POSIX threads

Introduction

- What is POSIX ?

  In order to software standard for different operating systems and their programs, the POSIX(also referred to as the open operating system standard) evolved, consisting of participants from IEEE and ISO. Their goal was to supply standards that would promote application portability across different platforms, to provide a UNIX-like computing environment.i.e. new software written on one machine run on another machine with different hardware.

  In 1988, the first standard POSIX 1003.1 was adopted. The purpose was to provide a C language standard.

  In 1992, IEEE 1003.2 POSIX shell standard and general utility programs were established.

- What is threads ?

  A thread is a sequence of control within a process. So, a process at least has one thread of execution.

  Multiple strands of execution in a single program are called **threads**.

- Difference between child process and thread

  Child process has its own variables and its own PID, can be scheduled independently.

  A new thread of execution gets its own stack(hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with its creator.

- Good or bad about threads

  The performance of an application that mixes input, calculation and output may be improved by running the three tasks as three separate threads. While the input or output thread is waiting for a connection, one of the other threads can continue.

The splitting of large calculation into two parts as different threads will not necessarily run quicker than on a single processor machine, as there are only so many CPU cycles to be had.

Switching between threads is less demanding resources than switching between processes.

In practice, using threads to save the overhead of a process is not a strong argument in LINUX. Debugging a multi-threaded program is much harder than a single threaded one.

Programming interface of threads

1. Create a new thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine),
                   void *arg);
```

pthread creates a new thread. The first argument is a pointer of type of pthread(defined in sys/types.h). The second argument sets thread attributes, normally use NULL as default value. The third argument specifies a function telling the thread where to start executing, and the forth argument is passed as a parameter to this specified function.

This function returns 0 for success, an error if anything goes wrong.

2. Terminate a thread

   ```
   #include <pthread.h>

   void pthread_exit(void *retval);
   ```

   This function terminates the calling thread, and returns a pointer to an object.

3. Wait for a thread

   ```
   #include <pthread.h>

   int pthread_join(pthread_t th,
                    void **thread_return);
   ```

   The first argument tells which thread to wait for, and the second argument is a pointer to a pointer to the return value from the thread. This function returns zero for success and an error code on failure.

A program with multi-threads

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

int numberofthreads = 3;
int count = 2;
char *message = "Ni hao Ma? How are you?";

void *new_thread(void *arg)
{
 /* this is routine of thread,
  define what the thread does.*/
  int i;
  for(i=0; i<count; i++){
    printf("%s\n", message);
    sleep(1);
  }/* thread terminates */
  pthread_exit("Thank you for CPU time\n");
}
```

```
int main()
{
  pthread_t thread;
  int i;
  void *thread_result;

  for(i=0; i<numberofthreads; i++)
  {
   /* create number of threads */
    if(pthread_create(&thread, NULL,
                   new_thread, (void*)message))
    {  printf("error creating a new thread\n");
       exit(1);
     }
   /* wait for this thread to terminate */
     pthread_join(thread, &thread_result);
   /* printout the terminate information */
     printf("%s\n", (char*)thread_result);
  }
  exit(EXIT_SUCCESS);
}
```

Compile this program with command line:

```
gcc -D_REENTRANT thread.c -o thread -lpthread
```

Synchronization with semaphores

1. what is semaphores?

   This word probably comes from the Railway transportation. No two trains on the same section of the same track are allowed.

   Semaphores are used to protect a piece of code so that only one thread of execution can run it at any one time.

   General semaphore is used to synchronize the processes; but here we use **binary semaphores** to implement synchronization of the threads.

2. Programming interface of binary semaphore

   (a) Initialize a semaphore

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
                  unsigned int value);
```

This function initializes a semaphore object pointed to by sem. Parameters:

- 1: *sem: where the semaphore object stored;
- 2: pshared: how many processes are sharing this semaphore, always be 0;
- 3: value: a initial value as you like, normally 0.

(b) Control the value of semaphore

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

Both functions track the thread executing. The first decreases the value of semaphore by 1, but always waits till the semaphore has a non-zero value;

While the second increases the value of semaphore by 1.

(c) Tidy up the semaphore and resources

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

Note: if you try to destroy a semaphore which some threads are waiting for, you will get error.

All functions above return 0 on success.

3. A program with main thread reading and another thread counting

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

#define WORKSIZE 1024
```

```
char workarea[WORKSIZE];
sem_t bin_sem;

void *thread_function(void *arg)
{
/*the thread will execute when semaphore
   value>0, and decrease the value by 1 */
  sem_wait(&bin_sem);
  while(strncmp("end", workarea, 3) !=0){
/*the thread counting the input stringlength*/
    printf("you input %d characters\n",
                    strlen(workarea)-1);
/*wait for the semaphore value >0 till the
  main thread increase it */
    sem_wait(&bin_sem);
   }
  pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
  int res;
  pthread_t thread;
  void *thread_result;
```

```
 /* initialize a semaphore */
 res= sem_init(&bin_sem,0,0);
 if(res != 0){
   perror("Semaphore initialization fail\n");
   exit(EXIT_FAILURE);
 }
 res=pthread_create(&thread,NULL,
                     thread_function, NULL);
 if(res!=0){
   perror("Thread creation fail\n");
   exit(EXIT_FAILURE);
 }
/*main thread read from the user's input */
 printf("Enter text,enter'end'to quit\n");
 while(strncmp("end", workarea, 3) !=0){
   fgets(workarea, WORKSIZE, stdin);
/*main thread increases the semaphore value,
   leave for another thread to work on */
   sem_post(&bin_sem);
 }
/*threads joint here */
 res=pthread_join(thread,&thread_result);
```

```
  if(res!=0){
    perror("thread join fail\n");
    exit(EXIT_FAILURE);
  }
 /*tidy up the address space */
  sem_destroy(&bin_sem);
  exit(EXIT_SUCCESS);
}
```

