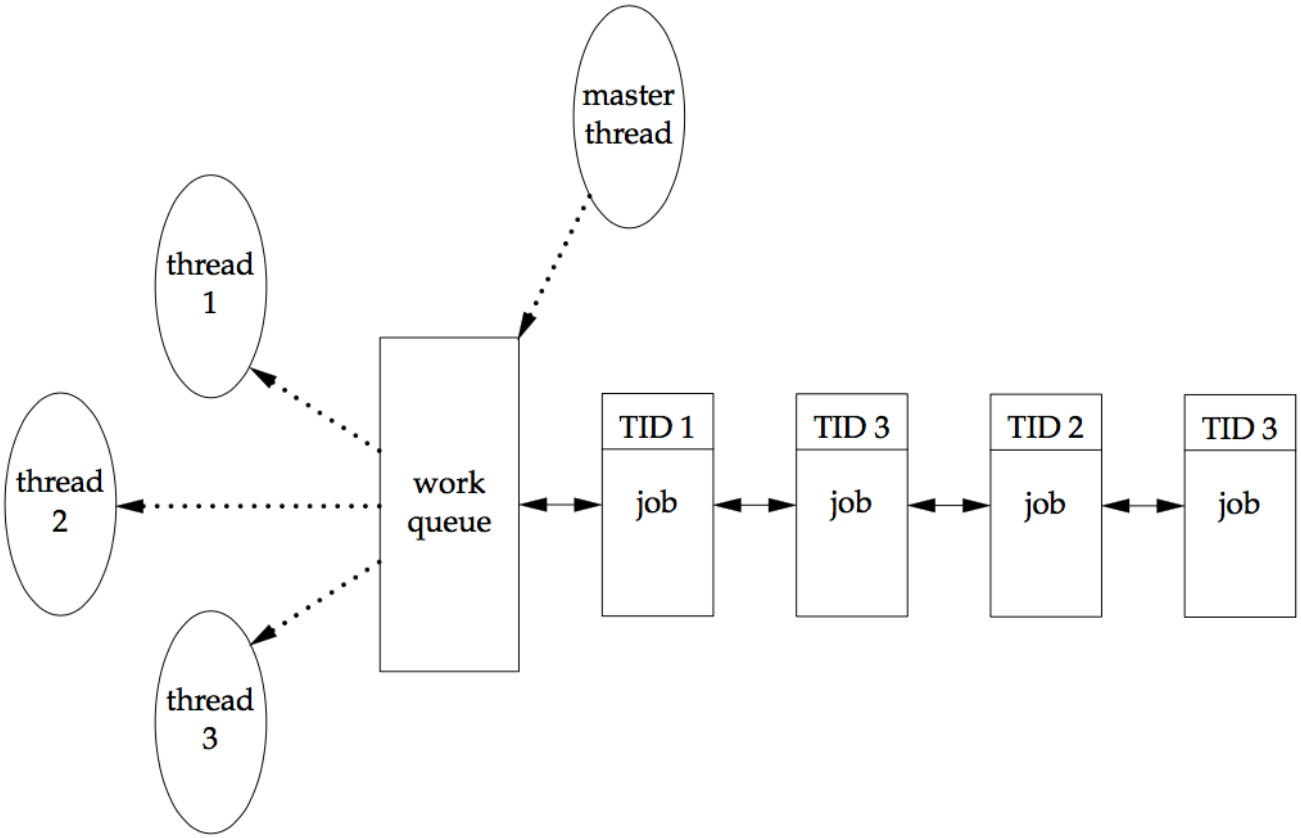

Figure: Work queue example

Thread Creation and Termination

Thread Creation

Additional threads can be created by calling the `pthread_create`

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void *),
                   void *restrict arg);
// Returns: 0 if OK, error number on failure
```

- The memory location pointed to by *tidp* is set to the thread ID of the newly created thread
- *attr* argument is used to customize various thread attributes.
- The newly created thread starts running at the address of the *start_rtn* function.
- This function takes a single argument, *arg*, which is a typeless pointer.
- When a thread is created, there is no guarantee which will run first: the newly created thread or the calling thread.
- set of pending signals for the thread is cleared.
- Note that the pthread functions usually return an error code when they fail. They don't set errno like the other POSIX functions.

Thread Creation: codes/print_thrID.c I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  pthread_t ntid;
7
8  void
9  printids(const char *s)
10 {
11     pid_t pid;
12     pthread_t tid;
13     pid = getpid();
14     tid = pthread_self();
15     printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
16       (unsigned long)tid, (unsigned long)tid);
17 }
18
19 void *
20 thr_fn(void *arg)
21 {
22     printids("new thread: ");
```

Thread Creation: codes/print_thrID.c II

```
23      return((void *)0);
24  }
25
26  int
27  main(void)
28  {
29  int err;
30      err = pthread_create(&ntid, NULL, thr_fn, NULL);
31      if (err != 0){
32          fprintf(stderr, "can't create thread");
33      }
34      printids("main thread:");
35      sleep(1);
36      exit(0);
37  }
```

```
> ./print_thrID
main thread: pid 22813 tid 140736689316800 (0x7fffd05fa3c0)
new thread:  pid 22813 tid 123145322315776 (0x700001314000)
```

Thread Creation: codes/print_thrID.c

There are two oddities

- The first is the need to sleep in the main thread.
  - If it doesn't sleep, the main thread might exit, thereby terminating the entire process before the new thread gets a chance to run.
  - This behavior is dependent on the operating system's threads implementation and scheduling algorithms.
- The second oddity is that the new thread obtains its thread ID by calling `pthread_self`

The main thread stores this ID in `ntid`, but the new thread can't safely use it. If the new thread runs before the main thread returns from calling `pthread_create`, then the new thread will see the uninitialized contents of `ntid` instead of the thread ID

Thread Termination

If any thread within a process calls `exit`, `_Exit`, or `_exit`, then the entire process terminates.

Thread Termination

If any thread within a process calls `exit`, `_Exit`, or `_exit`, then the entire process terminates.

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.

Thread Termination

If any thread within a process calls `exit`, `_Exit`, or `_exit`, then the entire process terminates.

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.

Thread Termination

If any thread within a process calls `exit`, `_Exit`, or `_exit`, then the entire process terminates.

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.
3. The thread can call `pthread_exit`.

Thread Termination cnt'd

pthread_exit

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

- The `rval_ptr` argument is a typeless pointer, similar to the single argument passed to the start routine.
- This pointer is available to other threads in the process by calling the `pthread_join` function
- The calling thread will block until the specified thread calls `pthread_exit`, returns from its start routine, or is canceled.
- If the thread simply returned from its start routine, `rval_ptr` will contain the return code.
- If the thread was canceled, the memory location specified by `rval_ptr` is set to `PTHREAD_CANCELED`.

