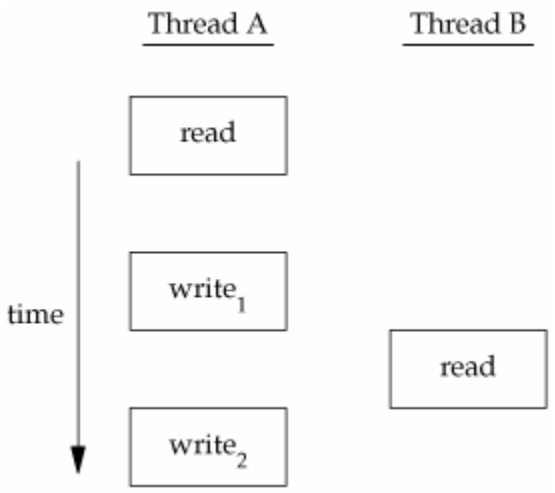
Lecture 16. Thread Synchronization

Thread Synchronization

When multiple threads share the same memory, each thread has to access the data in a consistent fashion, such that each thread runs to completion, and the read/write of data does not happen interspersed. The threads must not be allowed to write each other's data out of turn.

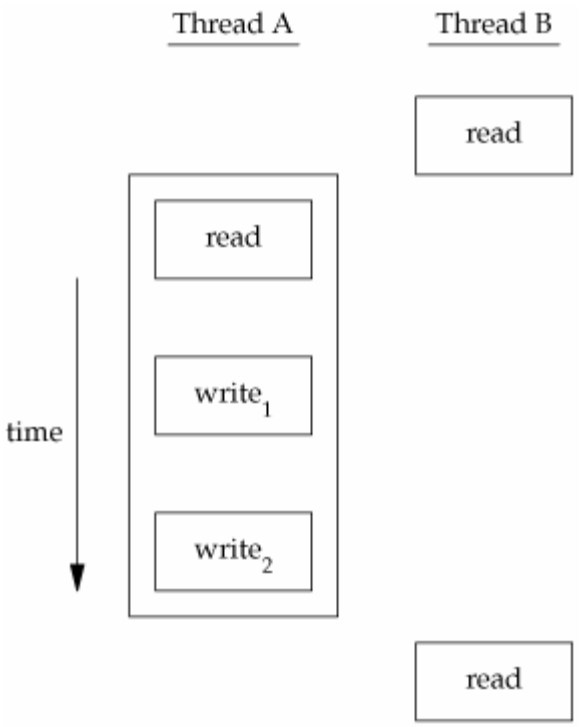
Example 1: The following diagram shows an example of two threads reading and writing the same variable. Thread A writes a value in two memory cycles. If Thread B attempts to read the same value in between these cycles, it will read an inconsistent value, perhaps the value before the write update.



To avoid this situation, threads must use a **lock**, which will allow only one of them to have access to the variable at a time. The following diagram shows what ideally has to happen: the read-write1-write2 cycles must happen *atomically*. Each thread attempting to read a given variable, acquires a lock. When it is done writing the variable, it releases the lock to the other thread. The ability to lock around reading a variable must hold across two or more threads.

Example 2: Consider a variable being incremented. There are three steps:
1. Read the memory address into a register
2. Increment the value in the register
3. Write back the incremented value in to the memory address

If two threads increment this variable almost at the same time, the results can be inconsistent. Depending on the value read by the second thread when it begins its increment operation, the resulting value could be +1 or +2.



If the increment is atomic, and the data is **sequentially consistent** (i.e. produces consistent results if the operations on it are performed in sequence), there is no need for additional synchronization. However, especially on modern hardware, there is no guarantee that the data is sequentially consistent, since write operations, taking multiple clock cycles, can also be separated into clock cycles distributed across multiple processors.

Besides hardware reasons for lack of consistency, there are also software reasons, arising from instructions that are not atomically executed.

