باسمه تعالی

دانشگاه صنعتی اصفهان

سیستم های عامل-پروژه دوم

موعد تحویل: جمعه ۱۲ دی ۹۹

شرح پروژه

در این پروژه شما با نحوه عملکرد برنامه های چند نخی [1] آشنا می‌شوید و یک پیاده سازی بدون استفاده از توابع POSIX انجام می‌دهید. ممکن است با وجود کتابخانه های آماده موجود، انجام این پروژه بنظر غیرکاربردی و خسته کننده بیاید (اختراع دوباره چرخ!). اما باید این نکته را در نظر داشته باشیم که انجام این پروژه نه تنها کمک می کند با چالش های پیاده سازی یک برنامه موازی بیشتر آشنا شویم بلکه به ما این دید را می‌دهد که چگونه موازی سازی در عمل صورت می گیرد و در آینده زمانی که از توابع آماده استفاده کنیم، می دانیم چه اتفاقاتی در پشت پرده در حال رخ دادن است؛ ضمن این که ممکن است در بعضی مواقع با توجه به محدودیت های سخت‌افزاری و نرم‌افزاری خاص مجبور به پیاده سازی این توابع باشیم.

در این پروژه قصد داریم تا یک کتابخانه برای مدیریت نخ ها پیاده سازی کنیم که آن را کتابخانه green thread می‌نامیم. علت انتخاب واژه green این است که مدیریت نخ ها و زمانبندی آن ها توسط سیستم عامل انجام **نمی‌شود**، بلکه همه تصمیم‌گیری ها و مدیریت تخصیص منابع به نخ ها توسط سیاست گذاری خودمان انجام می‌شود و ما به عنوان توسعه دهندگان این کتابخانه می‌توانیم تعیین کنیم که این سیاست ها چه باشند و مشخص کنیم از چه الگوریتمی برای زمانبندی استفاده شود، از چه ساختمان داده ای برای نگه داری اطلاعات نخ ها استفاده شود و...

۱ بررسی کتابخانه pthread

با توجه به این‌که سعی داریم کتابخانه مشابه pthread توسعه دهیم، بهتر است ابتدا نگاهی به توابع مهم این کتابخانه داشته باشیم:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    *(*start_routine) (void *), void *arg); //used to create a new thread

int pthread_yield(void);  // when programmer yileds its own thread
//to wait for a specific thread we use:
int pthread_join(pthread_t thread,void **retval);
```

این توابع محض یادآوری آورده شده اند برای اطلاعات بیشتر می‌توانید به man page رجوع کنید.

[1] multi thread

`man pthread_create` `man pthread_join` `man pthread_yield` `...`

هدف نهایی ما پیاده سازی توابعی شبیه توابع بالا است. باتوجه به این‌که نمی‌توانیم از کتابخانه pthread استفاده کنیم نیاز داریم به طریقی تعویض متن [۲] بین نخ ها را انجام دهیم.

۲ آشنایی با کتابخانه ucontext

این کتابخانه به ما کمک می‌کند تا بطور مستقیم با context ها در ارتباط باشیم و به سادگی بین آن ها تعویض متن انجام دهیم. ممکن است بصورت پیش فرض هدر فایل مربوط به توابع این کتابخانه نصب نباشد برای نصب در توزیع های بر مبنای Debian (مانند Ubuntu) می‌توانید از دستور زیر استفاده کنید:

```
sudo apt-get install linux-headers-$(uname -r)
```

سعی کنید مستقیم این کد را در ترمینال خود وارد نکنید و کمی به آن فکر کنید. آیا نکته جالبی در دستور بالا مشاهده می‌کنید؟ ☺

در انجام این پروژه احتمالا به توابع زیر نیاز پیدا خواهید کرد:

`getcontext()` `setcontext()` `swapcontext()` `makecontext()`

نحوه تعریف متغییر های struct مورد استفاده در این کتابخانه بصورت زیر است:

```
typedef struct ucontext_t {
struct ucontext_t *uc_link; //context to be resumed after this context
sigset_t          uc_sigmask;
stack_t           uc_stack; // stack of this context
mcontext_t        uc_mcontext;
...
} ucontext_t;