Thread Termination cnt'd

pthread_join

```
int pthread_join(pthread_t thread, void **rval_ptr);
// Returns: 0 if OK, error number on failure
```

- By calling `pthread_join`, we automatically place the thread with which we're joining in the detached state so that its resources can be recovered.
- If we are note interested in a thread's return value, we can set `rval_ptr` to NULL

Thread Termination: codes/exit-state.c I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6
7  void *
8  thr_fn1(void *arg)
9  {
10     printf("thread 1 returning\n");
11     return((void *)1);
12 }
13
14 void *
15 thr_fn2(void *arg)
16 {
17     printf("thread 2 exiting\n");
18     pthread_exit((void *)2);
19 }
20 int
21 main(void)
22 {
```

Thread Termination: codes/exit-state.c II

```
23      int err;
24      pthread_t tid1, tid2;
25      void *tret;
26      err = pthread_create(&tid1, NULL, thr_fn1, NULL);
27      if (err != 0){
28          fprintf(stderr, "can't create thread 1");
29      }
30
31      err = pthread_create(&tid2, NULL, thr_fn2, NULL);
32      if (err != 0){
33          fprintf(stderr, "can't create thread 2");
34      }
35
36      err = pthread_join(tid1, &tret);
37      if (err != 0){
38          fprintf(stderr, "can't join with thread 1");
39      }
40      printf("thread 1 exit code %ld\n", (long)tret);
41
42      err = pthread_join(tid2, &tret);
43      if (err != 0){
44          fprintf(stderr, "can't join with thread 2");
45      }
```

Thread Termination: codes/exit-state.c III

```
46
47      printf("thread 2 exit code %ld\n", (long)tret);
48      exit(0);
49  }
```

```
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

Thread Termination cnt'd

- The typeless pointer passed to `pthread_create` and `pthread_exit` can be used to pass more than a single value.
- Be careful that the memory used for the structure is still valid when the caller has completed.
- If a thread allocates a structure on its stack and passes a pointer to this structure to `pthread_exit`, then the stack might be destroyed and its memory reused for something else by the time the caller of `pthread_join` tries to use it.

Thread Termination cnt'd: exit-wrong.c I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  struct foo {
7     int a, b, c, d;
8  };
9
10 void
11 printfoo(const char *s, const struct foo *fp)
12 {
13     printf("%s", s);
14     printf(" structure at 0x%lx\n", (unsigned long)fp);
15     printf(" foo.a = %d\n", fp->a);
16     printf(" foo.b = %d\n", fp->b);
17     printf(" foo.c = %d\n", fp->c);
18     printf(" foo.d = %d\n", fp->d);
19 }
20
21 void *
22 thr_fn1(void *arg)
```

Thread Termination cnt'd: exit-wrong.c II

```
23  {
24      struct foo foo = {1, 2, 3, 4};
25      printfoo("thread 1:\n", &foo);
26      pthread_exit((void *)&foo);
27  }
28
29  void *
30  thr_fn2(void *arg)
31  {
32      printf("thread 2: ID is %lu\n", (unsigned long)pthread_self());
33      pthread_exit((void *)0);
34  }
35
36  int
37  main(void)
38  {
39      int err;
40      pthread_t tid1, tid2;
41      struct foo *fp;
42      err = pthread_create(&tid1, NULL, thr_fn1, NULL);
43      if (err != 0){
44          fprintf(stderr, "can't create thread 1");
45      }
```

Thread Termination cnt'd: exit-wrong.c III

```
46
47      err = pthread_join(tid1, (void *)&fp);
48      if (err != 0){
49          fprintf(stderr, "can't join with thread 1");
50      }
51
52      sleep(1);
53      printf("parent starting second thread\n");
54      err = pthread_create(&tid2, NULL, thr_fn2, NULL);
55      if (err != 0){
56          fprintf(stderr, "can't create thread 2");
57      }
58
59      sleep(1);
60      printfoo("parent:\n", fp);
61      exit(0);
62  }
```

Thread Termination cnt'd: exit-wrong.c IV

```
./exit-wrong
thread 1:
  structure at 0x7000052d8ed0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 123145389182976
parent:
  structure at 0x7000052d8ed0
Segmentation fault: 11
```

Even though the memory is still intact after the thread exits, we can't depend on this always being the case. It certainly isn't what we observe on the other platforms.

Thread Termination cnt'd

One thread can request that another in the same process be canceled by calling the `pthread_cancel` function.

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
// Returns: 0 if OK, error number on failure
```

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`.

However, a thread can elect to ignore or otherwise control how it is canceled.

- it merely makes the request.

Thread Termination cnt'd

A thread can arrange for functions to be called when it exits, similar to the way as the `atexit` function

The functions are known as *thread cleanup handlers*.

More than one cleanup handler can be established for a thread.

- The handlers are recorded in a stack, which means that they are executed in the reverse order from that with which they were registered.

Thread Termination cnt'd

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void *), void *arg);
void pthread_cleanup_pop(int execute};
```

The `pthread_cleanup_push` function schedules the cleanup function, *rtn*, to be called with the single argument, *arg*, when the thread performs one of the following actions:

1. Makes a call to `pthread_exit`
2. Responds to a cancellation request
3. Makes a call to `pthread_cleanup_pop` with a nonzero execute argument

Thread Termination cnt'd

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void *), void *arg);
void pthread_cleanup_pop(int execute};
```

If the *execute* argument is set to zero, the cleanup function is not called.

- In either case, `pthread_cleanup_pop` removes the cleanup handler established by the last call to `pthread_cleanup_push`.

They can be used as macros

Thread Termination cnt'd: codes/push-pop.c 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  void
7  cleanup(void *arg)
8  {
9      printf("cleanup: %s\n", (char *)arg);
10 }
11
12 void *
13 thr_fn1(void *arg)
14 {
15     printf("thread 1 start\n");
16     pthread_cleanup_push(cleanup, "thread 1 first handler");
17     pthread_cleanup_push(cleanup, "thread 1 second handler");
18     printf("thread 1 push complete\n");
19     if (arg)
20         return((void *)1);
21     pthread_cleanup_pop(0);
22     pthread_cleanup_pop(0);
```

Thread Termination cnt'd: codes/push-pop.c II

```
23      return((void *)1);
24  }
25
26  void *
27  thr_fn2(void *arg)
28  {
29      printf("thread 2 start\n");
30      pthread_cleanup_push(cleanup, "thread 2 first handler");
31      pthread_cleanup_push(cleanup, "thread 2 second handler");
32      printf("thread 2 push complete\n");
33      if (arg)
34          pthread_exit((void *)2);
35      pthread_cleanup_pop(0);
36      pthread_cleanup_pop(0);
37      pthread_exit((void *)2);
38  }
39
40
41  int
42  main(void)
43  {
44      int err;
45      pthread_t tid1, tid2;
```

Thread Termination cnt'd: codes/push-pop.c III

