پروژه دوم
درس سیستم عامل

پیاده سازی کتابخانه GREEN-THREAD

اعضای گروه

سارا برادران

فریدا فرپور

دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

ترم اول سال تحصیلی ۱۴۰۰ - ۱۳۹۹

۱.۰ تمرین اول

۱.۱.۰ قسمت الف

همانطور که در تصویر زیر مشخص است ضمن اجرای برنامه عبارت I am running به صورت مداوم هر ۱ ثانیه یکبار تا ابد چاپ خواهد شد حال به بررسی علت این امر می پردازیم: دستور getcontext(&one) استیت فعلی برنامه را در داخل متغیر one که از جنس ucontext است ذخیره می کند منظور از استیت فعلی برنامه مقادیر موجود در رجیستر های cpu من جمله رجیستر pc و... می باشد سپس عبارت I am running در خروجی چاپ شده و پس از وقفه ۱ ثانیه ای که به وسیله sleep(1) ایجاد شده است ، توسط دستور swapcontext(&two, &one) استیت فعلی برنامه ی درحال اجرا در در داخل متغیر two ذخیره شده و استیتی که پیشتر در داخل متغیر one ذخیره شده بود بر روی cpu اجرا می گردد بدین ترتیب با توجه به اینکه مقدار رجیستر pc ذخیره شده در ucontext one به خطی از برنامه که حاوی دستور getcontext(&one) بود اشاره می کند لذا مجددا برنامه به این خط از برنامه بازگشته مجددا عبارت I am running را چاپ کرده و این روند تا ابد تکرار می گردد لذا عمل چاپ عبارت I am running هرگز متوقف نمی شود. در حقیقت برنامه همانند یک حلقه عمل میکند. تصویر زیر اجرای این برنامه را نشان می دهد.

```
→ Desktop gcc ucontext-program.c
→ Desktop ./a.out
I am running :)
I am running :)
I am running :)
I am running :)
I am running :)
I am running :)
I am running :)
I am running :)
I am running :)
```