Example 3: Assume code that assigns a value to a variable, then makes a decision based on the value of that variable. If another thread modifies that variable in between the assignment and the decision, then, when the first thread returns, the decision will go the wrong way.

Mutexes

Mutexes are locks that protect the shared resource across multiple threads attempting to access it. While the mutex (lock) is set, any other thread that tries to set it will block until the mutex is released. If multiple threads are waiting for a mutex to unlock, the first one to run will acquire access to the lock and set it again. The other threads will wait in line until the mutex (lock) is released again etc.

A mutex variable is represented by the pthread_mutex_t data type. Before using it, it must be initialized to PTHREAD_MUTEX_INITIALIZER (for statically allocated mutexes), or we can call `pthread_mutex_init()` (for statically and dynamically allocated mutexes). For the latter type, if `malloc()` was called, for example, we have to call its pair, `pthread_mutex_destroy()`:

```
#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, `errno` if failure.

To set the mutex attributes to the default, we pass a NULL value for the second argument.

To set (lock)/unset (unlock) a mutex, we call the following functions:

```
#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, errno if failure.

If a thread can't block on finding the mutex locked by another thread, it can use `pthread_mutex_trylock()` to lock the mutex conditionally. When this function is called, if the mutex is unlocked, the calling thread will lock the mutex without

blocking and return 0. If the mutex is blocked, the function will return errno=EBUSY without locking it.

The following program creates two threads writing to the same file without synchronization:

```
#include <pthread.h>
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for read(), write(), sleep() */
#include <stdlib.h> /* for calloc() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */

pthread_t tid1, tid2;
void *tret1, *tret2;
int fd;

void *start_rtn1(void *arg);
void *start_rtn2(void *arg);

int main(void)
{
     int err;
     fd=open("/Users/awise/Stevens/Lecture16/file1.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

     err=pthread_create(&tid1, NULL, start_rtn1, NULL);
     if (err!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid1);

     err=pthread_create(&tid2, NULL, start_rtn2, NULL);
     if (err!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid2);
     err=pthread_join(tid1, &tret1); /* thread 1 joins the main
thread */
     err=pthread_join(tid2, &tret2); /* thread 2 joins the main
thread */
     printf("Thread %d has exit code %ld.\n", (int) tid1, (long)
tret1);
     printf("Thread %d has exit code %ld.\n", (int) tid2, (long)
tret2);
return 0;
}
```

```
void *start_rtn1(void *arg)
{
     printf("Doing stuff in thread 1...\n");
        char **name=calloc(3, 20*sizeof(char));
        char **it;
        name[0]="Adriana Wise\n";
        name[1]="Richard Stevens\n";
        name[2]="Evi Nemeth\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 1 returning...\n");
pthread_exit ((void *) 1);
}

void *start_rtn2(void *arg)
{
     printf("Doing stuff in thread 2...\n");
        char **name=calloc(4, 20*sizeof(char));
        char **it;
        name[0]="William Sakas\n";
        name[1]="Susan Epstein\n";
        name[2]="Stewart Weiss\n";
        name[3]="Subash Shankar\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 2 returning...\n");
pthread_exit ((void *) 2);
}
```

The outputs from two different runs show clearly that there is no synchronization between the threads:

```
$ more file1.txt
Adriana Wise
William Sakas
Richard Stevens
Susan Epstein
Evi Nemeth
Stewart Weiss
Subash Shankar
```

```
$ more file1.txt
William Sakas
Adriana Wise
Susan Epstein
Richard Stevens
Evi Nemeth
Stewart Weiss
Subash Shankar
```

Observe the order of execution and exit from the command line output as well:

```
$ ./mythreadjoin
Doing stuff in thread 1...
Doing stuff in thread 2...
Thread 1 returning...
Thread 2 returning...
Thread 56086528 has exit code 1.
Thread 56623104 has exit code 2.
```