```
46      void *tret;
47
48      err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
49      if (err != 0){
50          fprintf(stderr, "can't create thread 1");
51      }
52
53      err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
54      if (err != 0){
55          fprintf(stderr, "can't create thread 2");
56      }
57
58      err = pthread_join(tid1, &tret);
59      if (err != 0){
60          fprintf(stderr, "can't join with thread 1");
61      }
62
63      printf("thread 1 exit code %ld\n", (long)tret);
64
65      err = pthread_join(tid2, &tret);
66      if (err != 0){
67          fprintf(stderr, "can't join with thread 2");
68      }
```

Thread Termination cnt'd: codes/push-pop.c IV

```
69
70      printf("thread 2 exit code %ld\n", (long)tret);
71
72      exit(0);
73  }
```

```
./push-pop
thread 1 start
thread 2 start
thread 1 push complete
thread 2 push complete
cleanup: thread 1 second handler
cleanup: thread 2 second handler
cleanup: thread 2 first handler
Segmentation fault: 11
```

Thread Termination cnt'd

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Figure: Comparison of process and thread primitives

By default thread's termination status is retained until we call `pthread_join` for that thread.

Thread Termination cnt'd

A thread's underlying storage can be reclaimed immediately on termination if the thread has been detached.

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
//Returns: 0 if OK, error number on failure
```

we can create a thread that is already in the detached state by modifying the thread attributes we pass to `pthread_create`.

Thread Synchronization

Thread Synchronization

we need to make sure that each thread sees a consistent view of its data.

when one thread can modify a variable that other threads can read or modify

- we need to synchronize the threads to ensure that they don't use an invalid value when accessing the variable's memory contents.

When one thread modifies a variable, other threads can potentially see inconsistencies when reading the value of that variable.

Thread Synchronization cnt'd

thread A reads the variable and then writes a new value to it, but the write operation takes two memory cycles.

If thread B reads the same variable between the two write cycles, it will see an inconsistent value.

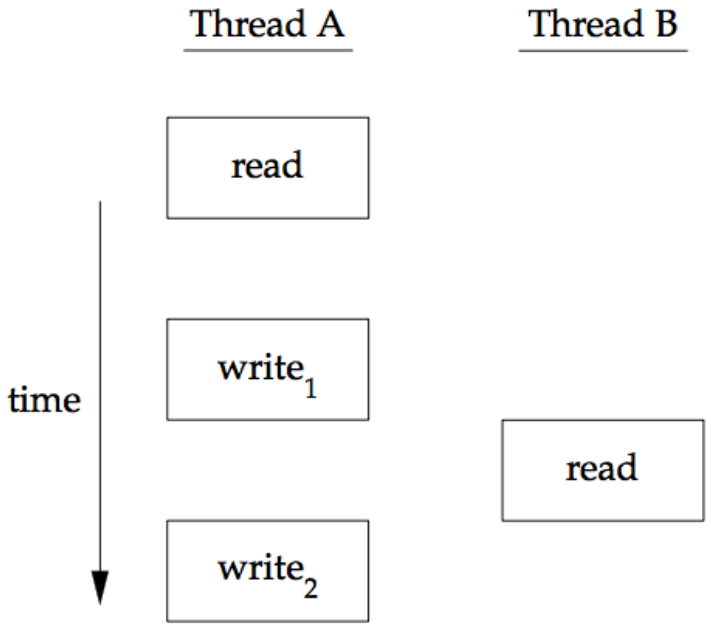

Figure: Interleaved memory cycles with two threads

Thread Synchronization cnt'd

To solve this problem, the threads have to use a lock that will allow only one thread to access the variable at a time.

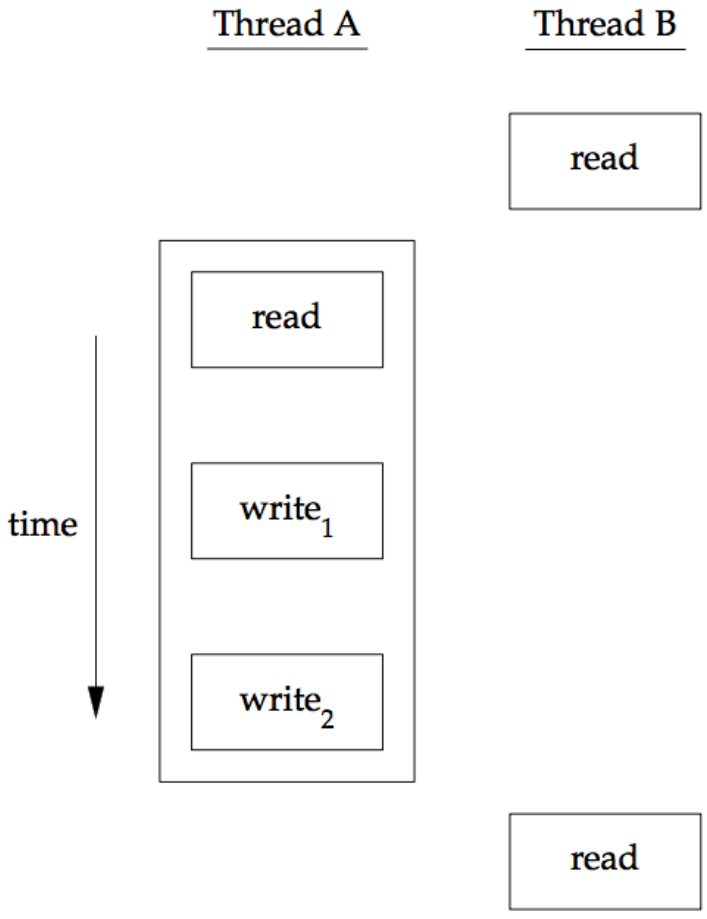

Figure: Two threads synchronizing memory access

Thread Synchronization cnt'd

We also need to synchronize two or more threads that might try to modify the same variable at the same time.

The increment operation is usually broken down into three steps.

1. Read the memory location into a register.
2. Increment the value in the register.
3. Write the new value back to the memory location

If two threads try to increment the same variable at almost the same time without synchronizing with each other, the results can be inconsistent.

If the modification is atomic, then there isn't a race.