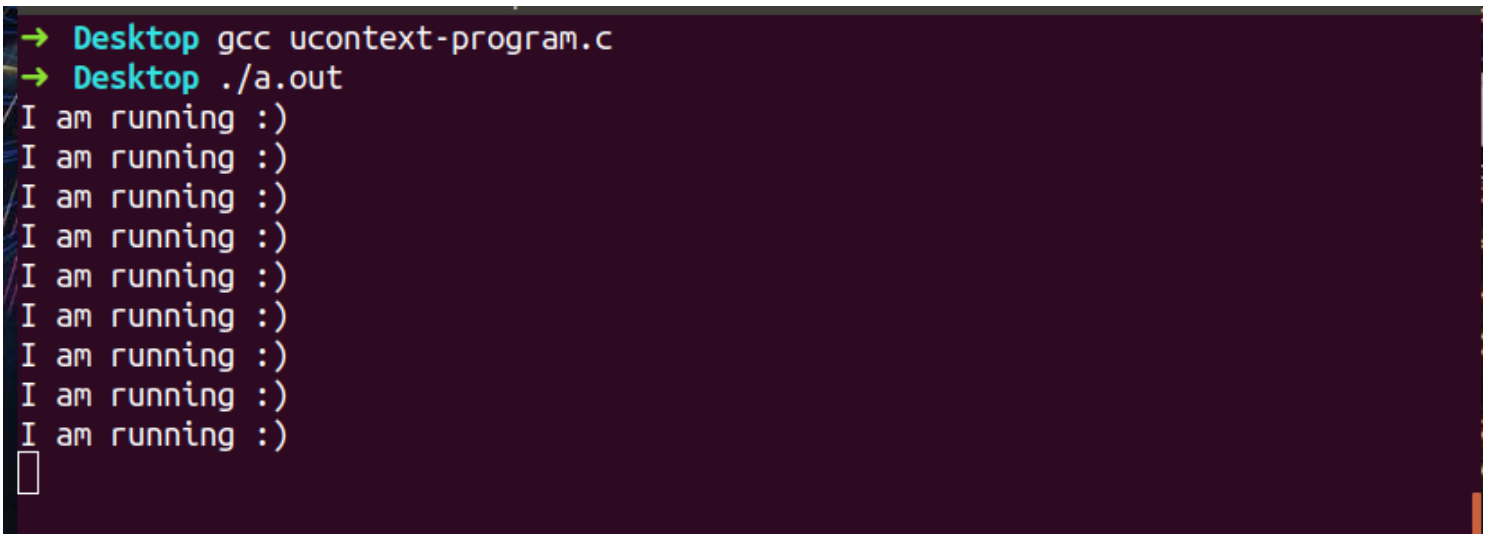
۲.۱.۰ قسمت ب

ب) برای اینکه صرفا ۲ بار عبارت I am running در خروجی چاپ گردد می بایست فقط و فقط یکبار دستور swapcontext(&two, &one) اجرا گردد بدین منظور می توان یک متغیر با مقدار اولیه صفر پیش از دستور getcontext(&one) تعریف کرد مثلا int var = 0 سپس پیش از اجرای دستور swapcontext(&two, &one) مقدار var را ۱ کرده و دو دستور swapcontext(&two, &one); Var = 1; را در داخل یک بلاک شرطی if(var == 0) قرار دهیم بدین ترتیب پس از یکبار اجرای تابع swapcontext مقدار متغیر var برابر ۱ شده و پس از بازگشت به خط حاوی دستور getcontext(&one) و چاپ مجدد عبارت I am running دیگر وارد بلاک if نمیگردد و لذا برنامه خاتمه می یابد بدون آنکه استیت برنامه مجددا تغییر کرده و به استیت ذخیره شده در one باز گردد. اگر خطوط کامنت شده را به کد اضافه کنیم دقیقا مشابه آنچه در بخش فوق توضیح داده شد تابع swapcontext فقط و فقط یکبار اجرا می شود و لذا نحوه اجرای برنامه بدین ترتیب خواهد بود : برنامه یکبارعبارت I am running را چاپ کرده سپس به دلیل صفر بودن مقدار done وارد بلاک if شده متغیر done را ۱ کرده و عمل swapcontext رخ می دهد و لذا صرفا یکبار به خط دستور getcontext(&one) بازگشته مجددا عبارت I am running را چاپ کرده و در ادامه به دلیل ۱ بودن متغیر done دیگر وارد بلاک if نمی شود و لذا دیگر هیچگونه عمل swapcontext اتفاق نخواهد افتاد و برنامه تا خط پایانی و رسیدن به return 0 اجرا گشته و خاتمه می یابد. تصویر زیر اجرای این برنامه همراه با متغیر done را نشان می دهد.

```
File  Edit  View  Search  Terminal  Help
→ Desktop gcc ucontext-program.c
→ Desktop ./a.out
I am running :)
I am running :)
→ Desktop
```

۳.۱.۰ قسمت ج

اگر به جای int done از register int done استفاده شود مجددا نحوه خروجی ها مشابه قسمت الف خواهد بود و عبارت I am running هر یک ثانیه یکبار تا ابد در خروجی چاپ خواهد شد. علت این امر واضح است زیرا نحوه تعریف متغیر به صورت رجیستر به کامپایلر این مجوز را می دهد که متغیر done می تواند به جای ذخیره شدن در استک در داخل رجیستر های cpu ذخیره شود از آن جایی که ضمن swapcontext مقدار رجیستر ها بازنشانی و rewrite می شود می توان گفت زمانی که دستور getcontext(&one) اجرا میگردد مقدار متغیر done برابر صفر است و لذا رجیستری از cpu که به این متغیر اختصاص یافته در این استیت حاوی مقدار ۰ خواهد بود با توجه به این قضیه اگرچه پیش از اجرای دستور swapcontext(&two, &one) در هر بار مقدار done برابر یک می شود اما پس از عمل swapcontext مقدار رجیستر مربوط به متغیر done با مقدار ذخیره شده این رجیستر در ucontext one جایگزین شده و مجددا صفر میگردد و لذا عملا بلاک if بی اثر شده و برنامه هر بار وارد این بلاک می گردد. تصویر زیر اجرای این برنامه را همراه با متغیر register int done نشان می دهد.



