Lecture 17. Thread Attributes

In general, the functions for managing attributes for objects (threads, processes) follow the same pattern:

1. Each object has its own type of attribute: threads have thread attributes, mutexes have mutex attributes etc.
2. Attributes are set to their default values by calling an initialization function.
3. Attributes are deallocated by calling a destroy function.
4. Each attribute has a function to get the value of the attribute from the attribute structure. This attribute value is passed to the getter function by reference.
5. Each attribute has a function to set the value of the attribute. This attribute value is passed to the setter function by value.

The following two function initialize and destroy a thread attribute object. The thread attribute object was one of the arguments to the `pthread_create()` function, which we initialized to the default values by passing NULL instead. With the `init` function, we can later customize the attributes:

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_h *attr);

int pthread_attr_destroy(pthread_attr_t *attr);
```

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

The attributes are summarized in the following table (ob=obsolete):

Name	Description	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
detachstate	detached thread attribute	•	•	•	•
guardsize	guard buffer size in bytes at end of thread stack		•	•	•
stackaddr	lowest address of thread stack	ob	•	•	ob
stacksize	size in bytes of thread stack	•	•	•	•

A thread whose memory resources we want released immediately upon the thread exiting, without waiting for its exit status, can be created as “detached”.

The attribute that starts out a thread in detached state is `detachstate`, for which purpose it can be set to PTHREAD_CREATE_DETACHED by calling the setter below. The other possible value is PTHREAD_CREATE_JOINABLE. Both values can be retrieved with the getter:

```
#include <pthread.h>

int pthread_attr_getdetachstate(
          const pthread_attr_t *restrict attr,
          int *detachstate);

int pthread_attr_setdetachstate(
          pthread_attr_t *attr,
          int detachstate);
```

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

The following program calls the setter to set *`detachstate`* to each of the two possible values in order, with two different outputs. We only insert the program once, and highlight the attribute that needs changing:

```
#include <pthread.h>
#include <stdlib.h> /* for malloc() */
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for STDIO_FILENO and write() */
#include <string.h> /* for strlen() */

void *start_rtn(void *arg);

int main(void)
{
     int err;
     pthread_t *tid=malloc(sizeof(pthread_attr_t));
     pthread_attr_t *attr=malloc(sizeof(pthread_attr_t));
     int *detachstate=malloc(sizeof(int));

     err=pthread_attr_init(attr);
     if (err!=0)
          return err;
     err=pthread_attr_setdetachstate(attr,
PTHREAD_CREATE_JOINABLE);
     if (err==0)
          err=pthread_create(tid, attr, start_rtn, NULL);
     pthread_attr_getdetachstate(attr, detachstate);
```

```
     printf("The detachstate attribute is %d.\n", *detachstate);
     pthread_attr_destroy(attr);
return err;
}

void *start_rtn(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(STDOUT_FILENO, *it,
strlen(*it));
        }
        printf("Thread 1 returning...\n");
pthread_exit ((void *) 1);
}
```

The output when `detachstate` is set to PTHREAD_CREATE_DETACH is:

```
$ ./mydetachattr
The detachstate attribute is 2.
Doing stuff in thread 1...
Adriana Wise
Richard Stevens
Evi Nemeth
Thread 1 returning...
```

The output when `detachstate` is set to PTHREAD_CREATE_JOINABLE is:

```
$ ./mydetachattr
The detachstate attribute is 1.
Doing stuff in thread 1...
```

These values are #defined in `/usr/include/pthread.h`:

```
#define PTHREAD_CREATE_JOINABLE       1
#define PTHREAD_CREATE_DETACHED       2
```

If the system supports thread stack attributes, the following are defined:

```
#include <pthread.h>

int pthread_attr_getstack(
          const pthread_attr_t *restrict attr,
          void **restrict stackaddr,
          size_t *restrict stacksize);

int pthread_attr_setstack(
          pthread_attr_t *attr,
          void *stackaddr, size_t stacksize);
```

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

When a process has multiple threads, the process' virtual address space is shared among the threads, each one acquiring its own stack. There are two situations when the thread stack size (variable ***`stacksize`***) has to be adjusted:

- diminished under the default value, if there are a large number of threads, such that the total stack size across multiple threads will not exceed the virtual address space of the entire process;
- increased above the default value, if there are allocations of large variables, or calls to functions many stacks deep, such that the default thread stack size will be large enough.

The ***`stackaddr`*** parameter is the lowest addressable address in the range of memory to be used by the thread's stack. However, this is not the start of the stack. If on a particular architecture addresses grow from higher to lower, ***`stackaddr`*** can become the end of the stack.

As an alternative to getting/setting ***`stacksize`*** by passing it by reference to `pthread_attr_getstack()`/by value to `pthread_attr_setstack()`, the following two functions can be used instead:

```
#include <pthread.h>

int pthread_attr_getstacksize(
          const pthread_attr_t *restrict attr,
          size_t *restrict stacksize);
```

Returns 0 if OK, `errno` if failure.

