CHAPTER VIII: INTRODUCTION TO THREADS IN UNIX

B. Boufama

University of Windsor

Thread Concepts

A single process does only one thing.
On the other hand, with threads we can do multiple things within s single process.
⇒ each thread does a specific task.
Advantages:

meaning tasks can run parallely

- Code for asynchronous events can be simplified
- Threads share memory and file descriptors
- Using multi-thread can be more efficient than using multi-process in a program.
- Improved response time for interactive programs.

Thread identification

Thread ID is an object of **pthread_t** (a structure)
⇒ cannot be treated as an integer.
A thread ID is local to a process

Linux uses unsigned long int for **pthread_t** whereas,
Mac OS uses a pointer to **pthread**.

⇒ no portable way to print its value.

To compare two threads, use
int pthread_equal(pthread_t t1,pthread_t t2)

The equivalent of **getpid()** is
pthread_t pthread_self(void);
It returns the thread ID of the caller.

Thread creation

```
int pthread_create(pthread_t *tidp,
                   const pthread_attr_t *attr,
                   void *(*start_rtn)(void *),
                   void *arg);
```

Returns 0 if OK, error number on failure.

- The newly created thread ID is stored in **tidp*
- *attr* for customizing thread attribute, NULL for default
- The new thread start running at address *start_rtn* function. This function takes one argument, *arg* (generic pointer).

Example

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

void* myThreadFunc (void* X){
  for(int i=0; i<20; i++)
    printf("new thread is here\n");
  return NULL;
}
int main (int argc, char *argv[]){
 pthread_t threadId;

 // Create a new thread to run myThreadFunc()
 pthread_create(&threadId,NULL, myThreadFunc,NULL);
 for(int i=0; i<20; i++)
  printf("Original thread is here\n");
 sleep(1);
 return(0);
} // compile with -lpthread library link
```

pthread_create will do something internally and add the thread id of the specific function in &threafId

returned value from myThreadFunc will be used in pthread_create

O/P
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
Original thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here
new thread is here

Orginal thread and new thread will get executed simultaneuosly, since it will take time for new thread to be created the old thread will complete printing first and then the new thread content will be printed. This might not be the same always, sometimes new thread and old thread contents might get printed simultaneously too

Thread termination

A single thread can exit without terminating the entire process in three ways:

- By returning from its routine with an exit code.
- Can be canceled by another thread (within the same process)
- By calling pthread_exit(void *rval_ptr)

function pointer

rval_ptr is available to other threads in the process that call

function pointer

*int pthread_join(pthread_t tid, (void *rval_ptr));*

returns 0 for success, error number otherwise.

Note that the caller will block until the specified thread terminates.

Example

```
#include <pthread.h>
#include <stdio.h>
void err_exit(int err, const char *str){
  fprintf(stderr, "%s\n", str);
  exit(err);
}
void* myThreadFunc1 (void* arg){
 printf("Thread1 returning\n");
 return ((void *) 1);
}
void* myThreadFunc2 (void* arg){
 printf("Thread2 exiting\n");
 pthread_exit ((void *) 2);
}
int main(int argc, char *argv[]){
 pthread_t tid1, tid2;
 int errVal;
 void *tRet;
```

1 is casted to void pointer, as this function will return anything.
1 will be casted to void * and returned.

Any return value can be stored

if(1) is true

P_thread_create returns 0 for success and others for failure

```
if((errVal=pthread_create(&tid1,NULL,
    myThreadFunc1,NULL)))
  err_exit(errVal, "Cannot create thread");
if((errVal=pthread_create(&tid2,NULL,
    myThreadFunc2,NULL)))
  err_exit(errVal, "Cannot create thread");
```

Returned value from myThreadFunc1 will be used internally in pthread_create

This join will make main to wait for thread1 to finish its execution.

```
if((errVal=pthread_join(tid1, &tRet)))
  err_exit(errVal, "Cannot join with thread 1");
printf("Thread1 exit code %ld\n", (long)tRet);
```

This join will make main to wait for thread2 to finish its execution.

```
if((errVal=pthread_join(tid2, &tRet)))
  err_exit(errVal, "Cannot join with thread 2");
printf("Thread2 exit code %ld\n", (long)tRet);
exit(0);
}
```

O/P

Thread1 returning
Thread2 exiting
Thread1 exit code 1
Thread2 exit code 2

We are calling pthread_join from main, meaning main will wait for tid1(i.e myThreadFunction1 to finish execution and return a value to pthread_Join and the returned value is also stored in &tRet. pthread_join will return 0 if success and error number if failure, that is what is stored in errVal.

Thread attributes

Attributes can be used for fine tuning threads.
To add attributes, follow the steps:

- Define a variable of type *pthread_attr_t*.
- Start with default values for the variable, using **pthread_attr_init**.
- Modify the attribute variable for the target attributes
- Pass a pointer to this variable when creating the thread.
- If not needed, free the attribute variable using *pthread_attr_destroy()*

Note that typically, only the attribute *detach state* is of interest to us.
By default a thread is created as a joinable thread but can be a detached thread if specified in the attributes.

Joinable threads behave like process as they become like zombies if their return code is not accepted.

A detached thread is automatiocally deleted when it returns. ⇒ Cannot be synchronized with other threads

A thread which has detach as attribute will be automatically deleted after its done with its execution / returned. This cannot be synchronized with other threads.

