Lecture 15. Threads

This chapter analyzes how threads allow control over a process, more precisely, allow performing multiple tasks within the environment of a process. All threads within a process have access to its resources. Therefore, sharing the process' resources among multiple threads raises the problem of consistency, also studied in this chapter (thread synchronization, next lecture).

Introduction to Threads

Threads allow a process to perform multiple tasks, with each thread performing a separate task. With this approach:

- Code becomes simpler. Asynchronous events, such as events triggering signals, can be handled within different threads of the same process;
- Threads, as opposed to multiple processes, make it easier to communicate, since they share the same resources;
- Tasks performed by a multithreaded process can be parallelized, whereas single thread processes would have to perform each task serially;
- Interactive programs can have threads dedicated to user input, separate from the other threads of the program.

Multiple threads do not necessarily mean multiprocessors—they can be implemented on single processor systems. This is because some threads may be able to run on the single processor system, while others are blocked, which still improves overall response time.

A thread within a process is characterized by:

- thread ID—identifies the thread within the process
- set of register values
- stack
- scheduling policy
- signal mask
- `errno` variable
- thread-specific data

All threads within a process share:

- text of the executable
- global and heap memory
- stack
- file descriptors

Thread Identification

Every thread has a **thread ID**. Unlike a process ID, which is unique throughout the system, the thread ID is meaningful only in the context of the given process. The data type representing a process ID is `pid_t`, while the data type representing a thread ID is `pthread_t`, which can be interpreted as an integer across different UNIX implementations. However, they are really structures, and the following function is needed to perform a comparison:

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```
Returns non-zero if equal, 0 otherwise.

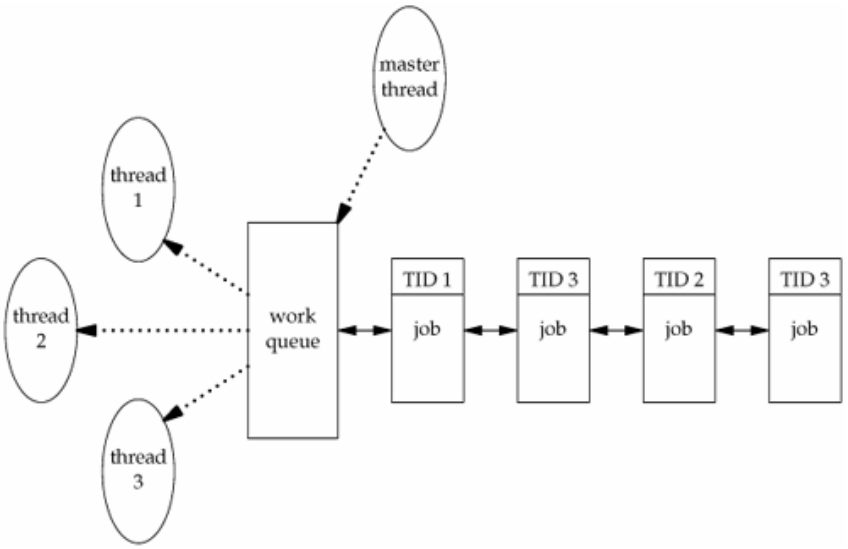
Since each implementation has its own structure for `pthread_t`, there is no portable way to extract values for the members of this structure for a thread.

A thread can return its own thread ID by calling the following function:

```
#include <pthread.h>

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

The thread ID return value from `pthread_self()` can be used as one of the arguments for `pthread_equal()`, to identify data structures tagged with the thread ID of a given thread. For example, a master thread may put work assignments on a queue and use the thread ID to control which worker thread takes which job:



In this diagram, there are 4 jobs in a queue, and 3 threads waiting to perform these jobs, as they move to the head of the queue. The thread master, though, instead of allowing whichever job moves forward to the head of the queue to go to any of the available threads, can assign to each thread only that job (or those jobs) which are marked with its particular thread ID (TID1, TID2, TID3). Thus, the compare between the thread ID of the waiting thread and the thread ID encoded into the structure associated with each job can be performed with `pthread_equal()`.

Thread Creation

By default, each process is created with one thread of control. As the process runs, additional threads can be created with the following function:

```
#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 if failure.

Arguments:

- *tidp*—pointer to a `pthread_t` structure, which is going to be filled in by this function when the new thread is created;
- *attr*—an argument specifying attributes of the thread to be created (will be discussed in detail later). If NULL, the default attribute set will be used;
- *start_rtn*—pointer to the *start_rtn()* function for the new thread;
- *arg*—if there are more than one argument to pass to the *start_rtn()* function, they can be organized in a struct, and this struct passed as the *arg* argument.

