Lab5
实验报告

姓名 李晓畅

学号 20307130261

班级 计算机科学技术

实现思路和问题回答

3.1 UTHREAD: SWITCHING BETWEEN THREADS

In this exercise you will design the context switch mechanism for a user-level threading system, and then implement it. To get you started, your xv6 has two files user/uthread.c and user/uthread_switch.S, and a rule in the Makefile to build a uthread program. uthread.c contains most of a user-level threading package, and code for three simple test threads. The threading package is missing some of the code to create a thread and to switch between threads.

> Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, `make grade` should say that your solution passes the `uthread` test.

Once you've finished, you should see the following output when you run `uthread` on xv6 (the three threads might start in a different order):

```
$ make qemu
...
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
...
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

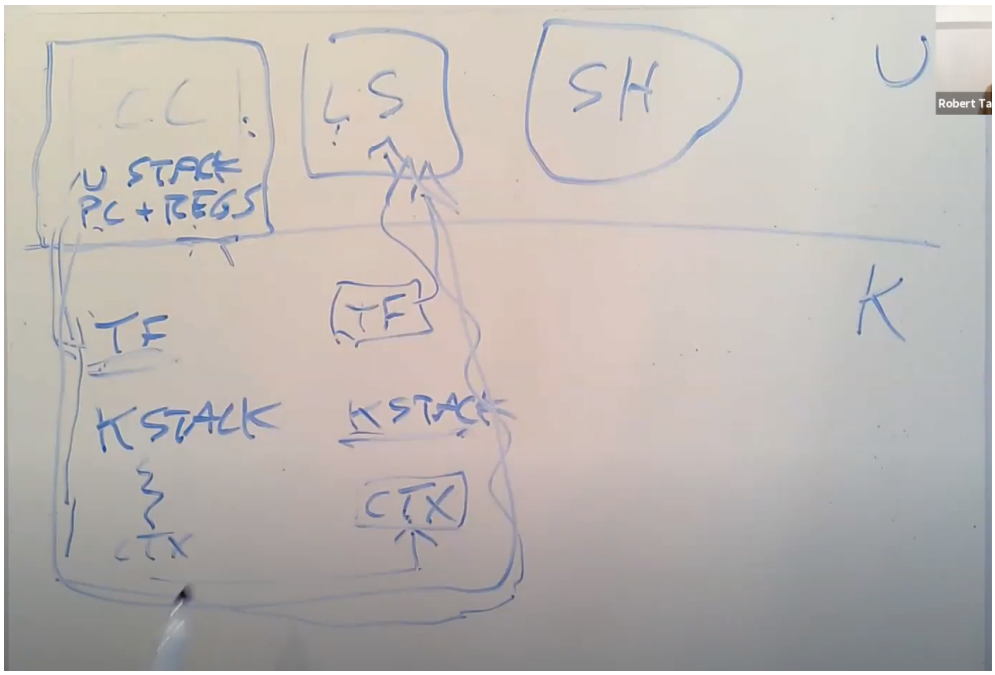
This output comes from the three test threads, each of which has a loop that prints a line and then yields the CPU to the other threads.

At this point, however, with no context switch code, you'll see no output.

You will need to add code to `thread_create()` and `thread_schedule()` in `user/uthread.c`, and `thread_switch` in `user/uthread_switch.S`. One goal is ensure that when `thread_schedule()` runs a given thread for the first time, the thread executes the function passed to `thread_create()`, on its own stack. Another goal is to ensure that `thread_switch` saves the registers of the thread being switched away from, restores the registers of the thread being switched to, and returns to the point in the latter thread's instructions where it last left off. You will have to decide where to save/restore registers; modifying `struct thread` to hold registers is a good plan. You'll need to add a call to `thread_switch` in `thread_schedule`; you can pass whatever arguments you need to `thread_switch`, but the intent is to switch from thread `t` to `next_thread`.

其实最关键的一点是要了解线程切换的过程到底是什么样的。

概念上说大概是：



分为下面几个步骤：

1. XV6 会首先会将 CC 程序的内核线程的内核寄存器保存在一个 context 对象中。

2.类似的，因为要切换到 LS 程序的内核线程，那么 LS 程序现在的状态必然是 RUNABLE，表明 LS 程序之前运行了一半。这同时也意味着 LS 程序的用户空间状态已经保存在了对应的 trapframe 中，更重要的是，LS 程序的内核线程对应的内核寄存器也已经保存在对应的 context 对象中。所以接下来，XV6 会恢复 LS 程序的内核线程的 context 对象，也就是恢复内核线程的寄存器。

3.之后 LS 会继续在它的内核线程栈上，完成它的中断处理程序（注，假设之前 LS 程序也是通过定时器中断触发的 pre-emptive scheduling 进入的内核）。

4.然后通过恢复 LS 程序的 trapframe 中的用户进程状态，返回到用户空间的 LS 程序中。

5.最后恢复执行 LS

下面可以按照提示进行：

- `thread_switch` needs to save/restore only the callee-save registers. Why?

参照上面的流程，Trapframe 已经实现了，那么我们在用户态下就拷贝一个寄存器状态就可以了 qwq。

个人决定还是把 context 包含在 thread 里比较好，包含在 trapframe 里其实应该也可以，但这样的话我们还要修改对应的代码。

