COMP 8567

Advanced Systems Programming

Threads

Outline

- Thread Concepts
- Thread Identification
- Thread Creation
- Thread Termination
- Thread Attributes
- Thread Synchronization
- Mutexes
- Threads vs Processes
- Summary

Thread Concepts

- A single process does only one thing.
- On the other hand, with threads we can do multiple things **within** a single process (concurrently)
- Each thread performs a specific task.
- Within a single process we can have **multiple threads** that **run concurrently**.
- Advantages:
  - Code for asynchronous events can be simplified
  - Threads share memory and file descriptors
  - Using multi-threading can be more efficient than using multi-process in a program.
  - Improved response time for interactive programs.

Thread Identification

Thread ID is an object or a member of data type **pthread_t** (a structure)

A thread ID is local to a process (threads in different processes many have the same id)

Linux uses **unsigned long int** for pthread_t

The equivalent of getpid() in threads is **pthread_t pthread_self(void);**

It returns the thread ID of the caller (to be discussed later)

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** is use for customizing the thread attribute, NULL for default
- The new thread starts running at address **start_rtn** function.
  - Pointer to a function which returns a generic pointer and takes a generic pointer as input
- Start_rtn takes one argument, **arg** (generic pointer).

```
//sample.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* func(void* p) {

  printf("From the thread function\n");
  sleep(1);
    pthread_exit(NULL);
  return NULL;
}
main() {
  pthread_t t1; // declare thread
  pthread_create(&t1, NULL, func, NULL);
  sleep(2);
  printf("From the main function\n");
}
```

loop.c //Loop in two threads within a program

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

void* myThreadFunc (){
for(;;)
{
printf("First thread\n");
}
return NULL;
}

void* myThreadFunc1 (){
for(;;)
{
printf("Second thread\n");
}
return NULL;
}
```

```
int main (int argc, char *argv[]){

pthread_t threadId1,threadId2;

pthread_create(&threadId1,NULL, &myThreadFunc,NULL);
pthread_create(&threadId2,NULL, &myThreadFunc1,NULL);

printf("This is the main thread\n");

return(0);
} // compile with -lpthread library link
```

Threads..

- Can the **same function** can be associated with multiple threads? Yes //multiple.c

```
pthread_t t1,t2;
pthread_create(&t1, NULL, func, NULL);
pthread_create(&t2, NULL, func, NULL);
```

- Can a value be passed to a thread function while creating a thread?
  - Yes. Thread functions can accept one parameter (A generic pointer, i.e void *)
- Can the calling thread wait for the called thread to complete its execution?
  - Yes. Using the function pthread_join(threadid, void ** retvalue);

//multiple.c

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

void* func(void* p) {
  printf("Message from thread %d\n",pthread_self());
  sleep(1);
    pthread_exit(NULL);
  return NULL;
}
main() {
  pthread_t t1,t2,t3,t4; // declare thread
  pthread_create(&t1, NULL, func, NULL);
  pthread_create(&t2, NULL, func, NULL);
  pthread_create(&t3, NULL, func, NULL);
  pthread_create(&t4, NULL, func, NULL);
  sleep(2);
  printf("From the main function\n");
}
```

Passing of (int) value to the thread // ipint.c

```
#include <pthread.h>

void* func(void* p)
  printf("From the thread function\n");
  int *num;
  num=p;
  printf("The value of the input paramter to the thread function is %d\n",*num);
  sleep(1);
  return NULL;
}
main() {
  pthread_t t1; // declare thread
  int a=900;
  pthread_create(&t1, NULL, func, &a);
  sleep(2);
  printf("From the main function\n");
}
```

//self.c   Thread id using pthread_self()

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

void* func(void* p) {
  printf("From the thread function, the id of the thread = %d\n", pthread_self()); //get current thread id
  return NULL;
}

main() {
  pthread_t t1,t2; // declare thread
  pthread_create(&t1, NULL, func, NULL);
  sleep(2);
   printf("The value of t1 after pthread_create() = %d\n", t1);
   printf("From the main function, the id of the main thread is = %d\n", pthread_self());