The following program creates a new thread and prints the new thread's ID:

```
#include <stdio.h> /* for printf() */
#include <pthread.h> /* for pthread_create(), pthread_self() */
#include <unistd.h> /* for getpid(), sleep() */

pthread_t newThreadID;

void PrintThreadIDs(char *string);
void *StartRoutine(void *arg);
```

```
int main(void)
{
     int err;

     err=pthread_create(&newThreadID, NULL, StartRoutine, NULL);
     if (err!=0)
          printf("Can't create thread, err=%d.\n", err);
     PrintThreadIDs("Main thread: \n");
     sleep(1);
return 0;
}

void PrintThreadIDs(char *string)
{
     pid_t pid;
     pthread_t tid;

     pid=getpid();
     tid=pthread_self();
     printf("%s pid=%lu, tid=%lu, tid(hex)=0x%lx.\n", string,
(unsigned long)pid, (unsigned long)tid, (unsigned long)tid);
}

void *StartRoutine(void *arg)
{
     PrintThreadIDs("New thread: \n");
return (void *)0;
}
```

The output is:

```
$ ./mynewthread
Main thread:
 pid=9724, tid=140735185580816, tid(hex)=0x7fff76be7310.
New thread:
 pid=9724, tid=4527505408, tid(hex)=0x10ddc4000.
```

The printout of the *pid* and *tid* had to cast to unsigned long from `pid_t` and `pthread_t`, respectively. The `sleep()` call in the main function was necessary in order for the main thread to not prematurely terminate, and thus give a chance to the new thread to execute. The `PrintThreadID()` function does not use the value filled in the argument `newThreadID` of the `pthread_create()` function, but calls `pthread_self()` instead. This is because, if the new thread runs before the main

thread returns from calling `pthread_create()`, the new thread will see an uninitialized contents of `newThreadID`.

If we experiment with this, and issue a `printf()` call with the value filled in for the threadID through the reference argument of `pthread_create()`, we get the following output, where the threadID values obtained are indeed different:

```
$ ./mynewthread3
Filled in argument: pid=9831, newThreadID=142663680,
newThreadID=0x880e000
New thread:
 pid=9831, tid=4437630976, tid(hex)=0x10880e000.
Main thread:
 pid=9831, tid=140735185580816, tid(hex)=0x7fff76be7310.
```

Thread Termination

A process will terminate if:

- one of its threads calls `exit()`, `_Exit()`, or `_exit()`;
- a signal sent to a thread will terminate the process as its default disposition.

A thread will terminate if:

1. it returns from its `start_rtn()`, which returns the thread's exit code;
2. it is terminated by another thread within the same process (via signal);
3. it calls `pthread_exit()` on itself.

The function `pthread_exit()` will therefore terminate a thread:

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

Argument:

- *`rval_ptr`*—a void pointer, which specifies the exit status of the thread.

```
#include <pthread.h>

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

Returns 0 if OK, errno on failure.

This function invokes a thread from within the calling thread, blocking the calling thread until the thread invoked through argument thread terminates (by either calling `pthread_exit()` on itself, returning normally from its `start_rtn()`, or being terminated by another thread).

Arguments:

- *`thread`*—threadID of the thread being invoked by the calling thread;
- *`rval_ptr`*—the address in memory containing the return exit status of the invoked thread. If passing NULL for this argument, the exit status is lost.

The following example shows how a thread can join another thread and how its exit status can be recovered with `pthread_join()`:

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

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

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

int main(void)
{
     int err;

     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(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");
     printf("Thread 1 returning...\n");
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 */
     printf("err1=%d\n", err);
     printf("tret1=%d\n", (int) tret1);
     printf("Thread 2 returning...\n");
pthread_exit ((void *) 2);
}
```

The output shows the exit status for both threads, according to the values set in each `start_rtn()` function, as well as the thread IDs for both threads, and the order in which they executed:

```
$ ./mythreadjoin
Doing stuff in thread 1...
Doing stuff in thread 2...
Thread 1 returning...
err1=0
tret1=1
Thread 2 returning...
Thread 35364864 has exit code 1.
Thread 35901440 has exit code 2.
```

A thread can issue a cancel thread request for another thread in the same process:

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Returns 0 if OK, errno on failure.

The default behavior is to set the exit status of the cancelled thread to PTHREAD_CANCELED. The specified behavior can, however, circumvent this default.

Remember that a process can establish a function to be called when the process exits, using the `atexit()` system call. Similarly, a thread can establish one or more functions to be called when the thread exits, called **thread cleanup handlers**. The thread cleanup handlers are recorded in a stack, meaning that they are executed in a LIFO order. The system calls to add/remove thread cleanup handlers to/from the stack are:

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

The cleanup function rtn will be called when the thread:

- calls `pthread_exit()`;
- is cancelled by another thread;
- calls `pthread_cleanup_pop()`.

These two functions must be used in matched pairs within the scope of a single thread.

The following program shows how thread exit handlers can be established within the thread `start_rtn()` function, and how these are automatically called then the threads exit in one of the circumstances listed above. This is a modification of the previous program, in which thread 1 acquires two cleanup handlers. However, because the argument passed to `pthread_cleanup_pop()` is 0, the handlers will not be called.