```
struct thread {
  char        stack[STACK_SIZE]; /* the thread's stack */
  int         state;             /* FREE, RUNNING, RUNNABLE */
  struct context context;
};
```

添加一个 context。虽然其实这个 context 和 proc.h 里包含的一样，实际上也确实只需要这些东西就可以了，不过最好还是不要，

```
// #include "kernel/proc.h"
```

因为还要一些东西才能编译，而且逻辑上关系没有那么大。抄一个过来就可以。

```
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

- • You can see the assembly code for uthread in user/uthread.asm, which may be handy for debugging.

切换 context 就照抄 swtch 就可以了。

```
thread_switch:
    /* YOUR CODE HERE */
    //just copy?
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)
        ret    /* return to ra */
```

这里注意一下它给的是一个基地址+变址的形式，所以我们应该传入两个地址作为参数，不然就要改代码。

- 具体实现就不难了。

在 thread_schedule 里简单调用一下。不用应该也可以，但是直接用内核的 swtch 肯定会增加切换的成本，实际上也没必要非得进入内核来切换 context。

```
  /* YOUR CODE HERE
   * Invoke thread_switch to switch from t to next_thread:
   * thread_switch(??, ??);
   */
  thread_switch((uint64) &t->context, (uint64) &next_thread->context);
```

另外在创建的时候添加对 context 的初始化。

```
101     t->context.sp = (uint64) t->stack+STACK_SIZE;
102
103     t->context.ra = (uint64) func;
```

大概就可以了。

测试结果截图：

```
$ make qemu-gdb
uthread: OK (7.4s)
```

运行成功

3.2 USING THREADS

In this assignment you will explore parallel programming with threads and locks using a hash table. You should do this assignment on a real Linux or MacOS computer (not xv6, not qemu) that has multiple cores. Most recent laptops have multicore processors.

This assignment uses the UNIX `pthread` threading library. You can find information about it from the manual page, with `man pthreads`, and you can look on the web, for example here, here, and here.

The file `notxv6/ph.c` contains a simple hash table that is correct if used from a single thread, but incorrect when used from multiple threads. In your main xv6 directory (perhaps `~/xv6-labs-2021`), type this:

```
$ make ph
$ ./ph 1
```

Note that to build `ph` the Makefile uses your OS's gcc, not the 6.S081 tools. The argument to `ph` specifies the number of threads that execute put and get operations on the the hash table. After running for a little while, `ph 1` will produce output similar to this:

```
100000 puts, 3.991 seconds, 25056 puts/second
0: 0 keys missing
100000 gets, 3.981 seconds, 25118 gets/second
```

The numbers you see may differ from this sample output by a factor of two or more, depending on how fast your computer is, whether it has multiple cores, and whether it's busy doing other things.