Thread Synchronization cnt'd

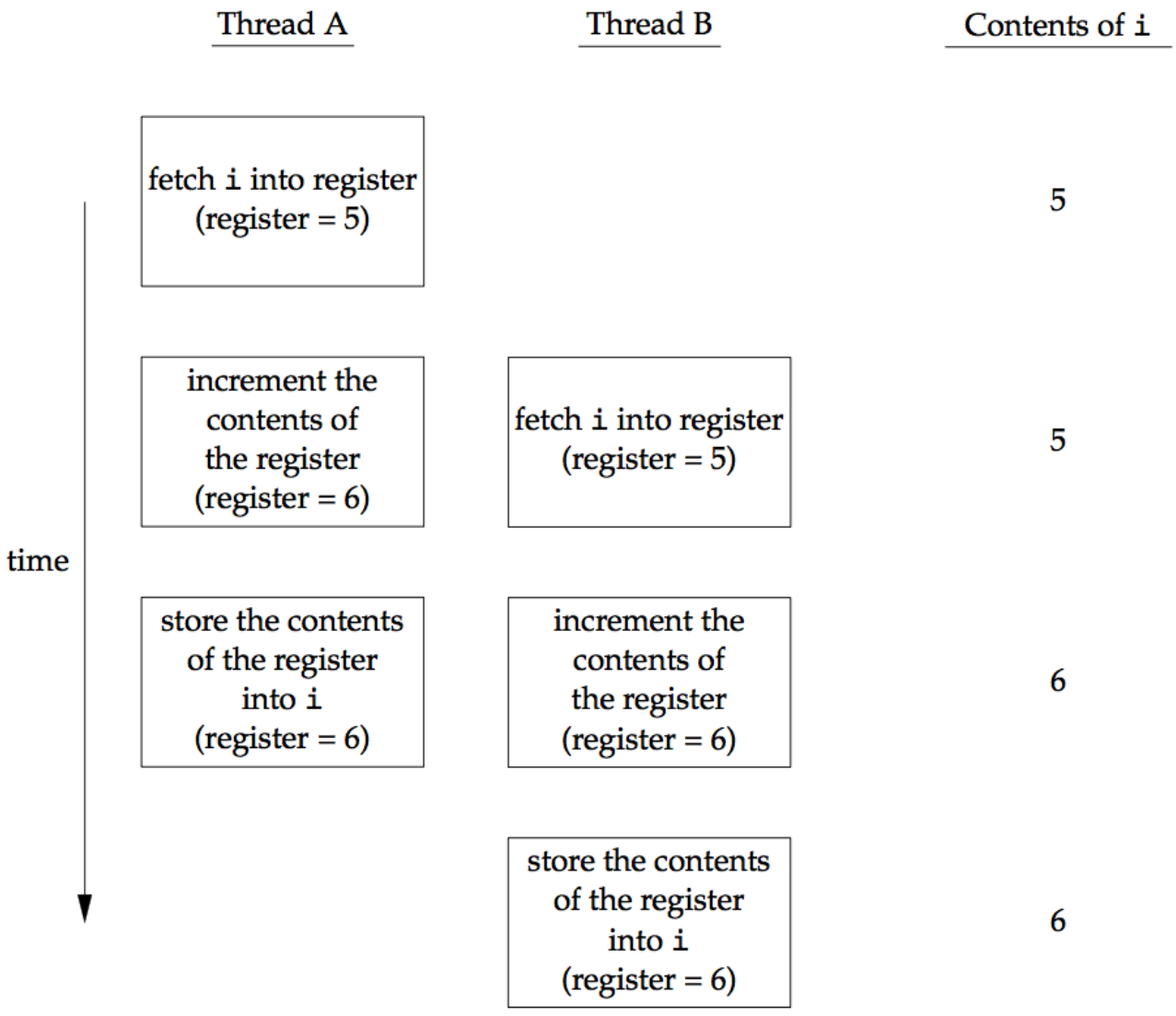

Figure: Two unsynchronized threads incrementing the same variable

Thread Synchronization : Mutexes

Mutexes

We can protect our data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces.

A mutex is basically a lock

- we set (lock) before accessing a shared resource and release (unlock) when we're done.
- While it is set, any other thread that tries to set it will block until we release it.
- If more than one thread is blocked when we unlock the mutex,
  - then all threads blocked on the lock will be made runnable,
  - and the first one to run will be able to set the lock.
- The others will see that the mutex is still locked and go back to waiting for it to become available again.

Mutexes cnt'd

Mutual-exclusion mechanism works only if we design our threads to follow the same data-access rules.

- The operating system doesn't serialize access to data for us.

If we allow one thread to access a shared resource without first acquiring a lock

- then inconsistencies can occur even though the rest of our threads do acquire the lock before attempting to access the shared resource

Mutexes cnt'd

A mutex variable is represented by the `pthread_mutex_t` data type.

We must first initialize it by either

- setting it to the constant `PTHREAD_MUTEX_INITIALIZER` (for statically allocated mutexes only)
- or calling `pthread_mutex_init`.

If we allocate the mutex dynamically (by calling malloc, for example), then we need to call `pthread_mutex_destroy` before freeing the memory.

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
//Both return: 0 if OK, error number on failure
```

To initialize a mutex with the default attributes, we set *attr* to `NULL`

Mutexes cnt'd

To lock a mutex, we call `pthread_mutex_lock`.

If the mutex is already locked,

- the calling thread will block until the mutex is unlocked.

To unlock a mutex, we call `pthread_mutex_unlock`.

```
#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
```

If a thread can't afford to block, it can use `pthread_mutex_trylock` to lock the mutex conditionally

Mutexes cnt'd: codes/mutex1.c I

Following example illustrates a mutex used to protect a data structure.

When more than one thread needs to access a dynamically allocated object, we can embed a reference count in the object to ensure that we don't free its memory before all threads are done using it

```
1  #include <stdlib.h>
2  #include <pthread.h>
3
4  struct foo {
5     int f_count;
6     pthread_mutex_t f_lock;
7     int f_id;
8     /* ... more stuff here ... */
9  };
10
11 struct foo *
12 foo_alloc(int id) /* allocate the object */
13 {
14    struct foo *fp;
15
```

Mutexes cnt'd: codes/mutex1.c II

```
16      if ((fp = malloc(sizeof(struct foo))) != NULL) {
17         fp->f_count = 1;
18         fp->f_id = id;
19         if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
20            free(fp);
21            return(NULL);
22         }
23         /* ... continue initialization ... */
24      }
25      return(fp);
26   }
27
28   void
29   foo_hold(struct foo *fp) /* add a reference to the object */
30   {
31      pthread_mutex_lock(&fp->f_lock);
32      fp->f_count++;
33      pthread_mutex_unlock(&fp->f_lock);
34   }
35
36   void
37   foo_rele(struct foo *fp) /* release a reference to the object */
38   {
```

Mutexes cnt'd: codes/mutex1.c III