The following revised code uses a mutex (lock) to protect each thread against "intrusion" of the shared resources (the output text file) by the other thread:

```
#include <pthread.h>
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for read(), write(), sleep() */
#include <stdlib.h> /* for calloc() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen(), strerr() */

pthread_t tid1, tid2;
void *tret1, *tret2;
int fd;
pthread_mutex_t *lock;

void *start_rtn1(void *arg);
void *start_rtn2(void *arg);

int main(void)
{
     int err;
     lock=malloc(sizeof(pthread_mutex_t));
     err=pthread_mutex_init(lock, NULL);
     if (err!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err));
     fd=open("/Users/awise/Stevens/Lecture16/file2.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

     err=pthread_create(&tid1, NULL, start_rtn1, NULL);
     if (err!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid1);

     err=pthread_create(&tid2, NULL, start_rtn2, NULL);
```

```
     if (err!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid2);

     err=pthread_join(tid1, &tret1); /* thread 1 joins the main
thread */
     err=pthread_join(tid2, &tret2); /* thread 2 joins the main
thread */
     printf("Thread %d has exit code %ld.\n", (int) tid1, (long)
tret1);
     printf("Thread %d has exit code %ld.\n", (int) tid2, (long)
tret2);
     err=pthread_mutex_destroy(lock);
return 0;
}

void *start_rtn1(void *arg)
{
     int err1=pthread_mutex_lock(lock);
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
     printf("Doing stuff in thread 1...\n");
        char **name=calloc(3, 20*sizeof(char));
        char **it;
        name[0]="Adriana Wise\n";
        name[1]="Richard Stevens\n";
        name[2]="Evi Nemeth\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 1 returning...\n");
     err1=pthread_mutex_unlock(lock);
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
pthread_exit ((void *) 1);
}

void *start_rtn2(void *arg)
{
     int err2=pthread_mutex_lock(lock);
     if (err2!=0)
```

```
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     printf("Doing stuff in thread 2...\n");
        char **name=calloc(4, 20*sizeof(char));
        char **it;
        name[0]="William Sakas\n";
        name[1]="Susan Epstein\n";
        name[2]="Stewart Weiss\n";
        name[3]="Subash Shankar\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 2 returning...\n");
     err2=pthread_mutex_unlock(lock);
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
pthread_exit ((void *) 2);
}
```

The output file shows the threads executing in order:

```
$ more file2.txt
Adriana Wise
Richard Stevens
Evi Nemeth
William Sakas
Susan Epstein
Stewart Weiss
Subash Shankar
```

The command line output shows the same thing:

```
$ ./mymutex
Doing stuff in thread 1...
Thread 1 returning...
Doing stuff in thread 2...
Thread 2 returning...
Thread 239394816 has exit code 1.
Thread 239931392 has exit code 2.
```

Deadlocks

A thread will deadlock itself if it tries to lock the same mutex twice. (We can try this with our program, in the start routine for Thread 1.) Deadlocks can be created, however, in multiple ways. For example, Thread 1 holds mutex1 and tries to lock mutex2, while Thread 2 holds mutex2 and tries to lock mutex1.

Deadlocks can be avoided by controlling the ordering in which mutexes are locked. If we have two mutexes we need to lock at the same time, if all threads lock mutex1 before mutex2, no deadlock can occur on account of using these two mutexes. There is potential for a deadlock if one thread attempts to lock the mutexes in the opposite order from the other thread.

Some architectures make it hard to impose a lock ordering. Using `pthread_mutex_trylock()` may help avoid deadlocking. If there are locks set, and the call to `pthread_mutex_trylock()` is successful, the thread calling it may proceed. But if the call fails, it can't proceed until the current lock(s) is/are released.

