دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

آزمایشگاه سیستم عامل

جلسه دهم

برنامه‌نویسی با Thread

پاییز ۱۴۰۲

POSIX threads

با مفهوم thread و مزایای برنامه‌نویسی multithread در درس سیستم عامل آشنا شدید. POSIX، کتابخانه pthread را برای برنامه‌نویسی چندنخی ارائه کرده است. در این کتابخانه علاوه بر ساخت thread، امکانات زیادی شامل تنظیم زمان‌بندهای مختلف از طریق pthread_attr و همزمانی برنامه چندنخی (synchronization) وجود دارد.

مدیریت Thread

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

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

با فراخوانی این تابع، روتینی که در آرگومان سوم مشخص شده‌است در یک نخ جدید، شروع به اجرا می‌کند (نخ جدید را نخ فرزند و نخی که pthread_create در آن فراخوانی شده نخ والد می‌گوییم). به این روتین اصطلاحا runner می‌گویند و الگوی آن در ادامه، مشخص شده است. آرگومان اول، کنترل‌کننده نخ است که پس از ساخته‌شدن نخ، در صورت موفقیت مقدار مخالف NULL خواهد داشت. آرگومان دوم ویژگی‌هایی را برای نخ جدید، تعریف می‌کند. در صورتی که بخواهیم الگوریتم زمان‌بندی یا الویت زمان‌بندی یا پارامترهای دیگر مربوط به نخ را تغییر دهیم، موارد مربوطه را باید در رکوردی از نوع pthread_attr_t تنظیم کرده (با استفاده از توابع مربوطه) و سپس در آرگومان دوم استفاده کنیم. اما اگر بخواهیم نخ با تنظیمات پیش‌فرض سیستم ساخته شود، مقدار آرگومان دوم را می‌توان NULL داد و یا از تابع pthread_attr_init برای مقداردهی رکورد آن استفاده کرد. آرگومان چهارم این تابع هم پارامترهای ورودی runner را مشخص می‌کند که باید حتماً همه پارامترها در قالب void* به تابع ارسال شود.

```
void * (void * arg)
{
        //execution routine
        pthread_exit(void * return_value);
}
```

تابع بالا نمونه‌ای از یک روتین runner است که آرگومان ورودی آن void* است. بدین ترتیب هر نوع آرگومان ورودی که برای این روتین نیاز باشد باید در قالب void* به آن ارسال شود و سپس در بدنه runner، به نوع دلخواه cast شود. اگر تعداد آرگومان مورد نیاز بیش از یک باشد، موارد مربوطه به صورت یک رکورد یا structure تعریف شده، cast به void* می‌شود و در آرگومان چهارم pthread_create ارسال می‌شود سپس در بدنه روتین runner، دوباره به نوع رکورد موردنظر cast می‌شود. تابع pthread_exit هم جهت خاتمه دادن به اجرای نخ در بدنه runner استفاده می‌شود و قابلیت ارسال یک مقدار خروجی را دارد. این مقدار خروجی توسط نخی که pthread_join را برای این نخ فراخوانی کرده دریافت می‌شود.

```
pthread_join(pthread_t thread, void **return_value);
```

با استفاده از این فراخوانی، می‌توان منتظر اتمام نخی با کنترل کننده thread شد. همچنین مقدار خروجی نخ موردنظر در آرگومان دوم دریافت می‌شود (این مقدار همان‌طور که در بخش قبل بیان شد، باید توسط pthread_exit در تابع runner ارسال شود.

```
gcc   code.c  -o  appName -lpthread
```

جهت کامپایل، برنامه‌ای که از توابع هدر pthread استفاده کرده است لازم است کتابخانه pthread در کامپایل اضافه شود.

Threads: Creating , Executing and Joining

```
/*
- this program creates 4 threads
- execution routine for threads is "routine1", we pass thread index (i) as
  execution routine argument
- each thread executes the routine in an arbitrary order, in this condition we have
  no control on order of execution
- at pthread_join(), master thread waits for worker threads to complete their
execution, then
  receives their “exit value” that is a random number generated in the thread’s
routine
*/

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

```
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>

#define THREADS 4
#define TIMEOUT 10

void *routine1(void * x)
{
   int *t = (int*)malloc(sizeof(int));
   *t = rand()%TIMEOUT;
   //int t = rand()%TIMEOUT; //replace two above lines with this line
   printf("threadIdx = %d, execution time = %d\n",*(int*)x, *t);
   for(int i=0; i<*t; i++)
      printf("threadIdx = %d; run = %d\n", *(int*)x, i);
   pthread_exit((void*)t);
}

