操作系统实验三

——UCORE lab5、6

班级：硬件一班 姓名：于华宇 学号：171491120

### 实验目的

###### lab5

（1）了解第一个用户进程创建过程

（2）了解系统调用框架的实现机制

（3）了解ucore如何实现系统调用sys\_fork/sys\_exec/sys\_exit/sys\_wait来进行进程管理

###### lab6

（1）理解操作系统的调度管理机制

（2）熟悉 ucore的系统调度器框架,以及缺省的Round-Robin调度算法

（3）基于调度器框架实现一个(Stride Scheduling)调度算法来替换缺省的调度算法

### 二、实验内容

**lab5：**实验5将创建用户进:程,让用户进程在用户态执行,且在需要ucore支持时,可通过系统调用来让ucore提供服务。为此需要构造出第一个用户进程,并通过系统调用sys\_fork/sys\_exec/sys\_exit/sys\_wait来支持运行不同的应用程序,完成对用户进程的执行过程的基本管理。

**lab6：**实验五完成了用户进程的管理,可在用户态运行多个进程。但到目前为止,采用的调度策略是很简单的FIFO调度策略。本次实验,主要是熟悉ucore的系统调度器框架,以及基于此框架的Round-Robin(RR)调度算法。然后参考RR调度算法的实现,完成Stride Scheduling调度算法。

lab5

###### 练习1 加载应用程序并执行(需要编码)

do\_execv函数调用load\_icode(位于kern/process/proc.c中)来加载并解析一个处于内存中的ELF执行文件格式的应用程序,建立相应的用户内存空间来放置应用程序的代码段、数据段等,且要设置好proc\_struct结构中的成员变量trapframe中的内容,确保在执行此进程后,能够从应用程序设定的起始执行地址开始执行。

**（经查询资料）**

* ELF文件：二进制文件，可执行文件。由ELF头，程序头表，节和节头表组成。

do\_execv函数会先回收自身进程所占用户空间，然后调用load\_icode，用新的程序覆盖内存空间，形成一个执行新程序的新进程。

**load\_icode()函数代码如下：**

**/\* load\_icode - load the content of binary program(ELF format) as the new content of current process**

**\* @binary: the memory addr of the content of binary program**

**\* @size: the size of the content of binary program**

**\*/**

**static int**

**load\_icode(unsigned char \*binary, size\_t size) {**

**if (current->mm != NULL) {**

**panic("load\_icode: current->mm must be empty.\n");**

**}**

**int ret = -E\_NO\_MEM;**

**struct mm\_struct \*mm;**

**//(1) create a new mm for current process**

**//调用mm\_create函数为申请内存管理的数据结构mm申请内存空间，并对mm进行初始化**

**if ((mm = mm\_create()) == NULL) {**

**goto bad\_mm;**

**}**

**//(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT**

**/\*调用setup\_pgdir函数申请一个页目录所需要的一个页大小的内存空间，并把ucore内核虚空间映射的内核页表的内容拷贝到这个新目录表中，最后mm->pgdir指向此页目录表，这就是进程新的页目录表，且能够正确映射内核\*/**

**if (setup\_pgdir(mm) != 0) {**

**goto bad\_pgdir\_cleanup\_mm;**

**}**

**//(3) copy TEXT/DATA section, build BSS parts in binary to memory space of process**

**/\*根据应用程序的执行码的起始位置来解析此ELF格式的执行程序，并调用mm\_map函数根据ELF格式的执行程序说明各个段的起始位置和大小建立对应的vma结构，并把vma插入到mm结构中，从而表明了用户进程的合法用户态虚拟地址空间\*/**

**struct Page \*page;**

**//(3.1) get the file header of the bianry program (ELF format)**

**struct elfhdr \*elf = (struct elfhdr \*)binary;**

**//(3.2) get the entry of the program section headers of the bianry program (ELF format)**

**struct proghdr \*ph = (struct proghdr \*)(binary + elf->e\_phoff);**

**//(3.3) This program is valid?**

**if (elf->e\_magic != ELF\_MAGIC) {**

**ret = -E\_INVAL\_ELF;**

**goto bad\_elf\_cleanup\_pgdir;**

**}**

**uint32\_t vm\_flags, perm;**

**struct proghdr \*ph\_end = ph + elf->e\_phnum;**

**for (; ph < ph\_end; ph ++) {**

**//(3.4) find every program section headers**

**if (ph->p\_type != ELF\_PT\_LOAD) {**

**continue ;**

**}**

**if (ph->p\_filesz > ph->p\_memsz) {**

**ret = -E\_INVAL\_ELF;**

**goto bad\_cleanup\_mmap;**

**}**

**if (ph->p\_filesz == 0) {**

**continue ;**

**}**

**//(3.5) call mm\_map fun to setup the new vma ( ph->p\_va, ph->p\_memsz)**

**vm\_flags = 0, perm = PTE\_U;**

**if (ph->p\_flags & ELF\_PF\_X) vm\_flags |= VM\_EXEC;**

**if (ph->p\_flags & ELF\_PF\_W) vm\_flags |= VM\_WRITE;**

**if (ph->p\_flags & ELF\_PF\_R) vm\_flags |= VM\_READ;**

**if (vm\_flags & VM\_WRITE) perm |= PTE\_W;**

**if ((ret = mm\_map(mm, ph->p\_va, ph->p\_memsz, vm\_flags, NULL)) != 0) {**

**goto bad\_cleanup\_mmap;**

**}**

**unsigned char \*from = binary + ph->p\_offset;**

**size\_t off, size;**

**uintptr\_t start = ph->p\_va, end, la = ROUNDDOWN(start, PGSIZE);**

**ret = -E\_NO\_MEM;**

**//(3.6) alloc memory, and copy the contents of every program section (from, from+end) to process's memory (la, la+end)**