`ph` runs two benchmarks. First it adds lots of keys to the hash table by calling `put()`, and prints the achieved rate in puts per second. The it fetches keys from the hash table with `get()`. It prints the number keys that should have been in the hash table as a result of the puts but are missing (zero in this case), and it prints the number of gets per second it achieved.

You can tell `ph` to use its hash table from multiple threads at the same time by giving it an argument greater than one. Try `ph 2`:

```
$ ./ph 2
100000 puts, 1.885 seconds, 53044 puts/second
1: 16579 keys missing
0: 16579 keys missing
200000 gets, 4.322 seconds, 46274 gets/second
```

The first line of this `ph 2` output indicates that when two threads concurrently add entries to the hash table, they achieve a total rate of 53,044 inserts per second. That's about twice the rate of the single thread from running `ph 1`. That's an excellent "parallel speedup" of about 2x, as much as one could possibly hope for (i.e. twice as many cores yielding twice as much work per unit time).

However, the two lines saying `16579 keys missing` indicate that a large number of keys that should have been in the hash table are not there. That is, the puts were supposed to add those keys to the hash table, but something went wrong. Have a look at `notxv6/ph.c`, particularly at `put()` and `insert()`.

我们一个个解决问题：

- Why are there missing keys with 2 threads, but not with 1 thread? Identify a sequence of events with 2 threads that can lead to a key being missing. Submit your sequence with a short explanation in answers-thread.txt

因为临界资源，即对应的桶，没有被保护，导致多线程情况下可能会对同一个桶的同一个链表表元并发写，导致后插入的元素实际上覆盖了之前插

入的元素，导致丢失。而在单线程条件下，一切都是顺序进行的，不会有并发问题。

具体说，两个进程可能将两个数差不多同时 hash 到同一个桶，并且几乎同时执行到`insert(key, value, &table[i], table[i]);`这一步，不论是由于他们实际上在不同 CPU 上运行而实际上同时运行到这里，还是因为他们的时间片耗尽，导致有一个线程被停止都可能造成这样的情况。这时，他们传入 insert 的都是同一个地址，也就是最后一个表元，对最后一个表元实际上重复添加了两次，前一次被后一次覆盖了。

类似的问题还很多，比如重复插入也是有可能的，但概率应该要小很多。所以最终呈现的效果是丢失了不少。

To avoid this sequence of events, insert lock and unlock statements in `put` and `get` in `notxv6/ph.c` so that the number of keys missing is always 0 with two threads. The relevant pthread calls are:

```
pthread_mutex_t lock;           // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);      // acquire lock
pthread_mutex_unlock(&lock);    // release lock
```

You're done when `make grade` says that your code passes the `ph_safe` test, which requires zero missing keys with two threads. It's OK at this point to fail the `ph_fast` test.

Don't forget to call `pthread_mutex_init()`. Test your code first with 1 thread, then test it with 2 threads. Is it correct (i.e. have you eliminated missing keys?)? Does the two-threaded version achieve parallel speedup (i.e. more total work per unit time) relative to the single-threaded version?

There are situations where concurrent `put()`s have no overlap in the memory they read or write in the hash table, and thus don't need a lock to protect against each other. Can you change `ph.c` to take advantage of such situations to obtain parallel speedup for some `put()`s? Hint: how about a lock per hash bucket?

一次只允许一个线程访问临界区就好了。互斥变量为我们提供了这样的方法。

To avoid this sequence of events, insert lock and unlock statements in `put` and `get` in `notxv6/ph.c` so that the number of keys missing is always 0 with two threads. The relevant pthread calls are:

```
pthread_mutex_t lock;           // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);      // acquire lock
pthread_mutex_unlock(&lock);    // release lock
```

You're done when `make grade` says that your code passes the `ph_safe` test, which requires zero missing keys with two threads. It's OK at this point to fail the `ph_fast` test.