۲.۰ تمرین دوم

۱.۲.۰ قسمت الف

به نظر میرسد در این برنامه قصد داریم از دو متغیر ucontext به عنوان دو نخ برای اجرای همزمان تابع do-job یکبار با آرگومان های ۱و ۳ و بار دیگر با آرگومان های ۲و ۵ استفاده کنیم خطوط اولیه برنامه این دو متغیر از جنس ucontext را initialize کرده و به کمک دستورات getcontext (&mContext[0]) و getcontext(&mContext[1]) استیت های برنامه حین اجرای این دو دستور را به ترتیب در mContext[0] و mContext[1] ذخیره می کند به علاوه به هر یک از آن ها یک استک با سایز 4k اختصاص یافته و آدرس این دو استک در بخش sssp از ساختار ucontext و سایز استک در بخش sssize از ساختار ذخیره می شود. و توسط تابع makecontext این دو نخ برای اجرا تابع هدف do-job آماده خواهند شد. سپس برنامه آغاز شده و در وهله اول مقدار running task id برابر صفر قرار گرفته است.تابع swapcontext(&mContextmain , mContext) استیت فعلی برنامه را در mContextmain ذخیره کرده و استیت فعلی برنامه را با استیت ذخیره شده در متغیر mContext[0] جایگزین می کند. نخ mContext[0] شروع به کار کرده و تابع dojob را با آرگومان های ۱و ۳ اجرا میکند for اول اجرا شده و سپس طی فراخوانی تابع yield ، نخ صفر cpu را رها کرده و استیت برنامه با استیت موجود در mContext[1] جایگزین خواهد شد. سپس نخ ۱ بر روی cpu اجرا گشته و تابع dojob را با آرگومان های ۲و ۵ اجرا می کند. مجددا for اول توسط این نخ اجرا گشته و سپس cpu را توسط تابع yield به نخ ۰ تسلیم می کند و لذا نخ صفر بر روی cpu قرار گرفته و از خط پس از ()yield اجرا گشته و پس از اجرای حلقه دوم خاتمه می یابد علی رغم اینکه کار نخ ۱ پایان نیافته بود اما دیگر cpu در اختیار ترد ۱ و برنامه main قرار نمی گیرد و عمل اجرا برنامه خاتمه می یابد. لذا عملکرد این کد صحیح نبوده و می بایست سیاستی اتخاذ شود که پس از اتمام کار ترد ۰ مجددا cpu در اختیار ترد های باقی مانده قرار بگیرد. (عکس از اجرای کد تمرین ۲)

```
→ Desktop ./a.out
#1 -> -3
#1 -> -2
#1 -> -1
yield from #0 to #1
#2 -> -5
#2 -> -4
#2 -> -3
#2 -> -2
#2 -> -1
yield from #1 to #0
#1 -> 1
#1 -> 2
#1 -> 3
→ Desktop
```

۲.۲.۰ قسمت ب

کد نوشته شده در این قسمت به گونه ای عمل می کند که ایرادات موجود در کد قسمت الف را رفع نماید بدین ترتیب بخش uclink از ساختارمتغیر های ucontext مربوط به نخ ۰ و ۱ به نخ mContextdone اشاره می کنند ولذا بدین ترتیب پس از آنکه اجرای تابع dojob توسط هر یک از نخ ها خاتمه یافت نخ mContextdone به صورت خودکار بر روی cpu قرار گرفته و این نخ تابع donejob را اجرا می نماید این تابع ابتدا فلگ وضعیت نخ اجرا شده در مرحله قبل (نخ با شماره running task id) را یک کرده و وضعیت فلگ نخ دیگر را بررسی می کند چنانچه کار این نخ به پایان نرسیده باشد متغیر running task id با شماره نخ دیگر مقدار دهی شده و توسط دستور swapcontext نخی که بخشی از کار آن باقی مانده است را بر روی cpu اجرا کرده و درصورتی که فلگ وضعیت نخ دیگر نیز ۱ باشد یعنی هر دو نخ تماما تابع مربوطه را اجرا کرده و خاتمه یافته باشند آن گاه صرفا متغیر done را برابر ۱ کرده ، از حلقه خارج شده و نخ mContextdone نیز پایان می یابد و لذا پس از آن نخی که بخش uclink نخ قبلی (mContextdone) بدان اشاره می کند یعنی همان ترد main بر روی cpu قرار گرفته و اجرا می گردد بدین ترتیب ترد main نیز تماما اجرا شده و برنامه به صورت مورد انتظار پایان می پذیرد. در برنامه مورد نظر از دو فلگ مجزای done-flag به منظور بررسی وضعیت هر نخ و از یک فلگ done برای بررسی وضعیت خاتمه کار هر دو نخ استفاده شده است. (عکس از اجرای ex2 fixed)

```
→ Desktop ./a.out
#1 -> -3
#1 -> -2
#1 -> -1
yield from #0 to #1
#2 -> -5
#2 -> -4
#2 -> -3
#2 -> -2
#2 -> -1
yield from #1 to #0
#1 -> 1
#1 -> 2
#1 -> 3
Process #0 terminated
#2 -> 1
#2 -> 2
#2 -> 3
#2 -> 4
#2 -> 5
Process #1 terminated
returned to main
→ Desktop
```