```
39      pthread_mutex_lock(&fp->f_lock);
40      if (--fp->f_count == 0) { /* last reference */
41          pthread_mutex_unlock(&fp->f_lock);
42          pthread_mutex_destroy(&fp->f_lock);
43          free(fp);
44      } else {
45          pthread_mutex_unlock(&fp->f_lock);
46      }
47  }
```

No locking is necessary necessary when we initialize the reference count to 1 in the `foo_alloc` function, because the allocating thread is the only reference to it so far.

In this example, we have ignored how threads find an object before calling `foo_hold`.

Even though the reference count is zero, it would be a mistake for `foo_rele` to free the object's memory if another thread is blocked on the mutex in a call to `foo_hold`.

Thread Synchronization : Dead-lock Avoidance

Deadlock Avoidance

when we use more than one mutex in our programs, a deadlock can occur

- if we allow one thread to hold a mutex and block while trying to lock a second mutex at the same time that another thread holding the second mutex tries to lock the first mutex.

Deadlocks can be avoided by carefully controlling the order in which mutexes are locked.

- If all threads always lock mutex A before mutex B, no deadlock can occur from the use of the two mutexes

Sometimes, an application's architecture makes it difficult to apply a lock ordering

Deadlock Avoidance cnt'd: `codes/mutex2.c` I

In this example, we use two mutexes to avoid dealocks by always lock them in the same order

The second mutex protects a hash list that we use to keep track of the foo data structure

`hashlock` mutex protects both the `fh` hash table and the `f_next` hash link field in the `foo` structure. The `f_lockmutex` in the `foo` structure protects access to the remainder of the foo structure's fields.

Deadlock Avoidance cnt'd: codes/mutex2.c II

```
1  #include <stdlib.h>
2  #include <pthread.h>
3
4  #define NHASH 29
5  #define HASH(id) (((unsigned long)id)%NHASH)
6
7  struct foo *fh[NHASH];
8
9  pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;
10
11 struct foo {
12    int f_count;
13    pthread_mutex_t f_lock;
14    int f_id;
15    struct foo *f_next; /* protected by hashlock */
16    /* ... more stuff here ... */
17 };
18
19
20
21 /* The allocation function locks the hash list lock, adds the new structure to
22  * a hash bucket, and before unlocking the hash list lock, locks the mutex in
23  * the new structure. Since the new structure is placed on a global list,
```

Deadlock Avoidance cnt'd: codes/mutex2.c III

```
24   * other threads can find it, so we need to block them if they try to access
25   * the new structure, until we are done initializing it.
26   */
27  struct foo *
28  foo_alloc(int id) /* allocate the object */
29  {
30     struct foo *fp;
31     int   idx;
32
33     if ((fp = malloc(sizeof(struct foo))) != NULL) {
34        fp->f_count = 1;
35        fp->f_id = id;
36        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
37           free(fp);
38           return(NULL);
39        }
40        idx = HASH(id);
41        pthread_mutex_lock(&hashlock);
42        fp->f_next = fh[idx];
43        fh[idx] = fp;
44        pthread_mutex_lock(&fp->f_lock);
45        pthread_mutex_unlock(&hashlock);
46        /* ... continue initialization ... */
```

Deadlock Avoidance cnt'd: codes/mutex2.c IV

```
47          pthread_mutex_unlock(&fp->f_lock);
48      }
49      return(fp);
50  }
51
52  void
53  foo_hold(struct foo *fp) /* add a reference to the object */
54  {
55      pthread_mutex_lock(&fp->f_lock);
56      fp->f_count++;
57      pthread_mutex_unlock(&fp->f_lock);
58  }
59
60
61  /* The foo_find function locks the hash list lock and searches for the
62   * requested structure. If it is found, we increase the reference count and
63   * return a pointer to the structure.
64   */
65  struct foo *
66  foo_find(int id) /* find an existing object */
67  {
68      struct foo *fp;
69
```

Deadlock Avoidance cnt'd: codes/mutex2.c V

```
70      pthread_mutex_lock(&hashlock);
71      for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
72         if (fp->f_id == id) {
73            foo_hold(fp);
74            break;
75         }
76      }
77      pthread_mutex_unlock(&hashlock);
78      return(fp);
79   }
80
81
82   /* Now with two locks, the foo_rele function is more complicated. If this is
83    * the last reference, we need to unlock the structure mutex so that we can
84    * acquire the hash list lock, since we'll need to remove the structure from
85    * the hash list. Then we reacquire the structure mutex. Because we could have
86    * blocked since the last time we held the structure mutex, we need to recheck
87    * the condition to see whether we still need to free the structure.
88    */
89   void
90   foo_rele(struct foo *fp) /* release a reference to the object */
91   {
92      struct foo *tfp;
```

Deadlock Avoidance cnt'd: codes/mutex2.c VI

```
93      int   idx;
94
95      pthread_mutex_lock(&fp->f_lock);
96      if (fp->f_count == 1) { /* last reference */
97         pthread_mutex_unlock(&fp->f_lock);
98         pthread_mutex_lock(&hashlock);
99         pthread_mutex_lock(&fp->f_lock);
100        /* need to recheck the condition */
101        if (fp->f_count != 1) {
102           fp->f_count--;
103           pthread_mutex_unlock(&fp->f_lock);
104           pthread_mutex_unlock(&hashlock);
105           return;
106        }
107        /* remove from list */
108        idx = HASH(fp->f_id);
109        tfp = fh[idx];
110        if (tfp == fp) {
111           fh[idx] = fp->f_next;
112        } else {
113           while (tfp->f_next != fp)
114              tfp = tfp->f_next;
115           tfp->f_next = fp->f_next;
```

Deadlock Avoidance cnt'd: codes/mutex2.c VII

```
116         }
117         pthread_mutex_unlock(&hashlock);
118         pthread_mutex_unlock(&fp->f_lock);
119         pthread_mutex_destroy(&fp->f_lock);
120         free(fp);
121     } else {
122         fp->f_count--;
123         pthread_mutex_unlock(&fp->f_lock);
124     }
125 }
```

Deadlock Avoidance cnt'd: codes/mutex3.c I

This locking approach is complex, so we need to revisit our design.

We can simplify things considerably by using the hash list lock to protect the structure reference count, too.

The structure mutex can be used to protect everything else in the foo structure.