I modified the example from earlier to include two mutexes in each start routine for the two threads, but unless I took out the release of the locks, it seemed to be still working every time:

```
#include <pthread.h>
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for read(), write(), sleep() */
#include <stdlib.h> /* for calloc() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen(), strerr() */

pthread_t tid1, tid2;
void *tret1, *tret2;
int fd;
pthread_mutex_t *lock1, *lock2;

void *start_rtn1(void *arg);
void *start_rtn2(void *arg);

int main(void)
{
     int err1, err2;
     lock1=malloc(sizeof(pthread_mutex_t));
     lock2=malloc(sizeof(pthread_mutex_t));
```

```
     err1=pthread_mutex_init(lock1, NULL);
     err2=pthread_mutex_init(lock2, NULL);
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     fd=open("/Users/awise/Stevens/Lecture16/file4.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

     err1=pthread_create(&tid1, NULL, start_rtn1, NULL);
     if (err1!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid1);

     err2=pthread_create(&tid2, NULL, start_rtn2, NULL);
     if (err2!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid2);

     //sleep(2);
     err1=pthread_join(tid1, &tret1); /* thread 1 joins the main
thread */
     err2=pthread_join(tid2, &tret2); /* thread 2 joins the main
thread */
     printf("Thread %d has exit code %ld.\n", (int) tid1, (long)
tret1);
     printf("Thread %d has exit code %ld.\n", (int) tid2, (long)
tret2);
     err1=pthread_mutex_destroy(lock1);
     err2=pthread_mutex_destroy(lock2);
return 0;
}

void *start_rtn1(void *arg)
{
     int err1=pthread_mutex_lock(lock1);
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
     err1=pthread_mutex_lock(lock2);
     printf("Doing stuff in thread 1...\n");
        char **name=calloc(3, 20*sizeof(char));
        char **it;
        name[0]="Adriana Wise\n";
```

```
        name[1]="Richard Stevens\n";
        name[2]="Evi Nemeth\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 1 returning...\n");
     //err1=pthread_mutex_unlock(lock1);
     //err1=pthread_mutex_unlock(lock2);
pthread_exit ((void *) 1);
}

void *start_rtn2(void *arg)
{
     int err2=pthread_mutex_lock(lock1);
     err2=pthread_mutex_lock(lock2);
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     printf("Doing stuff in thread 2...\n");
        char **name=calloc(4, 20*sizeof(char));
        char **it;
        name[0]="William Sakas\n";
        name[1]="Susan Epstein\n";
        name[2]="Stewart Weiss\n";
        name[3]="Subash Shankar\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 2 returning...\n");
     //err2=pthread_mutex_unlock(lock2);
     //err2=pthread_mutex_unlock(lock1);
pthread_exit ((void *) 2);
}
```

The output shows a deadlock, and the program must be stopped with `Ctrl-C`:

```
$ ./mydeadlock2
Doing stuff in thread 1...
Thread 1 returning...
^C
```

The `pthread_mutex_timedlock()` Function

This is a variant of the `pthread_mutex_lock()`, but with a timer. When the timer expires, `pthread_mutex_timedlock()` will return the error code ETIMEDOUT without locking the mutex. It is used to put an upper bound on the time a thread is blocked while waiting on another thread to release the mutex.

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

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

Returns 0 if OK, `errno` if failure.