۳.۰ پیاده سازی فاز اول

تصویر زیر نحوه اجرای برنامه ضمن پیاده سازی فاز اول را نشان می دهد به این ترتیب هر ترد پس از یک مرحله اجرا با فراخوانی تابع greenyield() پردازنده را رها کرده و آن را به ترد دیگر تسلیم می کند و این روند تا خاتمه کار هر ترد ادامه می یابد.

```
→ green ./green_test
thread #0 : 7
thread #1 : 7
thread #2 : 4
thread #0 : 6
thread #1 : 6
thread #2 : 3
thread #0 : 5
thread #1 : 5
thread #2 : 2
thread #0 : 4
thread #1 : 4
thread #2 : 1
thread #0 : 3
thread #1 : 3
thread #2 finished
thread #0 : 2
thread #1 : 2
thread #0 : 1
thread #1 : 1
thread #0 finished
thread #1 finished
Result for thread #0: 28
Result for thread #1: 28
Result for thread #2: 10
all threads are finished.
→ green
```

۱.۳.۰ پیاده سازی لینک لیست

تصویر زیر نتیجه پیاده سازی linked list و افزودن ترد های اجرایی به آن را نمایش می دهد همانطور که مشخص است در تابع greencreate() پس از ایجاد یک ترد جدید ترد مذکور به انتهای لینک لیست موجود اضافه شده و وضعیت لینک لیست چاپ می گردد (در این مرحله برای اختصاص یک id به هر ترد در داخل ساختار greent یک متغیر تحت عنوان gtid اضافه شده است که ضمن ایجاد هر ترد به صورت ترتیبی یک id به آن اختصاص داده می شود و تابع greenprint() پیاده سازی شده اقدام به چاپ لینک لیست حاوی ترد های ایجاد شده می نماید.)

```
➜ green ./green_test
main thread with gtid : 1
next thread with gtid : 2
=== new linklist ===
1 --> 2 -->  NULL
next thread with gtid : 3
=== new linklist ===
1 --> 2 --> 3 -->  NULL
next thread with gtid : 4
=== new linklist ===
1 --> 2 --> 3 --> 4 -->  NULL
thread #0 : 6
thread #1 : 2
thread #2 : 9
thread #0 : 5
thread #1 : 1
thread #2 : 8
thread #0 : 4
thread #1 finished
thread #2 : 7
thread #0 : 3
thread #2 : 6
thread #0 : 2
thread #2 : 5
thread #0 : 1
thread #2 : 4
thread #0 finished
thread #2 : 3
Result for thread #0: 21
Result for thread #1: 3
thread #2 : 2
thread #2 : 1
thread #2 finished
Result for thread #2: 45
all threads are finished.
➜ green
```

۴.۰ پیاده سازی فاز دوم

۱.۴.۰ قسمت الف

تصویر زیر نتیجه اجرای برنامه ای را نشان می دهد که با استفاده از timer interrupt هر یک ثانیه یکبار عبارت Hi را بر روی خروجی چاپ می کند فایل این برنامه تحت عنوان timertest.c ضمیمه شده است.

```
→ green ./a.out
Hi
Hi
Hi
Hi
Hi
```

```c
#define PERIOD 1
static sigset_t block;
struct sigaction act1;
struct timeval interval;
struct itimerval period;

char message[5];
void timer_handler(int signo){

    write( 1, message, sizeof( message ) );
}

int main(){

    bzero(message, 5);
    sprintf(message, "%s","Hi\n");

    sigemptyset(&block);
    sigaddset(&block, SIGVTALRM);
    act1.sa_flags = 0;
    act1.sa_mask = block;
    act1.sa_handler = timer_handler;
    assert(!sigaction(SIGVTALRM, (struct sigaction * ) & act1, NULL));
    interval.tv_sec = PERIOD;
    interval.tv_usec = 0;
    period.it_interval = interval;
    period.it_value = interval;
    setitimer(ITIMER_VIRTUAL, &period, NULL);

    while(1);

}
```