把 hash 表作为临界资源直接上锁是最直接简单的方法。不过这样做性能堪忧，归根结底我们还是只能一个一个地对 hash 表写。提示说得很清楚了，可以考虑给每个桶上一个锁。

```
pthread_mutex_t locks[NBUCKET];

void
init(){
  for(int i=0;i<NBUCKET;i++){
    pthread_mutex_init(&locks[i],NULL);
  }
};
```

所以要为每个桶创建锁，并初始化。

```
117     init();
```

在进行读写时要先获得锁，简单这样实现就可以实现功能了，而且性能受到的影响也少不少。下面的代码就是这样实现的。

实际上或许我们可以把读和写分开？这样设一把读者写者锁，只是检查 hash 表的是读者，要修改的是写者，这样允许很多读者同时检查 hash 表，应该还可以提升性能。

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;

  pthread_mutex_lock(&locks[i]);
  // is the key already present?
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
  pthread_mutex_unlock(&locks[i]);

}
```

Modify your code so that some `put` operations run in parallel while maintaining correctness. You're done when `make grade` says your code passes both the `ph_safe` and `ph_fast` tests. The `ph_fast` test requires that two threads yield at least 1.25 times as many puts/second as one thread.

不过其实这样的性能也足够通过测试了 qwq。

测试结果截图：

```
== Test ph_safe == make[1]: 进入目录“/home/wojtek/Desktop/xv6-lab5-2022”
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/wojtek/Desktop/xv6-lab5-2022”
ph_safe: OK (16.5s)
== Test ph_fast == make[1]: 进入目录“/home/wojtek/Desktop/xv6-lab5-2022”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/wojtek/Desktop/xv6-lab5-2022”
ph_fast: OK (34.5s)
```

运行成功

3.2 USING THREADS

In this assignment you'll implement a barrier: a point in an application at which all participating threads must wait until all other participating threads reach that point too. You'll use pthread condition variables, which are a sequence coordination technique similar to xv6's sleep and wakeup.

You should do this assignment on a real computer (not xv6, not qemu).

The file `notxv6/barrier.c` contains a broken barrier.

```
$ make barrier
$ ./barrier 2
barrier: notxv6/barrier.c:42: thread: Assertion `i == t' failed.
```

The 2 specifies the number of threads that synchronize on the barrier (`nthread` in `barrier.c`). Each thread executes a loop. In each loop iteration a thread calls `barrier()` and then sleeps for a random number of microseconds. The assert triggers, because one thread leaves the barrier before the other thread has reached the barrier. The desired behavior is that each thread blocks in `barrier()` until all `nthreads` of them have called `barrier()`.

> Your goal is to achieve the desired barrier behavior. In addition to the lock primitives that you have seen in the `ph` assignment, you will need the following new pthread primitives; look here and here for details.
>
> ```
> pthread_cond_wait(&cond, &mutex);  // go to sleep on cond, releasing lock mutex, acquiring upon wake up
> pthread_cond_broadcast(&cond);     // wake up every thread sleeping on cond
> ```
>
> Make sure your solution passes `make grade`'s `barrier` test.

`pthread_cond_wait` releases the `mutex` when called, and re-acquires the `mutex` before returning.

We have given you `barrier_init()`. Your job is to implement `barrier()` so that the panic doesn't occur. We've defined `struct barrier` for you; its fields are for your use.

前面提到的互斥锁有一个明显到缺点: 只有两种状态，锁定和非锁定。

而条件变量则通过允许线程阻塞并等待另一个线程发送唤醒信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。

> Your goal is to achieve the desired barrier behavior. In addition to the lock primitives that you have seen in the `ph` assignment, you will need the following new pthread primitives; look here and here for details.
>
> ```
> pthread_cond_wait(&cond, &mutex);  // go to sleep on cond, releasing lock mutex, acquiring upon wake up
> pthread_cond_broadcast(&cond);     // wake up every thread sleeping on cond
> ```
>
> Make sure your solution passes `make grade`'s `barrier` test.

再注意考察两个要求：

There are two issues that complicate your task:

- You have to deal with a succession of barrier calls, each of which we'll call a round. `bstate.round` records the current round. You should increment `bstate.round` each time all threads have reached the barrier.
- You have to handle the case in which one thread races around the loop before the others have exited the barrier. In particular, you are re-using the `bstate.nthread` variable from one round to the next. Make sure that a thread that leaves the barrier and races around the loop doesn't increase `bstate.nthread` while a previous round is still using it.

我们就不难实现了：

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  pthread_mutex_lock(&bstate.barrier_mutex);
  bstate.nthread++;
  if(bstate.nthread == nthread){
    bstate.nthread=0;
    bstate.round++;
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  else{
    pthread_cond_wait(&bstate.barrier_cond,&bstate.barrier_mutex);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);

}
```