int main ()
{
  pthread_t threads[THREADS];
  int thread_id[THREADS];
  for ( int i=0;i<THREADS;i++){
    thread_id[i] = i;
    pthread_create(&threads[i], NULL, routine1, (void *)&thread_id[i] );
    //replace two above lines with the below line
    //pthread_create(&threads[i], NULL, routine1, (void *)&i );
  }
  int *retval = (int*)malloc(sizeof(int));
  for (int i=0; i<THREADS; i++)
  {
    pthread_join(threads[i],(void**)&retval);
    printf("threadIdx %d finished, return_value = %d \n",i,*retval);
  }
  return 0;
}
```

به نحوه ارسال آرگومان‌های روتین thread و همچنین برگرداندن خروجی نخ دقت کنید. توجه داشته باشید که اگر آدرس‌ها در آرگومان‌های ورودی یا خروجی نخ‌های متفاوت مشترک باشند، ممکن است

در اثر همزمانی اجراها یا خارج‌شدن از scope، مقادیر مربوطه صحیح ارسال نشوند. برای بررسی این موضوع کد را با توجه به کامنتهای بین کدها تغییر داده و دوباره اجرا کنید.

مدیریت semaphore

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

پس از تعریف سمافور از نوع sem_t، آن را با sem_init مقداردهی اولیه می‌کنند که مقدار موردنظر توسط آرگومان سوم این تابع مشخص می‌شود. اگر آرگومان دوم این تابع صفر باشد، نشان می‌دهد که سمافور بین تردهای یک پروسس استفاده خواهد شد و در صورتی که مقدار آن مثبت باشد، سمافور جهت کنترل همزمانی بین پروسس‌های مختلف قابل استفاده است. در هرحال سمافور باید جایی تعریف شده باشد که قابل دسترس از همه تردها یا همه پروسس‌هایی که به آن نیاز دارند باشد. در برنامه‌نویسی multithread کافیست این سمافور به صورت global در پروسس والد تعریف شود. اما در برنامه‌نویسی multiprocess، باید به صورت shared_memory تعریف شود.

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

این دو تابع، جهت اخذ و آزادکردن سمافور قبل و بعد از ناحیه بحرانی استفاده می‌شوند.

```
int sem_getvalue(sem_t *sem, int *valp);
int sem_destroy(sem_t *sem);
```

در POSIX، این امکان وجود دارد که مقدار سمافور را با استفاده از تابع sem_getvalue بازیابی کرد. مقدار بازگشتی در valp در صورتی که سمافور آزاد نباشد، صفر و در غیر این صورت، تعداد نمونه‌های آزاد این سمافور را دربرخواهد داشت.

sem_destroy سمافوری که قبلاً init شده را از بین می‌برد. استفاده از سمافور destroyشده منجر به رفتار نامشخصی می‌شود.

Accessing variable “total”, avoiding multiple writes using semaphores

```
/*
- program creates 4 threads, assigns “routine1” as execution routine for each thread
- defines semaphore “sem1” in global space to be accessible by all threads
- each thread before entering its critical section, evaluates the value of “sem1”
```

```
- remark: sem_wait decrements semaphore /sem_post increments semaphore
*/
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define THREADS 4

sem_t sem1;
int total=0;
void *routine1(void * id )
{
	int idx=(int)id;
	sem_wait(&sem1);
	//beginning of critical section
	total+=1;
	printf("thread=%d and total=%d \n",idx,total);
	sleep(1);
	//end of critical section
	sem_post(&sem1);
	pthread_exit((void *)idx);
}

int main ()
{
	sem_init(&sem1,0,1);
	pthread_t threads[THREADS];

	for ( int i=0;i<THREADS;i++)
		pthread_create(&threads[i],NULL,routine1,(void *)i);


	for (int i=0; i<THREADS; i++)
		pthread_join(threads[i],NULL);

	return 0;
}
```

An Example of Busy-waiting

```
/*
- this program creates 4 threads and assigns “routine1” as execution routine for each thread
- threads will be synchronized by checking the value of variable “total”
- this method is called “busy-waiting”
*/
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define THREADS 4
#define SIZE 16

sem_t sem1;
int step=0;
int total=0;

void *func1(void * id )
{
   int idx=(int)id;
   //busy wait on step value
   while (step < SIZE)
   {
      while ( step%THREADS != idx );
      //beginning of critical section
      total+=1;
      printf("thread=%d and total=%d \n",idx,total);
      step++;
      sleep(1);
      //end of critical section
   }
   pthread_exit((void *)idx);

}
```

```
int main ()
{
   pthread_t threads[THREADS];

   for ( int i=0;i<THREADS;i++)
      pthread_create(&threads[i],NULL,func1,(void *)i);

   for (int i=0; i<THREADS; i++)
      pthread_join(threads[i],NULL);

   return 0;
}
```

این برنامه از سمافور استفاده نمی‌کند و با استفاده از busy waiting عملکرد سمافور را شبیه‌سازی می‌کند. هدف نهایی در این مثال، کنترل همزمانی روی مقدار total است، به صورتی که در هر لحظه فقط به یکی از نخ‌ها اجازه آپدیت این مقدار داده می‌شود. بدین منظور از متغیر سراسری**step** استفاده شده که نوبت هر نخ را مشخص می‌کند.