```
#include <pthread.h>

int pthread_attr_setstacksize(
          pthread_attr_t *attr, size_t stacksize);
```
Returns 0 if OK, `errno` if failure.

The following program illustrates the use of the former pair of functions to retrieve these two thread attributes, ***stacksize*** and ***stackaddr***:

```
#include <pthread.h>
#include <stdlib.h> /* for malloc() */
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for STDIO_FILENO and write() */
#include <string.h> /* for strlen() */

void *start_rtn(void *arg);

int main(void)
{
     int err;
     pthread_t *tid=malloc(sizeof(pthread_attr_t));
     pthread_attr_t *attr=malloc(sizeof(pthread_attr_t));
     int *detachstate=malloc(sizeof(int));
     void **stackaddr=calloc(sizeof(void), sizeof(void));
     size_t *stacksize=malloc(sizeof(size_t));

     err=pthread_attr_init(attr);
     if (err!=0)
          return err;
     err=pthread_attr_setdetachstate(attr,
PTHREAD_CREATE_DETACHED);
     if (err==0)
          err=pthread_create(tid, attr, start_rtn, NULL);
     pthread_attr_getdetachstate(attr, detachstate);
     pthread_attr_getstack(attr, stackaddr, stacksize);
     printf("The detachstate attribute is %d.\n", *detachstate);
     void **it;
     for (it=&stackaddr[0]; *it!=NULL; it++)
          printf("The stack address is %p.\n", *it);
     printf("The stack size is %zu.\n", *stacksize);
     pthread_attr_destroy(attr);
return err;
}
```

```
void *start_rtn(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(STDOUT_FILENO, *it,
strlen(*it));
        }
        printf("Thread 1 returning...\n");
pthread_exit ((void *) 1);
}
```

The output is:

```
$ ./mystacksize
The detachstate attribute is 2.
Doing stuff in thread 1...
Adriana Wise
Richard Stevens
Evi Nemeth
The stack address is 0xfffffffffff80000.
Thread 1 returning...
The stack size is 524288.
```

The *guardsize* attribute sets the amount of memory, past the end of the stack, that guards the thread against stack overflow. A commonly used default value is the system page size. Obviously, to disable it, *guardsize* can be set to 0. Setting *stackaddr* will also set *guardsize* to 0, as the system assumes that by doing that, the programmer wanted full control over the stack management.

```
#include <pthread.h>

int pthread_attr_getguardsize(
          const pthread_attr_t *restrict attr,
          size_t *restrict guardsize);

int pthread_attr_setguardsize(
          pthread_attr_t *attr, size_t guardsize);
```

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

The previous program was slightly modified to also show the default *guardsize*:

```
#include <pthread.h>
#include <stdlib.h> /* for malloc() */
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for STDIO_FILENO and write() */
#include <string.h> /* for strlen() */

void *start_rtn(void *arg);

int main(void)
{
     int err;
     pthread_t *tid=malloc(sizeof(pthread_attr_t));
     pthread_attr_t *attr=malloc(sizeof(pthread_attr_t));
     void **stackaddr=calloc(sizeof(void), sizeof(void));
     size_t *stacksize=malloc(sizeof(size_t));
     size_t *guardsize=malloc(sizeof(size_t));

     err=pthread_attr_init(attr);
     if (err!=0)
          return err;
     err=pthread_attr_setdetachstate(attr,
PTHREAD_CREATE_DETACHED);
     if (err==0)
          err=pthread_create(tid, attr, start_rtn, NULL);
     pthread_attr_getstack(attr, stackaddr, stacksize);
     pthread_attr_getguardsize(attr, guardsize);
     void **it;
     for (it=&stackaddr[0]; *it!=NULL; it++)
          printf("The stack address is %p.\n", *it);
     printf("The stack size is %zu.\n", *stacksize);
     printf("The guard size is %zu.\n", *guardsize);
     pthread_attr_destroy(attr);
return err;
}

void *start_rtn(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(STDOUT_FILENO, *it,
strlen(*it));
        }
        printf("Thread 1 returning...\n");
pthread_exit ((void *) 1);
}
```

The output shows the default *guardsize* of one page size:

```
$ ./myguardsize
The stack address is 0xfffffffffff80000.
Doing stuff in thread 1...
Adriana Wise
Richard Stevens
Evi Nemeth
The stack size is 524288.
Thread 1 returning...
The guard size is 4096.
```

Mutex Attributes

These are specified by the members of the `pthread_mutexattr_t` structure. So far, when we made a call to `pthread_mutex_init()`, we passed NULL to its *attr* argument. To set non-default mutex attributes, the following functions will initialize and destroy a `pthread_mutexattr_t` structure:

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

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

As always, the init function will initialize the mutex attribute object with the default settings.

Within a process, multiple threads can access the same mutex attribute object. The process-shared attribute (*pshared*) is set to PTHREAD_PROCESS_PRIVATE, in this case. If multiple processes share memory (map the same memory into their separate address spaces), *pshared* is set to PTHREAD_PROCESS_SHARED.

The following two function get and set the value of the ***pshared*** attribute:

```
#include <pthread.h>

int pthread_mutexattr_getpshared(
          const pthread_mutexattr_t *restrict attr,
          int *restrict pshared);

int pthread_mutexattr_setpshared(
          pthread_mutexattr_t *attr, int pshared);
```

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

The following program initializes attribute structures for two threads, and returns the value of the ***pshared*** attribute, which by default is 2 (…PRIVATE):

```
#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;
pthread_mutexattr_t *attr1, *attr2;
int *pshared1, *pshared2;

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");

     attr1=malloc(sizeof(pthread_mutexattr_t));
     pshared1=malloc(sizeof(int));
     err1=pthread_mutexattr_init(attr1);
     printf("err1=%d, %s\n", err1, strerror(err1));
     err1=pthread_mutexattr_getpshared(attr1, pshared1);
     printf("err1=%d, %s\n", err1, strerror(err1));
     printf("pshared=%d\n", *pshared1);
     err1=pthread_mutexattr_destroy(attr1);
     printf("err1=%d, %s\n", err1, strerror(err1));

     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");

     attr2=malloc(sizeof(pthread_mutexattr_t));
     pshared2=malloc(sizeof(int));
     err2=pthread_mutexattr_init(attr2);
     printf("err1=%d, %s\n", err2, strerror(err2));
     err2=pthread_mutexattr_getpshared(attr2, pshared2);
     printf("err2=%d, %s\n", err2, strerror(err2));
     printf("pshared=%d\n", *pshared2);
     err2=pthread_mutexattr_destroy(attr2);
     printf("err1=%d, %s\n", err2, strerror(err2));

     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 is:

```
$ ./mypshared
Doing stuff in thread 1...
err1=0, Undefined error: 0
err1=0, Undefined error: 0
pshared=2
err1=0, Undefined error: 0
Thread 1 returning...
Doing stuff in thread 2...
err1=0, Undefined error: 0
err2=0, Undefined error: 0
pshared=2
err1=0, Undefined error: 0
Thread 2 returning...
Thread 159543296 has exit code 1.
Thread 160079872 has exit code 2.
```

The values for the two constants are #defined in `/usr/include/pthread.h`:

```
#define PTHREAD_PROCESS_SHARED          1
#define PTHREAD_PROCESS_PRIVATE         2
```

If two processes are running, and a thread from the second process is blocked waiting for a thread in the first process to release a lock, but the first process terminates while its thread still holds the lock, the second thread would normally hang. However, if the mutex attribute robust is set, this allows the thread from the second process to acquire the lock from the thread of the terminating (first) process.

The getter and setter of this attribute are:

```
#include <pthread.h>

int pthread_mutexattr_getrobust(
          const pthread_mutexattr_t *restrict attr,
          int *restrict robust);
```

Returns 0 if OK, `errno` if failure.

```
#include <pthread.h>

int pthread_mutexattr_setrobust(
          pthread_mutexattr_t *attr, int robust);
```
Returns 0 if OK, `errno` if failure.