}
```

Thread Termination

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

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

rval_ptr is available(only) to other threads in the process that call:

pthread_join(pthread_t tid, void **rval_ptr)

- returns 0 on success, error number otherwise.

Note that the caller will blocked until the specified thread terminates (when pthread_join() is used) //see subsequent examples

withjoin.c //

- Code available on Brightspace

withoutjoin.c //

- Code available on Brightspace

cancel.c

// Please find the code on blackboard

//joinstring.c

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

void* func(void* p) {
  char * p1="\nReturn message from the thread: Hello\n";
  pthread_exit(p1);
}

main() {
  pthread_t t1; // declare thread
  void *ret;
  pthread_create(&t1, NULL, func, NULL);
  pthread_join(t1,&ret);
  printf("The return value from the thread is \n %s",ret);
}
```

//joinint.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *myThread()
{
  int * iptr;
  iptr=malloc(sizeof (*int));
  *iptr=5;
  pthread_exit(iptr);
}
int main()
{
  pthread_t tid;
  int *result;  //result is a pointer
  pthread_create(&tid, NULL, myThread, NULL);
  pthread_join(tid, (void *) &result);   // & result is the address of result (which is in in turn a pointer)
  printf("%d\n", *result);
  return 0;
}
```

Can a thread (other than the main thread) create another thread?

- tttt.c

Thread Attributes //The second parameter of pthread_create()

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()

```
detachstate
schedpolicy
schedparam
inheritsched
scope
stackaddr
stacksize
stack
guardsize
```

Note that typically, only the attribute *detach state* is of interest to us.

By **default** a thread is created as a joinable thread (i.e pthread_join() can be used to wait for the created thread) but can be converted into a detached thread if specified in the attributes.

detatch.c

```
void *Func(void *arg){
while(1)
{
printf("From the thread\n");
sleep(1);
}
pthread_exit (NULL);
}

int main(){
pthread_attr_t att;   // Define a variable of type pthread_attr_t
pthread_t tid;
pthread_attr_init(&att);   //  Initialize the variable with default value
pthread_attr_setdetachstate(&att,PTHREAD_CREATE_DETACHED);   //modify the attribute variable
pthread_create(&tid,&att, Func, NULL);   // Pass a pointer to this variable while creating the thread
pthread_join(tid,NULL);  //pthread_join will not make the main thread wait for the called thread (tid)
pthread_attr_destroy(&att);
printf("The calling thread does not wait for the called thread despite the pthread_join()\n");
return(0);
}
```

List of Methods Related to attr

int **pthread_attr_init**(pthread_attr_t *attr);

int **pthread_attr_destroy**(pthread_attr_t *attr);

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

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

int **pthread_attr_setstacksize**(pthread_attr_t *attr, size_t stacksize);

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

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

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

```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);

int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);  //  Most Commonly Used

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

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr, int *restrct inheritsched);

int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);

int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr, int *restrict policy);

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);

int pthread_attr_getscope(const pthread_attr_t *restrict attr, int *restrict contentionscope);
```

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-cycles and is architecture-dependent.
- E.g., incrementing a variable x requires (1) move value from x to a register, (2) increment the register and (3) move the new value back to x.
- In the meantime, if another thread reads from x, its value might be inconsistent
- A lock can be used to restrict access to a variable to one thread only

Mutex

- A mutex is a lock that we set (lock) before accessing a shared resource and then, release (unlock) after we are done.
- When a thread tries to lock a set mutex(a mutex which is already locked), 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 is statically allocated or,
- by calling pthread_mutex_init(), when the variable is dynamically allocated.

Functions that can be used for mutex:

- The following functions can be used for mutex:
- int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr); //inititalizes the mutex object with attr, however, if attr is NULL, the mutex is inititlized with default attributes
- int pthread_mutex_destroy(pthread_mutex_t *mutex); //Destroys the mutex object
- int pthread_mutex_lock(pthread_mutex_t *mutex); //Tries to lock and mutex object, and blocks if the mutex object is already locked
- int pthread_mutex_trylock(pthread_mutex_t *mutex);//Tries to lock and does not block if the mutex object is already locked
- int pthread_mutex_unlock(pthread_mutex_t *mutex); //Unlocks the mutex object
- All of the above functions return 0 if OK, error number otherwise

ticket1.c //without mutex

- Code is available on Blackboard

ticket2.c //with mutex

- Code is available on Blackboard

Thread vs. Processes

- Both processes and threads can be used when concurrency 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 computationally expensive than creating threads.
- Threads are preferred for executing very similar tasks in parallel, whereas processes are better for a variety of (heterogeneous) tasks.
- It is easier to share memory among threads compared with processes that require IPC mechanism

Summary

- Thread Concepts
- Thread Identification
- Thread Creation
- Thread Termination
- Thread Attributes
- Thread Synchronization
- Mutexes
- Threads vs Processes

Thank You