This example shows the use of `pthread_mutex_timedlock()`. However, this system call is not supported on MacOS X, so it remains a theoretical example:

```
#include <pthread.h>
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for read(), write(), sleep() */
#include <stdlib.h> /* for calloc() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen(), strerr() */
#include <time.h>

pthread_t tid1, tid2;
void *tret1, *tret2;
int fd;
pthread_mutex_t *lock;
struct timespec *tsptr;

void *start_rtn1(void *arg);
void *start_rtn2(void *arg);

int main(void)
{
     int err;
     lock=malloc(sizeof(pthread_mutex_t));
     err=pthread_mutex_init(lock, NULL);
     if (err!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err));
```

```
     fd=open("/Users/awise/Stevens/Lecture16/file6.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

     err=pthread_create(&tid1, NULL, start_rtn1, NULL);
     if (err!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid1);

     err=pthread_create(&tid2, NULL, start_rtn2, NULL);
     if (err!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid2);

     //sleep(2);
     err=pthread_join(tid1, &tret1); /* thread 1 joins the main
thread */
     err=pthread_join(tid2, &tret2); /* thread 2 joins the main
thread */
     printf("Thread %d has exit code %ld.\n", (int) tid1, (long)
tret1);
     printf("Thread %d has exit code %ld.\n", (int) tid2, (long)
tret2);
     err=pthread_mutex_destroy(lock);
return 0;
}

void *start_rtn1(void *arg)
{
     int err1=pthread_mutex_timedlock(lock, tsptr);
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
     printf("Doing stuff in thread 1...\n");
        char **name=calloc(3, 20*sizeof(char));
        char **it;
        name[0]="Adriana Wise\n";
        name[1]="Richard Stevens\n";
        name[2]="Evi Nemeth\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 1 returning...\n");
     err1=pthread_mutex_unlock(lock);
     if (err1!=0)
```

```
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
pthread_exit ((void *) 1);
}

void *start_rtn2(void *arg)
{
     int err2=pthread_mutex_lock(lock);
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     printf("Doing stuff in thread 2...\n");
        char **name=calloc(4, 20*sizeof(char));
        char **it;
        name[0]="William Sakas\n";
        name[1]="Susan Epstein\n";
        name[2]="Stewart Weiss\n";
        name[3]="Subash Shankar\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
                sleep(2);
        }
     printf("Thread 2 returning...\n");
     err2=pthread_mutex_unlock(lock);
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
pthread_exit ((void *) 2);
}
```

Reader-Writer Locks

With a mutex, the state of a thread is either locked or unlocked, and only one thread at a time can lock it. A reader-writer lock allows three states:

- locked in read mode
- locked in write mode
- unlocked

Only one thread at a time can hold a reader-writer lock locked in write mode, while several threads can hold a reader-writer lock in read mode.

They are used in situations where data structures need more frequent reading than modifying. They are also called **shared-exclusive locks**.

As with their two-state counterparts, r/w locks must be initialized and destroyed:

```
#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, `errno` on failure.

For a statically allocated reader-writer lock, we can initialize it with the constant PTHREAD_RWLOCK_INITIALIZER.

To lock a mutex in reader mode, we call `pthread_rwlock_rdlock()`. To lock a mutex in writer mode, we call `pthread_rwlock_wrlock()`. Either can be unlocked using the `pthread_rwlock_unlock()` function:

```
#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, `errno` if failure.