```
1  #include <stdlib.h>
2  #include <pthread.h>
3
4  #define NHASH 29
5  #define HASH(id) (((unsigned long)id)%NHASH)
6
7  struct foo *fh[NHASH];
8  pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;
9
10 struct foo {
11    int f_count; /* protected by hashlock */
12    pthread_mutex_t f_lock;
13    int f_id;
```

Deadlock Avoidance cnt'd: codes/mutex3.c II

```
14      struct foo *f_next; /* protected by hashlock */
15      /* ... more stuff here ... */
16  };
17
18  /* The lock-ordering issues surrounding the hash list and the reference count
19   * go away when we use the same lock for both purposes.
20   */
21  struct foo *
22  foo_alloc(int id) /* allocate the object */
23  {
24      struct foo *fp;
25      int   idx;
26
27      if ((fp = malloc(sizeof(struct foo))) != NULL) {
28          fp->f_count = 1;
29          fp->f_id = id;
30          if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
31              free(fp);
32              return(NULL);
33          }
34          idx = HASH(id);
35          pthread_mutex_lock(&hashlock);
36          fp->f_next = fh[idx];
```

Deadlock Avoidance cnt'd: codes/mutex3.c III

```
37          fh[idx] = fp;
38          pthread_mutex_lock(&fp->f_lock);
39          pthread_mutex_unlock(&hashlock);
40          /* ... continue initialization ... */
41          pthread_mutex_unlock(&fp->f_lock);
42      }
43      return(fp);
44  }
45
46  void
47  foo_hold(struct foo *fp) /* add a reference to the object */
48  {
49      pthread_mutex_lock(&hashlock);
50      fp->f_count++;
51      pthread_mutex_unlock(&hashlock);
52  }
53
54  struct foo *
55  foo_find(int id) /* find an existing object */
56  {
57      struct foo *fp;
58
59      pthread_mutex_lock(&hashlock);
```

Deadlock Avoidance cnt'd: codes/mutex3.c IV

```
60      for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
61          if (fp->f_id == id) {
62              fp->f_count++;
63              break;
64          }
65      }
66      pthread_mutex_unlock(&hashlock);
67      return(fp);
68  }
69
70  void
71  foo_rele(struct foo *fp) /* release a reference to the object */
72  {
73      struct foo *tfp;
74      int   idx;
75
76      pthread_mutex_lock(&hashlock);
77      if (--fp->f_count == 0) { /* last reference, remove from list */
78          idx = HASH(fp->f_id);
79          tfp = fh[idx];
80          if (tfp == fp) {
81              fh[idx] = fp->f_next;
82          } else {
```

Deadlock Avoidance cnt'd: codes/mutex3.c V

```
83          while (tfp->f_next != fp)
84             tfp = tfp->f_next;
85          tfp->f_next = fp->f_next;
86       }
87       pthread_mutex_unlock(&hashlock);
88       pthread_mutex_destroy(&fp->f_lock);
89       free(fp);
90    } else {
91       pthread_mutex_unlock(&hashlock);
92    }
93 }
```

Deadlock Avoidance cnt'd

If your locking granularity is too coarse,

- you end up with too many threads blocking behind the same locks, with little improvement possible from concurrency.

If your locking granularity is too fine,

- then you suffer bad performance from excess locking overhead, and you end up with complex code.

Deadlock Avoidance cnt'd

One additional mutex primitive allows us to bound the time that a thread blocks when a mutex it is trying to acquire is already locked.

The `pthread_mutex_timedlock` function is equivalent to `pthread_mutex_lock`,

- but if the timeout value is reached, `pthread_mutex_timedlock`

```
#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
```

pthread_mutex_timedlock Fuction: codes/timedlock.c I

how to use pthread_mutex_timedlock to avoid blocking indefinitely.

note that Mac OS does not have pthread_mutex_timedlock

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5  #include <pthread.h>
6
7  int
8  main(void)
9  {
10    int err;
11    struct timespec tout;
12    struct tm *tmp;
13    char buf[64];
14    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
15
16    pthread_mutex_lock(&lock);
17    printf("mutex is locked\n");
18    clock_gettime(CLOCK_REALTIME, &tout);
```

pthread_mutex_timedlock Fuction: codes/timedlock.c II

```
19      tmp = localtime(&tout.tv_sec);
20      strftime(buf, sizeof(buf), "%r", tmp);
21      printf("current time is %s\n", buf);
22      tout.tv_sec += 10; /* 10 seconds from now */
23      /* caution: this could lead to deadlock */
24      err = pthread_mutex_timedlock(&lock, &tout);
25      clock_gettime(CLOCK_REALTIME, &tout);
26      tmp = localtime(&tout.tv_sec);
27      strftime(buf, sizeof(buf), "%r", tmp);
28      printf("the time is now %s\n", buf);
29      if (err == 0)
30         printf("mutex locked again!\n");
31      else
32         printf("can't lock mutex again: %s\n", strerror(err));
33      exit(0);
34  }
```

This program deliberately locks a mutex it already owns to demonstrate how `pthread_mutex_timedlock` works. This strategy is not recommended in practice, because it can lead to deadlock.

Thread Synchronization : Reader-Writer Locks

Reader-Writer Locks

Reader–writer locks are similar to mutexes, except that they allow for higher degrees of parallelism.

Three states are possible with a reader–writer lock:

1. locked in read mode,
2. locked in write mode,
3. and unlocked.

Only one thread at a time can hold a reader–writer lock in write mode,

- but multiple threads can hold a reader–writer lock in read mode at the same time.

Reader-Writer Locks cnt'd

When a reader–writer lock is write locked,

- all threads attempting to lock it block until it is unlocked.

When a reader–writer lock is read locked,

- all threads attempting to lock it in read mode are given access,
- but any threads attempting to lock it in write mode block until all the threads have released their read locks.

Reader–writer locks are well suited for situations in which data structures are read more often than they are modified.

Reader–writer locks are also called shared–exclusive locks.

Reader-Writer Locks cnt'd

eader–writer locks must be initialized before use and destroyed before freeing their underlying memory.

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
// Both return: 0 if OK, error number on failure
```

A reader–writer lock is initialized by calling `pthread_rwlock_init`.

Before freeing the memory backing a reader–writer lock, we need to call `pthread_rwlock_destroy` to clean it up.

Reader-Writer Locks cnt'd

To lock a reader–writer lock in read mode, we call `pthread_rwlock_rdlock`.

To write lock a reader–writer lock, we call `pthread_rwlock_wrlock`.

Regardless of how we lock a reader–writer lock, we can unlock it by calling `pthread_rwlock_unlock`.

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
// All return: 0 if OK, error number on failure
```

Reader-Writer Locks cnt'd: codes/rwlock.c I

use of reader–writer lock

A queue of job requests is protected by a single reader–writer lock.

```
1  #include <stdlib.h>
2  #include <pthread.h>
3
4  struct job {
5     struct job *j_next;
6     struct job *j_prev;
7     pthread_t j_id; /* tells which thread handles this job */
8     /* ... more stuff here ... */
9  };
10
11 struct queue {
12    struct job *q_head;
13    struct job *q_tail;
14    pthread_rwlock_t q_lock;
15 };
16
17 /*
18  * Initialize a queue.
```