The following program shows an example of two processes, each with one thread. One of the processes (hence its thread) terminates while holding a lock for the other (e.g. child process terminates while holding a lock the parent needs to unlock):

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

#define BUFFSIZE 4096
#define FILESIZE 1474560
#define LISTSIZE 20

pthread_t tid1, tid2;
void *tret1, *tret2;
pthread_mutex_t *lock;
pthread_mutexattr_t *attr;
int err, err1, err2;
int *robust;

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

int main(int argc, char *argv[])
{
     char *pathname="/Users/awise/Stevens/Lecture11/myprog";
     //char **const list1=malloc(LISTSIZE*sizeof(char *));
     char *list1[LISTSIZE];
     list1[0]=pathname;
     list1[1]="Adriana Wise";
     list1[2]="Richard Stevens";
     list1[3]="Evi Nemeth";
     list1[4]=NULL;

     //char **const list2=malloc(LISTSIZE*sizeof(char *));
     char *list2[LISTSIZE];
```

```
     list2[0]=pathname;
     list2[1]="Alicia Beth Moore";
     list2[2]="Tom Petty";
     list2[3]="Sam Smith";
     list2[4]=NULL;

     lock=malloc(sizeof(pthread_mutex_t));
     err=pthread_mutex_init(lock, NULL);
     attr=malloc(sizeof(pthread_mutexattr_t));
     robust=malloc(sizeof(int));
     int pid=fork();
     if (pid==0) /* child process */
     {
          //tid1=pthread_self();
          err=pthread_mutex_init(lock, NULL);
          err1=pthread_create(&tid1, NULL, start_rtn1, NULL);
          err1=pthread_join(tid1, &tret1); /* thread 1 joins the
main thread */
          err1=pthread_mutexattr_getrobust(attr, robust);
          printf("The value of attribute robust is: %d.\n",
*robust);
          printf("Child's default thread has tid1=%d.\n", (int)
tid1);
          execv(pathname, list1);
          while (getppid()!=1) /* while parent's parent is not
init */
               sleep(2);
     }
     else /* parent process */
     {
          //tid2=pthread_self();
          err2=pthread_create(&tid2, NULL, start_rtn2, NULL);
          err2=pthread_join(tid2, &tret2); /* thread 1 joins the
main thread */
          printf("Parent's default thread has tid2=%d.\n", (int)
tid2);
          execv(pathname, list2);
     }
     if (pthread_equal(tid1, tid2)!=0)
          printf("Thread IDs are equal.\n");
     else
          printf("Although... tid1=%d and tid2=%d, these thread
IDs are NOT EQUAL.\n", (int) tid1, (int) tid2);
     err=pthread_mutex_destroy(lock);
return 0;
}
```

```
void *start_rtn1(void *arg)
{
     int err1=pthread_mutex_lock(lock);
     printf("Doing stuff in thread 1...\n");
     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);
     printf("Doing stuff in thread 2...\n");
     printf("Thread 2 returning...\n");
     //err2=pthread_mutex_unlock(lock);
     if (err1!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
pthread_exit ((void *) 2);
}
```

Alas, `pthread_mutexattr_robust()` is not supported on MacOS X, but what we did learn from this exercise is that the thread IDs of the threads created in a parent and a child process are identical!

```
$ ./myrobust
Doing stuff in thread 2...
Thread 2 returning...
Parent's default thread has tid2=4706304.
Doing stuff in thread 1...
Thread 1 returning...
Child's default thread has tid1=4706304.
Alicia Beth Moore
Adriana Wise
Tom Petty
Richard Stevens
Sam Smith
Evi Nemeth
```

Currently, `pthread_mutexattr_robust()` is only supported on Linux and Solaris.

The following function can be called to return a value indicating that the state of the application associated with a given mutex is consistent, before unlocking the mutex:

```
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t *mutex);
```
Returns 0 if OK, `errno` if failure.

If a thread unlocks a mutex without calling `pthread_mutex_consistent()` first, other threads currently blocked waiting on the mutex to be released will see `errno` errors of ENOTRECOVERABLE.

Another mutex attribute is *`type`*. There are 4 types:

- PTHREAD_MUTEX_NORMAL—a mutex type that doesn't do any error checking or deadlock detection
- PTHREAD_MUTEX_ERRORCHECK—a mutex type that does error checking
- PTHREAD_MUTEX_RECURSIVE—a mutex type that allows the same thread to lock it multiple times before unlocking it. It maintains a lock count (for the number of times the same lock was set), to match that with the number of times the lock is released
- PTHREAD_MUTEX_DEFAULT—a mutex providing default characteristics. BSD maps this type to PTHREAD_MUTEX_NORMAL

This attribute can be gotten and set with the following functions:

```
#include <pthread.h>

int pthread_mutexattr_gettype(
          const pthread_mutexattr_t *restrict attr,
          int *restrict type);

int pthread_mutexattr_settype(
          pthread_mutexattr_t *attr, int type);
```
Both return 0 if OK, `errno` if failure.

The following program retrieves types of mutexes:

```
#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;
pthread_mutexattr_t *attr;
int *pshared, *type;

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

int main(void)
{
     int err;
     lock=malloc(sizeof(pthread_mutex_t));
     attr=malloc(sizeof(pthread_mutexattr_t));
     pshared=malloc(sizeof(int));
     type=malloc(sizeof(int));

     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 */
```

```
     err=pthread_mutexattr_init(attr);
     err=pthread_mutexattr_getpshared(attr, pshared);
     printf("%d, %s\n", err, strerror(err));
     printf("pshared=%d\n", *pshared);

     err=pthread_mutexattr_gettype(attr, type);
     printf("%d, %s\n", err, strerror(err));
     printf("type=%d\n", *type);

     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_mutexattr_destroy(attr);
     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));
        }
     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));
        }
     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 shows a 0 type:

```
$ ./mytype
Doing stuff in thread 1...
Thread 1 returning...
Doing stuff in thread 2...
Thread 2 returning...
0, Undefined error: 0
pshared=2
0, Undefined error: 0
type=0
Thread 204144640 has exit code 1.
Thread 204681216 has exit code 2.
```

The types are #defined in `/usr/include/pthread.h`:

```
#define PTHREAD_MUTEX_NORMAL            0
#define PTHREAD_MUTEX_ERRORCHECK        1
#define PTHREAD_MUTEX_RECURSIVE         2
#define PTHREAD_MUTEX_DEFAULT           PTHREAD_MUTEX_NORMAL
```

The following program creates two nested threads, and uses recursive locks within each one of them, to protect them from one another. The type of the lock is set with `pthread_mutexattr_settype()` taking the *`type`* argument passed by value as PTHREAD_MUTEX_RECURSIVE:

```
#include <pthread.h>
#include <stdio.h> /* for printf() */
#include <unistd.h> /* for sleep() */
#include <stdlib.h> /* for malloc() */
#include <string.h> /* for strerror() */

pthread_t tid1, tid2;
void *tret1, *tret2;
pthread_mutex_t *lock;
pthread_mutexattr_t *attr;
int *type;

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

int main(void)
{
     int err;
     lock=malloc(sizeof(pthread_mutex_t));
     attr=malloc(sizeof(pthread_mutexattr_t));
     type=malloc(sizeof(int));

     err=pthread_mutex_init(lock, NULL);
     err=pthread_mutexattr_init(attr);
     err=pthread_mutexattr_settype(attr,
PTHREAD_MUTEX_RECURSIVE);
     err=pthread_mutexattr_gettype(attr, type);
     printf("%d, %s\n", err, strerror(err));
     printf("type=%d\n", *type);

     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(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_mutexattr_destroy(attr);
     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");
     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)
{
     printf("Doing stuff in thread 2...\n");
     int err=pthread_join(tid1, &tret1); /* thread 1 joins
thread 2 */
     int err2=pthread_mutex_lock(lock);
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     printf("err1=%d\n", err);
     printf("tret1=%d\n", (int) tret1);
     err2=pthread_mutex_unlock(lock);
     if (err2!=0)
          printf("Could not create mutex, err=%s.\n",
strerror(err2));
     printf("Thread 2 returning...\n");
pthread_exit ((void *) 2);
}
```

The output to this program shows the mutex type as 2, corresponding to the value #defined in `/usr/include/pthread.h` as PTHREAD_MUTEX_RECURSIVE:

```
$ ./myrecursive2
0, Undefined error: 0
type=2
Doing stuff in thread 2...
Doing stuff in thread 1...
Thread 1 returning...
err1=0
tret1=1
Thread 2 returning...
Thread 74760192 has exit code 1.
Thread 75296768 has exit code 2.
```