The following example creates a writer thread and reader thread. When the writer thread sets a **read lock**, the other thread attempting to lock it in **read mode** is NOT given access. But when the writer thread sets a **write lock**, the reader thread trying to lock it in **read mode** IS given read access. The reader thread writes what it reads into a file—this is how we test if it had read access:

```
#include <pthread.h>
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for read(), write(), sleep() */
#include <stdlib.h> /* for calloc() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen(), strerr() */

#define BUFFSIZE 20
pthread_t tid1, tid2;
```

```
void *tret1, *tret2;
int fd, fd2;
pthread_rwlock_t *rwlock1, *rwlock2;

void *start_rtn1(void *arg);
void *start_rtn2(void *arg);

int main(void)
{
     int err1, err2;
     rwlock1=malloc(sizeof(pthread_rwlock_t));
     rwlock2=malloc(sizeof(pthread_rwlock_t));
     err1=pthread_rwlock_init(rwlock1, NULL);
     err2=pthread_rwlock_init(rwlock2, NULL);
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     fd=open("/Users/awise/Stevens/Lecture16/file5.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     fd2=open("/Users/awise/Stevens/Lecture16/file55.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

     err1=pthread_create(&tid1, NULL, start_rtn1, NULL);
     if (err1!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid1);

     err2=pthread_create(&tid2, NULL, start_rtn2, NULL);
     if (err2!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid2);

     //sleep(2);
     err1=pthread_join(tid1, &tret1); /* thread 1 joins the main
thread */
     err2=pthread_join(tid2, &tret2); /* thread 2 joins the main
thread */
     printf("Thread %d has exit code %ld.\n", (int) tid1, (long)
tret1);
     printf("Thread %d has exit code %ld.\n", (int) tid2, (long)
tret2);
     err1=pthread_rwlock_destroy(rwlock1);
     err2=pthread_rwlock_destroy(rwlock2);
```

```
return 0;
}

/* writer thread */
void *start_rtn1(void *arg)
{
     int err1=pthread_rwlock_wrlock(rwlock1); /* writer set a
write lock */
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err1));
     printf("Doing stuff in thread 1...\n");
        char **name=calloc(3, 20*sizeof(char));
        char **it;
        name[0]="Adriana Wise\n";
        name[1]="Richard Stevens\n";
        name[2]="Evi Nemeth\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd, *it, strlen(*it));
        }
     printf("Thread 1 returning...\n");
     err1=pthread_rwlock_unlock(rwlock1);
pthread_exit ((void *) 1);
}

/* reader thread */
void *start_rtn2(void *arg)
{
     int err2=pthread_rwlock_rdlock(rwlock2);
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     printf("Doing stuff in thread 2...\n");

     //want to READ from fd and write to fd2
     char buff[BUFFSIZE];
     ssize_t bytes;
     int i=0;
     while ((bytes=read(fd, buff, BUFFSIZE))>0)
     {
          if ((write(STDOUT_FILENO, buff, bytes))!=bytes)
               write(STDOUT_FILENO, "Write error\n", 20);
          lseek(fd2, i*20, SEEK_SET);
          write(fd2, buff, bytes);
          i++;
```

```
        }
        printf("Thread 2 returning...\n");
        err2=pthread_rwlock_unlock(rwlock2);
pthread_exit ((void *) 2);
}
```

The output shows that thread 2 was given access and could write what it read both at the command line and into the output text file:

```
$ ./myreaderwriter
Doing stuff in thread 2...
Adriana Wise
Richard Stevens
Evi Nemeth
Doing stuff in thread 1...
Thread 2 returning...
Thread 1 returning...
Thread 243245056 has exit code 1.
Thread 243781632 has exit code 2.
$ more file55.txt
Adriana Wise
Richard Stevens
Evi Nemeth
```

Reader-Writer Locking with Timeouts

Just like mutexes, reader-writer locks can be locked with a timeout, to avoid blocking a thread indefinitely while trying to acquire a lock:

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

int pthread_rwlock_timedrdlock(
          pthread_rwlock_t *restrict rwlock,
          const struct timespec *restrict tsptr);

int pthread_rwlock_timedwrlock(
          pthread_rwlock_t *restrict rwlock,
          const struct timespec *restrict tsptr);
```

Both return 0 if OK, `errno` if failure.

Spin Locks

Spin locks are like mutexes, except that instead of blocking a process by sleeping, they block the process in a state of busy-waiting ("spinning") until the lock can be acquired. They are used in situations where we don't want a thread to be put to sleep just waiting on a lock of a very short duration.

```
#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, `errno` if failure.

The pshared argument holds the process-shared attribute, which indicates how the spin-lock can be acquired:

- if pshared==PTHREAD_PROCESS_SHARED, the spin-lock can be acquired by threads that have access to the lock's underlying memory, even from different processes;
- if pshared==PTHREAD_PROCESS_PRIVATE, the spin-lock can be acquired only from threads from the same process.

The lock, trylock, and unlock functions are the same:

```
#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, `errno` if failure.

Barriers

Barriers are a synchronization mechanism that can be used to coordinate multiple threads working in parallel. A barrier allows a thread to wait until all cooperating threads have reached the same point, and then continue executing from there. The `pthread_join()` function we examined in Lecture 15 acted as a barrier to allow one thread to wait until another thread exits.

The following two functions initialize and destroy 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, `errno` if failure.

The count argument specifies the number of threads that must reach the barrier before all threads will be allowed again to continue. The attr argument can be set to NULL for the default settings.

The `pthread_barrier_wait()` function is used to indicate that a thread has exited and is waiting for 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, `errno` if failure.

The thread calling `pthread_barrier_wait()` is put to sleep if the barrier count is not yet satisfied. If the caller thread is the last one, then all threads will be awakened and will continue to execute.

The one arbitrary thread which will read a return value of PHTREAD_… will continue as the master thread to act on the results of the work done by all the other threads. All these other threads will see a return value of 0.

The following example has two threads writing into two different files, with the master thread reading from the files and writing into a third one [the buffers read from each of the two source files]. The master thread calls `pthread_barrier_wait()` to wait for the two worker threads to complete their execution in order.