遇到路障的线程要更改路障的信息，因而要先获得路障的锁。如果阻塞在路障上的线程足够多，就把他们全部唤醒放行，这时轮数增加，被阻塞线程数清零。不够多线程就必须被阻塞。完成后必须解锁路障信息。

测试结果：

```
== Test barrier == make[1]: 进入目录“/home/wojtek/Desktop/xv6-lab5-2022”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录“/home/wojtek/Desktop/xv6-lab5-2022”
barrier: OK (8.9s)
```

成功。

问题和解决

如何实现 CONTEXT

在第一部分加入 context 后就有这样的错：

那是因为编译器并不知道我们说的 context 是什么。

如果直接加入

```
4  // #include "kernel/proc.h"
```

或许并不是一个好选择，虽然我们只需要 context,但 C 可不允许我只 import 我想要的东西。这就意味着如果全部加进去的话就有更多相关的部分需要被加进去，即使这些部分和我们的实现一点关系都没有。

所以直接抄一份或许是更好的选择。

然后就可以了。

```
14  struct context {
15    uint64 ra;
16    uint64 sp;
17
18    // callee-saved
19    uint64 s0;
20    uint64 s1;
21    uint64 s2;
22    uint64 s3;
23    uint64 s4;
24    uint64 s5;
25    uint64 s6;
26    uint64 s7;
27    uint64 s8;
28    uint64 s9;
29    uint64 s10;
30    uint64 s11;
31  };
```

另外，因为 xv6 一个进程对应的是一个线程，所以其实就用 trapframe 里的那个应该也行。但问题在于这就要改其他（很多）地方的代码，而且逻辑上进程和线程也不是同样的东西，所以最好不要这样做。这应该也不是作者希望得到的解决方法。

为什么要有线程?

Xv6 系统中一个进程只包含一个进程，线程调度实际上还得设计进程调度，在此基础上我们实际上还额外进行了一层抽象，要保存恢复更多信息，即便可能是重复的。那这样的线程有意义吗?

回顾线程的意义:

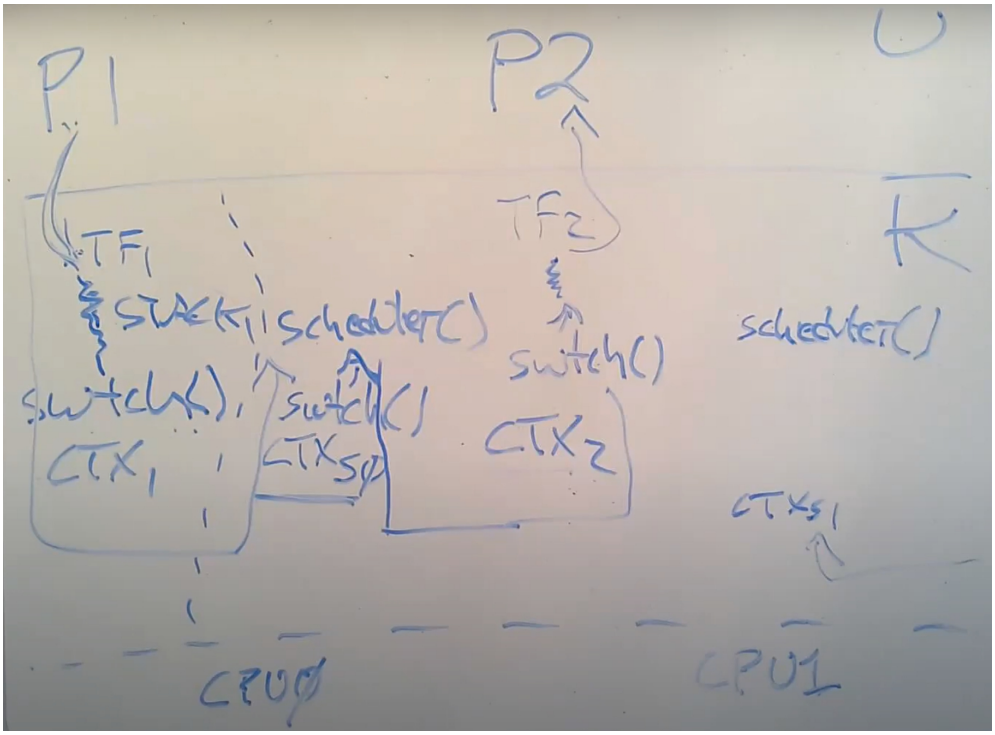
- 首先，人们希望他们的计算机在同一时间不是只执行一个任务。有可能计算机需要执行分时复用的任务。甚至在一个单用户的计算机或你会运行多个进程，并期望计算机完成所有的任务而不仅仅只是一个任务。
- 其次，多线程可以让程序的结构变得简单。线程在有些场合可以帮助程序员将代码以简单优雅的方式进行组织，并减少复杂度。
- 最后，使用多线程可以通过并行运算，在拥有多核 CPU 的计算机上获得更快的处理速度。常见的方式是将程序进行拆分，并通过线程在不同的 CPU 核上运行程序的不同部分。你可以认为 XV6 就是一个多 CPU 并行运算的程序。

从中我们至少我们可以确定一点，线程用起来比进程方便得多。实际上我们用起来大概也是这样的，创建线程更方便。也就是所谓"简单优雅"。

实验总结

本次实验中我们学习了 xv6 的进程调度。

具体过程大概是这样的:



考虑时间片用尽了被迫切换的线程会经历什么：

- 首先一个定时器中断强迫 CPU 从用户空间进程切换到内核，trampoline 代码将用户寄存器保存于用户进程对应的 trapframe 对象中；
- 之后在内核中运行 usertrap，来实际执行相应的中断处理程序。这时，CPU 正在进程 P1 的内核线程和内核栈上，执行内核中普通的 C 代码；
- 假设进程 P1 对应的内核线程决定它想出让 CPU，它会做很多工作，这个我们稍后会看，但是最后它会调用 swtch 函数（译注：switch 是 C 语言关键字，因此这个函数命名为 swtch 来避免冲突），这是整个线程切换的核心函数之一；
- swtch 函数会保存用户进程 P1 对应内核线程的寄存器至 context 对象。所以目前为止有两类寄存器：用户寄存器存在 trapframe 中，内核线程的寄存器存在 context 中。
- 在 schedulder 函数中会做一些清理工作，例如将进程 P1 设置成 RUNABLE 状态。之后再通过进程表单找到下一个 RUNABLE 进程。假设找到的下一个进程是 P2（虽然也有可能找到的还是 P1），schedulder 函数会再次调用 swtch 函数，完成下面步骤：
- 先保存自己的寄存器到调度器线程的 context 对象
- 找到进程 P2 之前保存的 context，恢复其中的寄存器

- 因为进程 P2 在进入 RUNABLE 状态之前，如刚刚介绍的进程 P1 一样，必然也调用了 swtch 函数。所以之前的 swtch 函数会被恢复，并返回到进程 P2 所在的系统调用或者中断处理程序中
- 不论是系统调用也好中断处理程序也好，在从用户空间进入到内核空间时会保存用户寄存器到 trapframe 对象。所以当内核程序执行完成之后，trapframe 中的用户寄存器会被恢复
- 最后用户进程 P2 就恢复运行了

。