۲.۴.۰ قسمت ب استفاده از timer interrupt در کتابخانه green-thread

تصویر زیر نحوه اجرای برنامه ضمن پیاده سازی فاز دوم را نشان می دهد به این ترتیب هر ترد پس از اجرا به مدت n میکرو ثانیه cpu را تسلیم ترد بعدی کرده و مجددا در انتهای صف ترد ها قرار می گیرد. بدین ترتیب دیگر نیازی به فراخوانی تابع ()green-yield توسط ترد ها نمی باشد و این تابع در بازه های زمانی که تایمر time out می کند به صورت خودکار اجرا خواهد شد.

```
→ green ./green_test
thread #0 : 9
thread #1 : 3
thread #1 : 2
thread #1 : 1
thread #1 finished
thread #2 : 10
thread #2 : 9
thread #2 : 8
thread #2 : 7
thread #2 : 6
thread #2 : 5
thread #0 : 8
thread #0 : 7
thread #0 : 6
thread #0 : 5
thread #0 : 4
thread #0 : 3
thread #0 : 2
thread #0 : 1
thread #0 finished
thread #2 : 4
thread #2 : 3
thread #2 : 2
thread #2 : 1
thread #2 finished
Result for thread #0: 45
Result for thread #1: 6
Result for thread #2: 55
all threads are finished.
→ green
```

۳.۴.۰ قسمت ج

در حالت کلی برنامه نوشته شده در این قسمت در برخی از مراتب اجرا با خطا برخورد کرده و به درستی خاتمه نمی پذیرد. مشکل برنامه در حقیقت ناشی از این امر است که ۱۰ میکروثانیه زمان بسیار کمی است که در ualarm استفاده شده و به همین دلیل اگر فرضا در یکی از توابع در حال انجام عملیات تعویض متن باشیم و در همین هنگام interupt رخ دهد آنگاه صف ترد ها تغییر کرده و برنامه می تواند در ادامه دچار مشکل شود و یا سناریو ای که در آن ضمن اجرای دستور malloc وقفه رخ داده و تعویض متن صورت پذیرد می تواند برنامه را با خطا مواجه سازد و بدین ترتیب هر زمان که قرار است ترد اجرایی بخشی از کار خود را انجام دهد مجددا تایمر متوقف شده و برنامه به interupt برخورد می کند و عمل تعویض متن صورت می پذیرد به همین سبب در مراتبی از اجرا برنامه با مشکل مواجه می شود. به صورت اولیه راه حل این مشکل افزایش زمان تایمر است برای مثال اگر به جای استفاده از تایمر ۱۰ میکرو ثانیه از تایمر با ۱۰۰۰ میکرو ثانیه استفاده کنیم خطای برنامه اطلاح خواهد شد

۴.۴.۰ قسمت د

در حالت کلی یک راه حل بهتر و اصولی تر می تواند این باشد که در ابتدای توابع interrupt مربوطه را غیر فعال کرده و مجددا در انتهای توابع آن را فعال نماییم به این ترتیب ضمن اجرای یک تابع timer interrupt ها بافر شده و پس از اتمام کار تابع مجددا فعال خواهند شد. بدین منظور از دستور sigprocmask(SIG-BLOCK, &block, NULL) جهت فعال کردن و از دستور sigprocmask(SIG-UNBLOCK, &block, NULL) جهت غیر فعال کردن وقفه ها استفاده می نماییم. بدین منظور می توان توابع فعال و غیر فعال کردن وقفه ها را که در برنامه green2.c در ابتدا و انتهای توابع قرار داده شده uncomment کرد. البته باید توجه داشت که راه حل عنوان شده بهینه نیست زیرا در اکثر موارد وقفه ها بافر می شوند و لذا میزان multi threading برنامه کاهش می یابد.

۵.۰ uname

دستور uname -r در حقیقت ورژن کرنل لینوکس را به عنوان خروجی باز می گرداند (-r : kernel version number and release level) به وسیله قرار دادن این دستور در $() در حقیقت همانند این است که خروجی این دستور را با عبارت sudo apt-get install linux-headers- کانکت کرده و اجرا نماییم در واقع دو دستور زیر در سیستم با کرنل ورژن 4.15.0-128-generic به صورت یکسان اجرا خواهند شد.

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

sudo apt-get install linux-headers-4.15.0-128-generic
```

```
➜  ~ uname
Linux
➜  ~ uname -r
4.15.0-128-generic
➜  ~
```

```
➜  ~ sudo apt-get install linux-headers-$(uname -r)
Reading package lists... Done
Building dependency tree
Reading state information... Done
linux-headers-4.15.0-128-generic is already the newest version (4.15.0-128.131).
The following packages were automatically installed and are no longer required:
  libqt5positioning5 libqt5qml5 libqt5quick5 libqt5sensors5 libqt5webchannel5
  libqt5webkit5 libxmlb1 phantomjs python3-dateutil python3-pyxattr rtmpdump
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 79 not upgraded.
➜  ~
```