Reader-Writer Locks cnt'd: codes/rwlock.c II

```
19   */
20  int
21  queue_init(struct queue *qp)
22  {
23     int err;
24
25     qp->q_head = NULL;
26     qp->q_tail = NULL;
27     err = pthread_rwlock_init(&qp->q_lock, NULL);
28     if (err != 0)
29        return(err);
30     /* ... continue initialization ... */
31     return(0);
32  }
33
34  /*
35   * Insert a job at the head of the queue.
36   */
37  void
38  job_insert(struct queue *qp, struct job *jp)
39  {
40     pthread_rwlock_wrlock(&qp->q_lock);
41     jp->j_next = qp->q_head;
```

Reader-Writer Locks cnt'd: codes/rwlock.c III

```
42      jp->j_prev = NULL;
43      if (qp->q_head != NULL)
44          qp->q_head->j_prev = jp;
45      else
46          qp->q_tail = jp; /* list was empty */
47      qp->q_head = jp;
48      pthread_rwlock_unlock(&qp->q_lock);
49  }
50
51  /*
52   * Append a job on the tail of the queue.
53   * we lock the queue's readerwriter lock in write mode whenever we need to add
54   * a job to the queue or remove a job from the queue.
55   */
56  void
57  job_append(struct queue *qp, struct job *jp)
58  {
59      pthread_rwlock_wrlock(&qp->q_lock);
60      jp->j_next = NULL;
61      jp->j_prev = qp->q_tail;
62      if (qp->q_tail != NULL)
63          qp->q_tail->j_next = jp;
64      else
```

Reader-Writer Locks cnt'd: codes/rwlock.c IV

```
65          qp->q_head = jp; /* list was empty */
66       qp->q_tail = jp;
67       pthread_rwlock_unlock(&qp->q_lock);
68    }
69
70    /*
71     * Remove the given job from a queue.
72     */
73    void
74    job_remove(struct queue *qp, struct job *jp)
75    {
76       pthread_rwlock_wrlock(&qp->q_lock);
77       if (jp == qp->q_head) {
78          qp->q_head = jp->j_next;
79          if (qp->q_tail == jp)
80             qp->q_tail = NULL;
81          else
82             jp->j_next->j_prev = jp->j_prev;
83       } else if (jp == qp->q_tail) {
84          qp->q_tail = jp->j_prev;
85          jp->j_prev->j_next = jp->j_next;
86       } else {
87          jp->j_prev->j_next = jp->j_next;
```

Reader-Writer Locks cnt'd: codes/rwlock.c V

```
88          jp->j_next->j_prev = jp->j_prev;
89      }
90      pthread_rwlock_unlock(&qp->q_lock);
91  }
92
93  /*
94   * Find a job for the given thread ID.
95   * Whenever we search the queue, we grab the lock in read mode, allowing all
96   * the worker threads to search the queue concurrently.
97   */
98  struct job *
99  job_find(struct queue *qp, pthread_t id)
100 {
101     struct job *jp;
102
103     if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
104         return(NULL);
105
106     for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
107         if (pthread_equal(jp->j_id, id))
108             break;
109
110     pthread_rwlock_unlock(&qp->q_lock);
```

Reader-Writer Locks cnt'd: codes/rwlock.c VI

```
111     return(jp);
112   }
```

Thread Synchronization : Condition Variables

Condition Variables

Condition variables are another synchronization mechanism available to threads.

- When used with mutexes, condition variables allow threads to wait in a race-free way for arbitrary conditions to occur.

The condition itself is protected by a mutex.

- A thread must first lock the mutex to change the condition state.
- Other threads will not notice the change until they acquire the mutex,
- because the mutex must be locked to be able to evaluate the condition.

Condition Variables cnt'd

Before a condition variable is used, it must first be initialized.

A condition variable, represented by the `pthread_cond_t` data type, can be initialized in two ways.

- We can assign the constant `PTHREAD_COND_INITIALIZER` to a statically allocated condition variable,
- if the condition variable is allocated dynamically, we can use the `pthread_cond_init` function to initialize it.

We can use the `pthread_cond_destroy` function to deinitialize a condition variable before freeing its underlying memory.

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
// Both return: 0 if OK, error number on fa
```

Condition Variables cnt'd

`pthread_cond_wait` to wait for a condition to be true

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex,
                      const struct timespec *restrict tsptr);
// Both return: 0 if OK, error number on failure
```

The mutex passed to `pthread_cond_wait` protects the condition.

The caller passes it locked to the function, which then atomically places the calling thread on the list of threads waiting for the condition and unlocks the mutex.

When `pthread_cond_wait` returns, the mutex is again locked.

Condition Variables cnt'd

There are two functions to notify threads that a condition has been satisfied.

- The `pthread_cond_signal` function will wake up at least one thread waiting on a condition,
- whereas the `pthread_cond_broadcast` function will wake up all threads waiting on a condition.

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
// Both return: 0 if OK, error number on f
```

When we call `pthread_cond_signal` or `pthread_cond_broadcast`, we are said to be signaling the thread or condition. We have to be careful to signal the threads only after changing the state of the condition.

Condition Variables cnt'd: codes/condvar.c I

example of how to use a condition variable and a mutex together to synchronize threads.

```
1  #include <pthread.h>
2
3  struct msg {
4     struct msg *m_next;
5     /* ... more stuff here ... */
6  };
7
8  struct msg *workq;
9
10 pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
11
12 pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
13
14 void
15 process_msg(void)
16 {
17    struct msg *mp;
18
```

Condition Variables cnt'd: codes/condvar.c II

```
19      for (;;) {
20          pthread_mutex_lock(&qlock);
21          while (workq == NULL)
22              pthread_cond_wait(&qready, &qlock);
23          mp = workq;
24          workq = mp->m_next;
25          pthread_mutex_unlock(&qlock);
26          /* now process the message mp */
27      }
28  }
29
30  void
31  enqueue_msg(struct msg *mp)
32  {
33      pthread_mutex_lock(&qlock);
34      mp->m_next = workq;
35      workq = mp;
36      pthread_mutex_unlock(&qlock);
37      pthread_cond_signal(&qready);
38  }
```

Condition Variables cnt'd: codes/condvar.c III

We protect the condition with a mutex and evaluate the condition in a while loop.

- When we put a message on the work queue, we need to hold the mutex,
- but we don't need to hold the mutex when we signal the waiting threads.

Since we check the condition in a while loop, this doesn't present a problem;

- a thread will wake up, find that the queue is still empty, and go back to waiting again.

Thread Synchronization : Spin Locks

Spin Locks

A spin lock is like a mutex,

- process is blocked by busy-waiting (spinning) until the lock can be acquired

A spin lock could be used in situations where locks are held for short periods of times and threads don't want to incur the cost of being descheduled.

Spin Locks cnt'd

Spin locks are often used as low-level primitives to implement other types of locks.

while a thread is spinning and waiting for a lock to become available, the CPU can't do anything else.

Spin Locks cnt'd

We can initialize a spin lock with the `pthread_spin_init` function.

To deinitialize a spin lock, we can call the `pthread_spin_destroy` function.