```
void *Func(void *arg){
 Sleep(10);
 printf("I am the thread working\n");
 pthread_exit ((void *) 0);
}
int main(){
 pthread_attr_t att;
 pthread tid;
 pthread_attr_init(&att);
 pthread_attr_setdetachstate(&att,
                             PTHREAD_CREATE_DETACHED);
 pthread_create(&tip, &att, Func, NULL);
 pthread_attr_destroy(&att);
 printf("No need to join and I cannot anyway\n");
 sleep(1);  return(0);
}
```

If I add a sleep here then the main thread will not wait for this new thread to finish. Main will end the program and the o/p in this case will be, see below

saipavans-MBP:finals yugapriya$ ./a.out
No need to join and I cannot anyway
saipavans-MBP:finals yugapriya$

Creating a struct variable

initializing an object for struct and adding default values in it

This method will set thread to be created as detached in struct att

creating a thread with the attributes set in att(i.e)detached thread.

Destroying the attribute att

Diff between normal thread creation and with att -detached is that: In normal thread creation, the thread will wait for the called process to accept its return value.If thread att is set to detached then the thread will not wait for the called process to accept its return value.

Thread synchronization

Because threads share data, consistency is required
⇒ Synchronization
E.g., If a thread is writing to a variable, another thread must wait until the write is completed before reading it.
If the write is atomic, then there is no issue.
However, a write operation usually takes a few cpu-cycle and is architecture-dependent.
E.g., incrementing a variabe x requires to (1) move value from x to a register, (2) increment the register and (3) move the new value back to x.

⇒ A lock can be used to restrict access to a variable to one thread only.

Mutexes

A *mutex* is a lock that we set (lock) before accessing a shared resource then, release (unlock) after we are done.
When a thread tries to lock a set *mutex*, it will block.

A *mutex* variable is of *pthread_mutex_t* type

A *mutex* variable should be first initialized by

- setting it to the constant PTHREAD_MUTEX_INITIALIZER, when the variable was statically allocated or,
- by calling *pthread_mutex_init()*, when the variable was dynamically allocated.

The following functions can be used for mutex:

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

int pthread_mutex_destroy(pthread_mutex_t *mutex);

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 the above return 0 if OK, error number otherwise

```
struct foo{      // Example using mutex
  int  f_count; // to protect a datastructure
  pthread_mutex_t f_lock;
  int f_id;
  :
}
```

foo_alloc is a function that returns a pointer to struct foo

```
struct foo *foo_alloc(int id){
 struct foo *fp;
 if((fp=malloc(sizeof(struct foo)))){
   fp->f_count=1;
   fp->f_id=id;
   if(pthread_mutex_init(&fp->f_lock, NULL)){
      free(fp);
      return(NULL);
   }
 :
 }
 return(fp);
}
```

https://www.thegeekstuff.com/2012/05/c-mutex-examples/

O/P of 2nd code with lock in the above link

Job 1 started

Job 1 finished

Job 2 started

Job 2 finished

```
void foo_hold(struct foo *fp){
  pthread_mutex_lock(&fp->f_lock);
  fp->f_count++;
  pthread_mutex_unlock(&fp->f_lock);
}
void foo_rele(struct foo *fp){
  pthread_mutex_lock(&fp->f_lock);
  if(--fp->f_count==0){ // last reference
    pthread_mutex_destroy(&fp->f_lock);
    free(fp);
  }else
    pthread_mutex_unlock(&fp->f_lock);
}
```

Threads vs Processes

Both processes and threads can be used When concurency is advantageous.
Which one to use?

- Unlike threads, child processes may run a different executable using *exec()*.
- Unlike a process, a thread might harm other threads because of memory sharing, e.g., with wrong pointer contents.
- Creating processes is more costly than creating threads.
- Threads are prefered for executing very similar tasks in parallel, whereas processes are better for heterogeneous tasks.
- It is easier to share memory amon threads compared with processes that require IPC mechanism.

For more information, read the followings:

- poincare.matf.bg.ac.rs/ ivana//courses/ps/sistemi_knjige/pomocno/apue.pdf (chapters XI and XII)
- www.advancedlinuxprogramming.com/alp-folder/alp-ch04-threads.pdf

Concurrency indicates that more than one thread is making progress, but the threads are not actually running simultaneously. The switching between threads happens quickly enough that the threads might appear to run simultaneously.