```
#include <pthread.h>
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for read(), write(), sleep() */
#include <stdlib.h> /* for calloc() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen(), strerr() */
#include "barrier.h" /* for pthread_barrier_t, online fix */

#define BUFFSIZE 4096

pthread_t tid1, tid2;
void *tret1, *tret2;
int fd, fd1, fd2;
pthread_barrier_t *barrier, *barrier1, *barrier2;
pthread_barrierattr_t *attr;

void *start_rtn1(void *arg);
void *start_rtn2(void *arg);

char **names1, **names2;

int main(void)
{
     int err, err1, err2;
     names1=calloc(3, 20*sizeof(char));
     names2=calloc(4, 20*sizeof(char));
     barrier=malloc(sizeof(pthread_barrier_t));
     barrier1=malloc(sizeof(pthread_barrier_t));
     barrier2=malloc(sizeof(pthread_barrier_t));
     attr=malloc(sizeof(pthread_barrierattr_t));
     err=pthread_barrier_init(barrier, attr, 1);
     err1=pthread_barrier_init(barrier1, attr, 1);
     err2=pthread_barrier_init(barrier2, attr, 1);
     if (err!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err));
     fd1=open("/Users/awise/Stevens/Lecture16/file6.1.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     fd2=open("/Users/awise/Stevens/Lecture16/file6.2.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     fd=open("/Users/awise/Stevens/Lecture16/file6.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

     err=pthread_create(&tid1, NULL, start_rtn1, NULL);
     if (err!=0)
```

```
          printf("Error: thread %d could not be created.\n",
(int) tid1);

     err=pthread_create(&tid2, NULL, start_rtn2, NULL);
     if (err!=0)
          printf("Error: thread %d could not be created.\n",
(int) tid2);

     err=pthread_join(tid1, &tret1); /* thread 1 joins the main
thread */
     err=pthread_join(tid2, &tret2); /* thread 2 joins the main
thread */

     /* master thread reads from each file and outputs to a 3rd
file */
     printf("After call to pthread_barrier_wait()... err=%d, %s
\n", err, strerror(err));
     lseek(fd1, 0, SEEK_SET);
     char buff1[BUFFSIZE];
     int bytes1=read(fd1, buff1, BUFFSIZE);
     err=pthread_barrier_wait(barrier);
     lseek(fd2, 0, SEEK_SET);
     char buff2[BUFFSIZE];
     int bytes2=read(fd2, buff2, BUFFSIZE);
        printf("After having waited for both threads...\n");
        write(STDOUT_FILENO, buff1, bytes1);
        write(STDOUT_FILENO, buff2, bytes2);
        //lseek(fd2, i*20, SEEK_SET);
     char *buff;
     //buff=strcat(buff1, buff2);
        write(fd, buff1, bytes1);
        write(fd, buff2, bytes2);

     printf("Thread %d has exit code %ld.\n", (int) tid1, (long)
tret1);
     printf("Thread %d has exit code %ld.\n", (int) tid2, (long)
tret2);
     err=pthread_barrier_destroy(barrier);
     err=pthread_barrier_destroy(barrier1);
     err=pthread_barrier_destroy(barrier2);
return 0;
}