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

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

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

```
void cleanup1(void *arg);
void cleanup2(void *arg);

int main(void)
{
     int err;

     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(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");
     pthread_cleanup_push(cleanup1, "Thread 1 handler 1");
     pthread_cleanup_push(cleanup2, "Thread 1 handler 2");
     printf("handler 1 and handler 2 pushed for thread 1.\n");
     pthread_cleanup_pop(0);
     pthread_cleanup_pop(0);
     printf("Thread 1 returning...\n");
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 */
     printf("err1=%d\n", err);
     printf("tret1=%d\n", (int) tret1);
```

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

void cleanup1(void *arg)
{
     printf("cleanup1: %s\n", (char *)arg);
}

void cleanup2(void *arg)
{
     printf("cleanup2: %s\n", (char *)arg);
}
```

The output shows the moment when the cleanup handlers were set up, but none of the output from within any of them, since they will not be executed:

```
$ ./mythreadcleanup
Doing stuff in thread 2...
Doing stuff in thread 1...
handler 1 and handler 2 pushed for thread 1.
Thread 1 returning...
err1=0
tret1=1
Thread 2 returning...
Thread 208408576 has exit code 1.
Thread 208945152 has exit code 2.
```

Here is the same program with non-zero arguments to `pthread_cleanup_pop()`, which causes the cleanup handlers to be called:

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

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

void *start_rtn1(void *arg);
void *start_rtn2(void *arg);
void cleanup1(void *arg);
void cleanup2(void *arg);

int main(void)
{
```

```
     int err;

     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(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");
     pthread_cleanup_push(cleanup1, "Thread 1 handler 1");
     pthread_cleanup_push(cleanup2, "Thread 1 handler 2");
     printf("handler 1 and handler 2 pushed for thread 1.\n");
     pthread_cleanup_pop(1);
     pthread_cleanup_pop(1);
     printf("Thread 1 returning...\n");
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 */
     printf("err1=%d\n", err);
     printf("tret1=%d\n", (int) tret1);
     printf("Thread 2 returning...\n");
pthread_exit ((void *) 2);
}

void cleanup1(void *arg)
```

```
{
     printf("cleanup1: %s\n", (char *)arg);
}

void cleanup2(void *arg)
{
     printf("cleanup2: %s\n", (char *)arg);
}
```

The output now shows the actions prescribed in the two exit handlers for thread 1. It also shows the LIFO order in which the cleanup handlers were popped, handler 2 before handler 1:

```
$ ./mythreadcleanup2
Doing stuff in thread 2...
Doing stuff in thread 1...
handler 1 and handler 2 pushed for thread 1.
cleanup2: Thread 1 handler 2
cleanup1: Thread 1 handler 1
Thread 1 returning...
err1=0
tret1=1
Thread 2 returning...
Thread 223334400 has exit code 1.
Thread 223870976 has exit code 2.
```

The following table shows a correspondence between process and thread functions:

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_cancel_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

To make a thread release its storage to the caller thread sooner than by waiting for `pthread_join()` to invoke its cleanup handler and wait for its return, the system call `pthread_detach()` can be called instead. If this function was called, however,

the `pthread_join()` function can no longer return a valid exit status for the thread. Instead, it will return EINVAL.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns 0 if OK, errno on failure.

The following example shows how to use `pthread_detach()` prior to the call to `pthread_join()`, and how this causes `pthread_join()` to return 22, corresponding to `errno=EINVAL`, from `/usr/include/sys/errno.h`:

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

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

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

int main(void)
{
     int err;

     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(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");
     printf("Thread 1 returning...\n");
pthread_exit ((void *) 1);
}

void *start_rtn2(void *arg)
{
     printf("Doing stuff in thread 2...\n");
     pthread_detach(tid1);
     int err=pthread_join(tid1, &tret1); /* thread 1 joins
thread 2 */
     printf("Exit status thread 1 is %d=%s.\n", err,
strerror(err));
     printf("Thread 2 returning...\n");
pthread_exit ((void *) 2);
}
```

The output, which includes a printout of the return exit status of `start_rtn1()`, which is thread 1's handler, shows that the call to `pthread_join()` no longer returns a valid exit status for thread 1, which had been detached:

```
$ ./mythreaddetach
Doing stuff in thread 1...
Doing stuff in thread 2...
Thread 1 returning...
Exit status thread 1 is 22=Invalid argument.
Thread 2 returning...
Thread 58167296 has exit code 0.
Thread 58703872 has exit code 2.
```

The `errno` values and their all caps corresponding names, together with their error messages (returned by `strerror()`) are listed in `<sys/errno.h>`. This is the value corresponding to EINVAL ("e invalid"):

```
#define EINVAL          22              /* Invalid argument */
```