typedef struct stack_t {
void     *ss_sp;  //Stack base or pointer.
size_t    ss_size;  //Stack size.
int       ss_flags;
...
} stack_t;
```

نحوه عملکرد توابع بصورت زیر است:

- **`int getcontext(ucontext_t *ucp);`**

  در این تابع متغییر از نوع `ucontext_t` مقدار دهی اولیه می‌شود و اطلاعات مربوط به context در حال اجرا در آن ذخیره می‌شود.

- **`int setcontext(const ucontext_t *ucp);`**

  در این تابع متغییر از نوع `ucontext_t` به عنوان ورودی دریافت می‌شود و به اجرا در می‌آید. (اطلاعات آن بر روی context در حال اجرا overwrite می‌شود!)

- **`int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);`**

  در این تابع اطلاعات context موجود در آرگومان اول ذخیره می‌شود و اطلاعات مورد نیاز برای اجرای context

[۲] contex switch

جدید از آرگومان دوم خوانده می‌شود.

- `void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);`
  در این تابع متغییر ucp که به آن اشاره شده است و مقادیر `ucp->uc_stack` و `ucp->uc_link` و تابع هدف بروز می‌شوند. از این به بعد زمانی که این context توسط توابع setcontext() و یا swapcontext() به اجرا در می‌آید، این تابع اجرا می‌شود. توجه شود که آرگومان سوم تعداد آرگومان‌های ورودی های تابع است و از آرگومان چهارم به بعد این ورودی ها به‌ترتیب به تابع هدف پاس داده می‌شوند.

با کمک man page می‌توانید اطلاعات بیشتری درمورد توابع بالا کسب کنید.

۳ دستگرمی با کتابخانه ucontex (۴۰۰ نمره)

اکنون که با این کتابخانه آشنا شده‌اید و در مورد آن مطالعه کردید، با انجام دو تمرین ساده این آشنایی را تکمیل می کنیم تا تسلط شما بر کتابخانه بیشتر شود و روند پیاده سازی تسهیل شود.

۱.۳ تمرین اول (۱۵۰ نمره)

برنامه‌ی زیر را در نظر بگیرید:

```
#include <stdlib.h>
#include <stdio.h>
#include <ucontext.h>
#include <unistd.h>
int main ( ) {
  //int done = 0 ;
  ucontext_t one ;
  ucontext_t two ;
  getcontext (&one ) ;
  printf ( "I am running :) \n" ) ;
  sleep(1);
//  if(!done) {
//    done = 1 ;
    swapcontext (&two , &one ) ;
//  }
  return 0 ;
}
```

الف) (۵۰ نمره) این برنامه را کمپایل و اجرا کنید. چه خروجی دریافت می کنید؟چرا؟

ب) (۵۰ نمره) در صورتی که بخواهیم این پیام تنها دوبار چاپ شود چه پیشنهادی دارید؟ اگر خط های comment شده را به کد اضافه کنیم چه خروجی دریافت می کنید؟ در مورد علت آن توضیح دهید.

ج) (۵۰ نمره) اگر در تعریف متغیر done بجای int از register int استفاده کنیم. خروجی چه تغییری می کند؟ فکر می کنید چرا؟

راهنمایی: به این فکر کنید که چه اطلاعاتی در هر context ذخیره می‌شود.

۲.۳ تمرین دوم (۲۵۰ نمره)

در این تمرین می‌خواهیم برنامه ای بنویسیم که عملکرد دو نخ همزمان را شبیه سازی کند. تابعی که قصد داریم در این دو نخ اجرا شود تابع زیر است:

```
void do_job(int p, int  max_num)
{
  for(int i=-max_num;i<0;i++)
      printf("#%d -> %d\n",p,i);
  yield();
  for(int i=1;i<=max_num;i++)
      printf("#%d -> %d\n",p,i);
}
```

از آن جایی که می‌خواهیم در هر context یک تابع را صدا بزنیم پس باید به هر context یک stack جداگانه اختصاص دهیم.(ابتدا به اندازه مورد نیاز حافظه در نظر بگیریم (4KB) سپس آن را به context موردنظر نسبت می‌دهیم. کد را بصورت زیر تکمیل می کنیم:

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

static int running_task_id;
static ucontext_t mContext[2];
static ucontext_t mContext_main;

void yield()
{
    printf("yield from #%d to #%d\n",running_task_id,1-running_task_id);
     if(running_task_id==0)
     {
         running_task_id=1;
         swapcontext(&mContext[0],&mContext[1]);
     }else
     {
         running_task_id=0;
         swapcontext(&mContext[1],&mContext[0]);
     }
}
void do_job(int p, int  max_num)
{
    for(int i=-max_num;i<0;i++)
        printf("#%d -> %d\n",p,i);
    yield();
    for(int i=1;i<=max_num;i++)
        printf("#%d -> %d\n",p,i);
}
int main(int argc,char * argv[])
```