void *start_rtn1(void *arg)
{
     printf("Doing stuff in thread 1...\n");
```

```
        char **name=calloc(3, 20*sizeof(char));
        char **it;
        name[0]="Adriana Wise\n";
        name[1]="Richard Stevens\n";
        name[2]="Evi Nemeth\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd1, *it, strlen(*it));
        }
     int err=pthread_barrier_wait(barrier);
     sleep(5);
        name[0]="Bridget Wise\n";
        name[1]="Rosanna Wise\n";
        name[2]="Shannon Wise\n";
        sleep(2);
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd1, *it, strlen(*it));
        }
     err=pthread_barrier_wait(barrier);
     printf("Thread 1 returning...\n");
pthread_exit ((void *) 1);
}

void *start_rtn2(void *arg)
{
     printf("Doing stuff in thread 2...\n");
        char **name=calloc(4, 20*sizeof(char));
        char **it;
        name[0]="William Sakas\n";
        name[1]="Susan Epstein\n";
        name[2]="Stewart Weiss\n";
        name[3]="Subash Shankar\n";
        sleep(2);
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd2, *it, strlen(*it));
        }
     int err=pthread_barrier_wait(barrier);
        name[0]="James Wise\n";
        name[1]="D'Arcy Wise\n";
        name[2]="Ian Wise\n";
        name[3]="Enid Wise\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(fd2, *it, strlen(*it));
```

```
        }
     err=pthread_barrier_wait(barrier);
     printf("Thread 2 returning...\n");
pthread_exit ((void *) 2);
}
```

This program produced the following output:

```
$ ./mybarrier
Doing stuff in thread 2...
Doing stuff in thread 1...
1 threads remaining
1 threads remaining
1 threads remaining
Thread 2 returning...
1 threads remaining
Thread 1 returning...
After call to pthread_barrier_wait()... err=0, Undefined error:
0
1 threads remaining
After having waited for both threads...
Adriana Wise
Richard Stevens
Evi Nemeth
Bridget Wise
Rosanna Wise
Shannon Wise
William Sakas
Susan Epstein
Stewart Weiss
Subash Shankar
James Wise
D'Arcy Wise
Ian Wise
Enid Wise
Thread 59731968 has exit code 1.
Thread 60268544 has exit code 2.
```

Also, the implementation of barriers is based on the following header file, `barrier.h`, since barriers are not supported on MacOS X:

```
#ifdef __APPLE__

#ifndef PTHREAD_BARRIER_H_
#define PTHREAD_BARRIER_H_
```

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

typedef int pthread_barrierattr_t;
typedef struct
{
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int count;
    int tripCount;
} pthread_barrier_t;


int pthread_barrier_init(pthread_barrier_t *barrier, const
pthread_barrierattr_t *attr, unsigned int count)
{
    if(count == 0)
    {
        errno = EINVAL;
        return -1;
    }
    if(pthread_mutex_init(&barrier->mutex, 0) < 0)
    {
        return -1;
    }
    if(pthread_cond_init(&barrier->cond, 0) < 0)
    {
        pthread_mutex_destroy(&barrier->mutex);
        return -1;
    }
    barrier->tripCount = count;
    barrier->count = 0;

    return 0;
}

int pthread_barrier_destroy(pthread_barrier_t *barrier)
{
    pthread_cond_destroy(&barrier->cond);
    pthread_mutex_destroy(&barrier->mutex);
    return 0;
}

int pthread_barrier_wait(pthread_barrier_t *barrier)
{
```

```
    pthread_mutex_lock(&barrier->mutex);
    ++(barrier->count);
    printf("%d threads remaining\n", barrier->count);
    if(barrier->count >= barrier->tripCount)
    {
        barrier->count = 0;
        pthread_cond_broadcast(&barrier->cond);
        pthread_mutex_unlock(&barrier->mutex);
        return 1;
    }
    else
    {
        pthread_cond_wait(&barrier->cond, &(barrier->mutex));
        pthread_mutex_unlock(&barrier->mutex);
        return 0;
    }
}

#endif // PTHREAD_BARRIER_H_
#endif // __APPLE__
```