```
#include <pthread.h>
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
// Both return: 0 if OK, error number on fai
```

The *pshared* argument represents the process-shared attribute, which indicates how the spin lock will be acquired

- `PTHREAD_PROCESS_SHARED` - shared among processes
- `PTHREAD_PROCESS_PRIVATE` - shared within processes

Spin Locks cnt'd

To lock the spin lock,

- we can call either `pthread_spin_lock`, which will spin until the lock is acquired,
- or `pthread_spin_trylock`, which will return the `EBUSY` error if the lock can't be acquired immediately.

```
#include <pthread.h>
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
All return: 0 if OK, error number on failure
```

Note that if a spin lock is currently unlocked,

- then the `pthread_spin_lock` function can lock it without spinning.
- If the thread already has it locked or unlocked, the results are undefined.

Thread Synchronization : Barriers

Barriers

Barriers are a synchronization mechanism that can be used to coordinate multiple threads working in parallel.

- A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there.
- the `pthread_join` function acts as a barrier to allow one thread to wait until another thread exits.

They allow an arbitrary number of threads to wait until all of the threads have completed processing,

- but the threads don't have to exit.
- They can continue working after all threads have reached the barrier.

Barriers cnt'd

We can use the `pthread_barrier_init` function to initialize a barrier, we can use the `pthread_barrier_destroy` function to deinitialize a barrier.

```
#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned int count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
// Both return: 0 if OK, error number on failure
```

- *count* argument to specify the number of threads that must reach the barrier before all of the threads will be allowed to continue.
- *attr* argument to specify the attributes of the barrier object (*attr* to NULL = default)
- resoureces initialized with `pthread_barrier_init` must be deinitialized with `pthread_barrier_destroy`

Barriers cnt'd

We use the `pthread_barrier_wait` function to indicate that a thread is done with its work and is ready to wait for all the other threads to catch up.

```
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier);
// Returns: 0 or PTHREAD_BARRIER_SERIAL_THREAD if OK, error number on failure
```

The thread calling `pthread_barrier_wait` is put to sleep

- if the barrier count (set in the call to `pthread_barrier_init`) is not yet satisfied.
- If the thread is the last one to `call pthread_barrier_wait`, thereby satisfying the barrier count, all of the threads are awakened.
- Once the barrier count is reached and the threads are unblocked, the barrier can be used again. The count can't be changed

Barriers cnt'd: code/barrier.c I

note that mac OS does not have pthread_barrier_wait

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5  #include <limits.h>
6  #include <sys/time.h>
7  #include "bar_mac.h"
8
9  #define NTHR 8    /* number of threads */
10 #define NUMNUM 8000000L   /* number of numbers to sort */
11 #define TNUM (NUMNUM/NTHR) /* number to sort per thread */
12
13 long nums[NUMNUM];
14 long snums[NUMNUM];
15
16 pthread_barrier_t b;
17
18 #ifdef SOLARIS
19 #define heapsort qsort
20 #elif linux
```

Barriers cnt'd: code/barrier.c II

```
21  #define heapsort qsort
22  #else
23  extern int heapsort(void *, size_t, size_t,
24                      int (*)(const void *, const void *));
25  #endif
26
27  /*
28   * Compare two long integers (helper function for heapsort)
29   */
30  int
31  complong(const void *arg1, const void *arg2)
32  {
33     long l1 = *(long *)arg1;
34     long l2 = *(long *)arg2;
35
36     if (l1 == l2)
37        return 0;
38     else if (l1 < l2)
39        return -1;
40     else
41        return 1;
42  }
43
```

Barriers cnt'd: code/barrier.c III

```
44  /*
45   * Worker thread to sort a portion of the set of numbers.
46   */
47  void *
48  thr_fn(void *arg)
49  {
50     long idx = (long)arg;
51
52     heapsort(&nums[idx], TNUM, sizeof(long), complong);
53     pthread_barrier_wait(&b);
54
55     /*
56      * Go off and perform more work ...
57      */
58     return((void *)0);
59  }
60
61  /*
62   * Merge the results of the individual sorted ranges.
63   */
64  void
65  merge()
66  {
```

Barriers cnt'd: code/barrier.c IV

```
67      long idx[NTHR];
68      long i, minidx, sidx, num;
69
70      for (i = 0; i < NTHR; i++)
71         idx[i] = i * TNUM;
72      for (sidx = 0; sidx < NUMNUM; sidx++) {
73         num = LONG_MAX;
74         for (i = 0; i < NTHR; i++) {
75            if ((idx[i] < (i+1)*TNUM) && (nums[idx[i]] < num)) {
76               num = nums[idx[i]];
77               minidx = i;
78            }
79         }
80         snums[sidx] = nums[idx[minidx]];
81         idx[minidx]++;
82      }
83   }
84
85   int
86   main()
87   {
88      unsigned long i;
89      struct timeval start, end;
```

Barriers cnt'd: code/barrier.c V

```
90      long long  startusec, endusec;
91      double   elapsed;
92      int    err;
93      pthread_t  tid;
94
95      /*
96       * Create the initial set of numbers to sort.
97       */
98      srandom(1);
99      for (i = 0; i < NUMNUM; i++)
100        nums[i] = random();
101
102     /*
103      * Create 8 threads to sort the numbers.
104      */
105     gettimeofday(&start, NULL);
106     pthread_barrier_init(&b, NULL, NTHR+1);
107     for (i = 0; i < NTHR; i++) {
108        err = pthread_create(&tid, NULL, thr_fn, (void *)(i * TNUM));
109        if (err != 0){
110           fprintf(stderr, "can't create thread");
111        }
112     }
```

Barriers cnt'd: code/barrier.c VI

```
113     pthread_barrier_wait(&b);
114     merge();
115     gettimeofday(&end, NULL);
116
117     /*
118      * Print the sorted list.
119      */
120     startusec = start.tv_sec * 1000000 + start.tv_usec;
121     endusec = end.tv_sec * 1000000 + end.tv_usec;
122     elapsed = (double)(endusec - startusec) / 1000000.0;
123     printf("sort took %.4f seconds\n", elapsed);
124     for (i = 0; i < NUMNUM; i++)
125        printf("%ld\n", snums[i]);
126     exit(0);
127   }
```

Barriers cnt'd