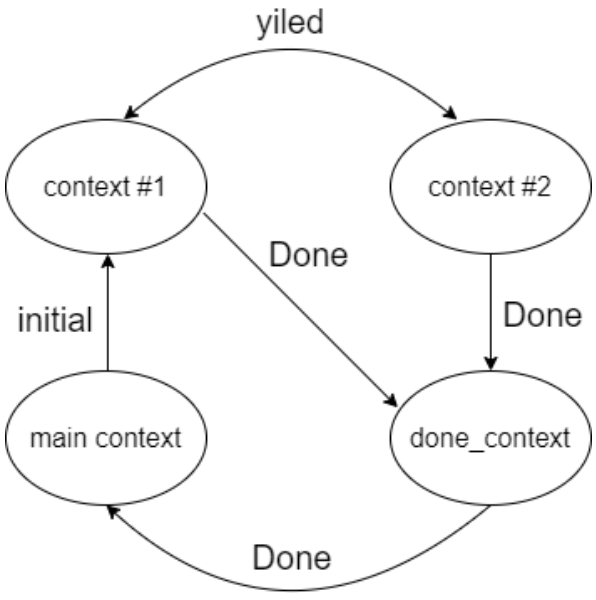
```
{
    char stack1[4*1024];
    char stack2[4*1024];

    getcontext(&mContext[0]);
    mContext[0].uc_stack.ss_sp=stack1;
    mContext[0].uc_stack.ss_size=sizeof(stack1);
    makecontext(&mContext[0],(void(*) (void))do_job,2,1, 3);
//-------------------------------------------------
    getcontext(&mContext[1]);
    mContext[1].uc_stack.ss_sp=stack2;
    mContext[1].uc_stack.ss_size=sizeof(stack2);
    makecontext(&mContext[1],(void (*) (void))do_job,2,2, 5);

    running_task_id=0;
    swapcontext(&mContext_main,mContext);
    printf("returned to main");
    return 0;
}
```

الف) (۱۰۰ نمره) کد بالا را کمپایل و اجرا کنید. خروجی آن را تحلیل کنید و در مورد آن توضیح دهید. آیا برنامه به درستی اجرا شده است؟ چرا؟

ب) (۱۵۰ نمره) برای حل مشکل این برنامه راه های مختلفی وجود دارد. در این قسمت سعی می کنیم یکی از ساده ترین آن ها را پیاده کنیم. یک context جدید به نام done تعریف می کنیم تا زمانی که اجرای هر context تمام شد به این context برویم. برای این که بهتر متوجه تغییر حالت بین context های مختلف شویم می توانیم آن را با یک automata نمایش دهیم. در شکل ۱ این دیاگرام رسم شده است.



شکل ۱: دیاگرام نحوه تعویض متن

با توجه به این که می‌خواهیم بر روی تمام شدن هر وظیفه بصورت جداگانه کنترل داشته باشیم باید برای هر کدام از context ها یک flag جداگانه در نظر بگیریم تا بدانیم آیا اجرای آن context تمام شده است یا خیر؟! کد اصلاح شده را می توانید

در فایل ex2_fiexed.c مشاهده کنید. کد را کامپایل و اجرا کنید، خروجی آن را در گزارش خود بیاورید و در مورد کارکرد آن توضیح دهید.

٤ فاز اول پیاده سازی کتابخانه(٦٠٠ نمره)

برای ساده‌تر کردن توسعه این کتابخانه، پیاده سازی آن را به دو قسمت تقسیم می‌کنیم. در قسمت اول کتابخانه green thread را بصورت ساده پیاده سازی می‌کنیم که در آن تعویض متن بین نخ ها تنها درصورتی اتفاق می‌افتد که خود برنامه نویس اجازه آن را بدهد. (تابع yield و یا wait را صدا بزند) بر خلاف تمرین دوم که در آن تنها ۲ نخ داشتیم در اینجا ممکن است چندین نخ داشته باشیم پس لازم است یک الگوریتم زمانبندی مناسب ارائه کنیم. برای سادگی کار، الگوریتم زمانبدی را FIFO در نظر بگیرید. در نهایت توابعی که باید پیاده کنیم توابع زیر هستند:

```
int green_create(green_t * thread,void *(* fun) (void*),void * arg);
int green_yield();
int green_join(green_t * thread, void ** val);
void green_thread();
```

توجه شود که تابع green_thread یک تابع داخلی در کتابخانه ما است و برنامه نویس آن را صدا نمی‌زند. اکنون عملکرد توابع فوق را مورد بررسی قرار می‌دهیم:

- green_thread: در این تابع fun و arg به ترتیب، تابع و تنها آرگومان آن است که به نخ نسبت داده می‌شود. نخ ایجاد شده در thread قرار می‌گیرد و مقدار ۰ بر می‌گردد.
- green_yield: در این تابع برنامه نویس اعلام می‌کند که نیازی به CPU ندارد و می‌توانیم تعویض متن را انجام دهیم.
- green_join: در این تابع منتظر می‌مانیم تا پروسس اعلام شده خاتمه یابد و مقدار آن را در متغییر val بر می‌گردانیم.
- green_thread: در این تابع ابتدا تابع هدف را صدا می‌کنیم. پس از اجرای کامل چک می‌کنیم اگر نخ دیگری منتظر این نخ بوده است، آن را وارد صف ready می‌کنیم. نتیجه خروجی و وضعیت zombie بودن نخ را در ساختمان داده مربوط به نخ ذخیره می‌کنیم.