Synchronization with mutexes

1. what is mutex?

   Mutex: Mutual Exclusion Locks

   It allows the programmer to lock an object, so that only one thread can access it at any one time.

   To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.

2. Programming interface of mutex

   (a) Initialize a semaphore

   ```
   #include <pthread.h>
   int pthread_mutex_init(pthread_mutex_t
                                        *mutex,
                    const pthread_mutexattr_t
                                     *mutexattr);
   ```

This function initializes an object that is pointed to by mutex. Parameters:

- 1: *mutex: where the locked object stored;
- 2: *mutexattr : mutex attribute, default value “fast”, but normally, given as “NULL”.

(b) Lock and unlock the mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t
                                *mutex);
int pthread_mutex_unlock(pthread_mutex_t
                                  *mutex);
```

Both functions take the *mutex as a parameter and lock or unlock the object to which it points to.

(c) Throw away the mutex

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t
                                   *mutex);
```

Note: if you try to destroy a mutex which is locked, you will get error.

All functions above return 0 on success, and an error code on failure.

3. A program with main thread reading and another thread counting

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>
#define WORKSIZE 1024

/*declaring the workarea with size of 1024
  for user's input */
char workarea[WORKSIZE];
/*declare a mutex*/
pthread_mutex_t mutex;
/*declare a variable to inform quit*/
int time_to_exit =0;
```

```
void *thread_function(void *arg)
{
/*this is the counting thread to count the
    string length of the terminal input */

/*lock the workarea, start counting */
 pthread_mutex_lock(&mutex);
  while(strncmp("end", workarea, 3) !=0){
    printf("you input %d characters\n",
strlen(workarea)-1);
/*finished counting, set the workarea empty*/
    workarea[0]= '\0';
/*unlock the workarea */
    pthread_mutex_unlock(&mutex);
    sleep(1); /* rest */

/*relock the workarea */
    pthread_mutex_lock(&mutex);
/*check the workarea is empty or not*/
    while(workarea[0]=='\0'){
/*if it is empty, unlock the space,
leave for another thread to read in*/
      pthread_mutex_unlock(&mutex);
      sleep(1); /* rest */
/* relock the workarea*/
      pthread_mutex_lock(&mutex);
    }
  }/* otherwise, if‘‘end’’ is entered by user,
      set the time_to_exit to 1 */
  time_to_exit=1;
/* set the workarea empty*/
  workarea[0]='\0';
/* unlock the workarea*/
  pthread_mutex_unlock(&mutex);
/* counting thread exit */
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int res;
  pthread_t thread;
  void *thread_result;

/*initialize the mutex */
  res= pthread_mutex_init(&mutex,NULL);
  if(res != 0){
    perror("Mutex initialization fail\n");
    exit(EXIT_FAILURE);
  }

/*create a thread and execute the counting
                 thread as define above */
  res=pthread_create(&thread, NULL,
                 thread_function, NULL);
  if(res!=0){
    perror("Thread creation fail\n");
    exit(EXIT_FAILURE);
  }

/*lock the workarea, start to read in
                       user's input*/
  pthread_mutex_lock(&mutex);
  printf("Enter some text, enter 'end'
                        to finish\n");

/*continue read in, unless the time_to_exit
        is set to 1 by the counting thread*/
  while(!time_to_exit){
```

```
/*read in from terminal */
    fgets(workarea, WORKSIZE, stdin);
/*unlock the workarea */
    pthread_mutex_unlock(&mutex);
/*if program is not terminated */
    while(1){ /* relock the workarea*/
      pthread_mutex_lock(&mutex);
/*check if the workarea has been set empty
  by another thread */
      if(workarea[0]!='\0'){
/*if the workarea is not yet be set empty,
unclock the place. leave for the counting
thread to count and reset workarea empty.*/
     pthread_mutex_unlock(&mutex);
     sleep(1); /* rest */
     }/*otherwise, exit*/
      else{
           break;
     }
    }/*while*/
  }/* while time_to_exit == 1,
    * unlock the workarea*/
  pthread_mutex_unlock(&mutex);
```

```
/*join the threads */
  res=pthread_join(thread,&thread_result);
  if(res!=0){
    perror("thread join fail\n");
    exit(EXIT_FAILURE);
  }/*throw the mutex, tidy up */
  pthread_mutex_destroy(&mutex);
/*terminate the program */
  exit(EXIT_SUCCESS);
}
```

THE END