هدر فایل پیشنهادی (green.h)

```
#include <ucontext.h>
#ifndef GREEN_H
#define GREEN_H
typedef struct green_t {
  ucontext_t * context ;
  void *(*fun)(void *);
  void * arg;
  struct green_t * next;
  struct green_t * join;
  void * retval;
  int zombie;
} green_t;
```

```
int green_create(green_t * thread , void *(*fun)(void *),void * arg);
int green_yield();
int green_join(green_t * thread, void ** val);
#endif
```

در کنار این فایل pdf پوشه ای به نام green وجود دارد که در آن فایل های کمکی قرار گرفته اند. این فایل ها می‌توانند روند پیاده سازی را برای شما ساده‌تر کنند، زمانی که پیاده سازی خود را تکمیل کردید کافی است از دستور make استفاده کنید تا برنامه ساخته شود پس از اجرای برنامه باید خروجی مشابه شکل ۲ دریافت کنید. (با دستور make برنامه `posix_test` نیز ساخته می‌شود که بتوانید عملکرد برنامه را در هر دو کتابخانه ببینید و مقایسه کنید.)

```
thread #0 : 3
thread #1 : 5
thread #0 : 2
thread #1 : 4
thread #0 : 1
thread #1 : 3
thread #0 finished
thread #1 : 2
thread #1 : 1
thread #1 finished
all threads are finished.
```

شکل ۲: خروجی برنامه `green_test`

۵ فاز دوم پیاده سازی کتابخانه(۱۰۰+۲۰۰ نمره)

در پیاده سازی که در قسمت اول انجام شد، تعویض متن تنها در صورتی انجام می‌شد که خود نخ اعلام کند نیازی به cpu ندارد (یا منتظر نخ دیگری شده باشد.) در این قسمت قصد داریم روند فوق را خودکار کنیم. یعنی به گونه ای برنامه ریزی انجام دهیم که در زمان های مقرر عملیات تعویض متن انجام شود(صرف نظر از اینکه نخ در حال اجرای چه دستوری است). برای انجام اینکار نیاز داریم به طریقی تابع `green_yield` را به طور خودکار در زمان های خاصی صدا بزنیم. برای تحریک خودکار می‌توانیم از سیگنال ها استفاده کنیم.

الف) در مورد نحوه پیاده سازی interrupt timer ها تحقیق کنید و برنامه ای بنویسید که هر ثانیه یکبار پیام Hi را بر روی صفحه نمایش چاپ کند.(توجه کنید که باید از تایمر استفاده کنید و تابع main نباید درگیر باشد!)[۳]

ب) اکنون که با روند کار با interrupt timer ها آشنا شدید، کتابخانه خود را تکمیل کنید؛ بگونه ای که تعویض متن هر $10\mu s$ یکبار بطور خودکار انجام شود.

ج) برنامه را چندین بار اجرا کنید. چه اتفاقی می افتد!؟ فکر می کنید علت آن چیست؟ راه حلی برای حل مشکل ارائه دهید.

د) (اختیاری) راه حلی که برای حل مشکل ارائه دادید را پیاده سازی کنید، بگونه ای که مشکل بطور کامل حل شود.

[۳] نکته ای که باید به آن توجه کنید این است که درون تابع handler نمی توانید از توابعی مانند printf استفاده کنید، چون asynch_signal_safe نیستند. در صورتی که قصد دارید پیامی چاپ کنید می توانید از فراخوانی سیستمی write استفاده کنید.

شیوه تحویل

برای این تمرین می بایست یک فلدر به نام studentid_prj2 بسازید (به جای studentid باید شماره دانشجویی خود را قرار دهید) که شامل موارد زیر باشد:

۱. یک فایل pdf: شامل پاسخ به سوالات این گزارش (ترجیحا به زبان فارسی) به همراه خروجی های کد ها و توضیح در مورد کارکرد آن‌ها که می بایست با استفاده از LaTeX ایجاد شده باشد.

۲. یک فلدر که همان green است که شما آن را کامل کرده اید. بعد از اتمام کار می‌توانید با یک بار اجرای دستور زیر، در داخل این فلدر، فایل های غیر source code را پاک نمایید تا حجم فایل ارسالی بی جهت بزرگ نشود.

```
make clean
```

سپس فلدر خود را با دستور زیر بایگانی و فشرده سازی کنید.

```
tar zcf studentid_prj2.tgz studentid_prj2
```

و تنها فایل studentid_prj2.tgz را در سامانه یکتا در قسمت مربوط به پروژه دوم بارگذاری کنید.

موفق باشید