**/\*调用根据执行程序各个段的大小分配的物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了\*/**

**end = ph->p\_va + ph->p\_filesz;**

**//(3.6.1) copy TEXT/DATA section of bianry program**

**while (start < end) {**

**if ((page = pgdir\_alloc\_page(mm->pgdir, la, perm)) == NULL) {**

**goto bad\_cleanup\_mmap;**

**}**

**off = start - la, size = PGSIZE - off, la += PGSIZE;**

**if (end < la) {**

**size -= la - end;**

**}**

**memcpy(page2kva(page) + off, from, size);**

**start += size, from += size;**

**}**

**//(3.6.2) build BSS section of binary program**

**end = ph->p\_va + ph->p\_memsz;**

**if (start < la) {**

**/\* ph->p\_memsz == ph->p\_filesz \*/**

**if (start == end) {**

**continue ;**

**}**

**off = start + PGSIZE - la, size = PGSIZE - off;**

**if (end < la) {**

**size -= la - end;**

**}**

**memset(page2kva(page) + off, 0, size);**

**start += size;**

**assert((end < la && start == end) || (end >= la && start == la));**

**}**

**while (start < end) {**

**if ((page = pgdir\_alloc\_page(mm->pgdir, la, perm)) == NULL) {**

**goto bad\_cleanup\_mmap;**

**}**

**off = start - la, size = PGSIZE - off, la += PGSIZE;**

**if (end < la) {**

**size -= la - end;**

**}**

**memset(page2kva(page) + off, 0, size);**

**start += size;**

**}**

**}**

**//(4) build user stack memory**

**/\*需要给用户进程设置用户栈，为此调用mm\_mmap函数建立用户栈的vma结构，明确用户栈的位置在用户虚空间的顶端，大小为256个页，即1MB，并分配一定数量的物理内存且建立好栈的虚地址与物理地址映射关系\*/**

**vm\_flags = VM\_READ | VM\_WRITE | VM\_STACK;**

**if ((ret = mm\_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm\_flags, NULL)) != 0) {**

**goto bad\_cleanup\_mmap;**

**}**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-PGSIZE , PTE\_USER) != NULL);**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-2\*PGSIZE , PTE\_USER) != NULL);**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-3\*PGSIZE , PTE\_USER) != NULL);**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-4\*PGSIZE , PTE\_USER) != NULL);**

**//(5) set current process's mm, sr3, and set CR3 reg = physical addr of Page Directory**

**/\*至此，进程内的内存管理vma和mm数据结构已经建立完成，于是把mm->pgdir赋值到cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被hello的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好\*/**

**mm\_count\_inc(mm);**

**current->mm = mm;**

**current->cr3 = PADDR(mm->pgdir);**

**lcr3(PADDR(mm->pgdir));**

**//(6) setup trapframe for user environment**

**/\*先清空进程中的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让cpu转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断。\*/**

**struct trapframe \*tf = current->tf;**

**memset(tf, 0, sizeof(struct trapframe));//清空中断帧**

**/\* LAB5:EXERCISE1 YOUR CODE**

**\* should set tf\_cs,tf\_ds,tf\_es,tf\_ss,tf\_esp,tf\_eip,tf\_eflags**

**\* NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from kernel. So**

**\* tf\_cs should be USER\_CS segment (see memlayout.h)**

**\* tf\_ds=tf\_es=tf\_ss should be USER\_DS segment**

**\* tf\_esp should be the top addr of user stack (USTACKTOP)**

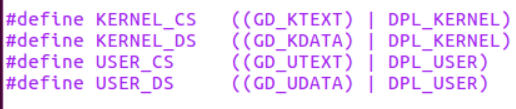
**\* tf\_eip should be the entry point of this binary program (elf->e\_entry)**

**\* tf\_eflags should be set to enable computer to produce Interrupt**

**\*/**

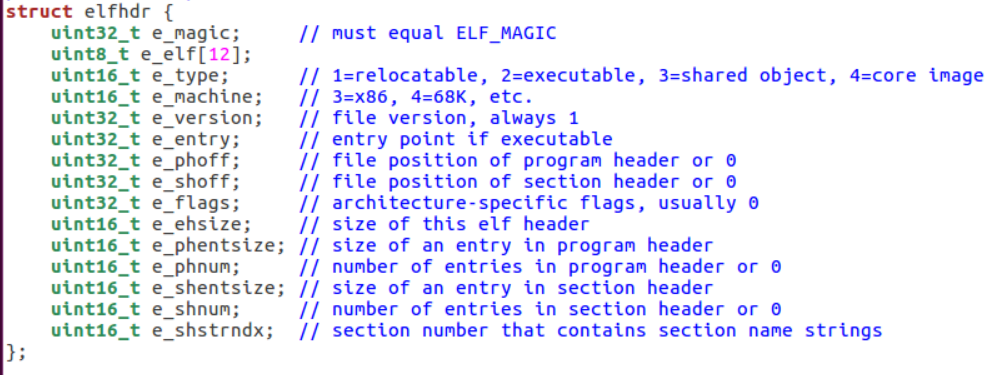
**tf->tf\_cs = USER\_CS;//将tf\_cs设置为用户态，这个定义在memlayout.h，有一个宏定义已经定义了用户态和内核态。**

**tf->tf\_ds = tf->tf\_es = tf->tf\_ss = USER\_DS; // tf->tf\_ds = tf->tf\_es = tf->tf\_ss也需要设置为用户态，定义位置同上**

****

**tf->tf\_esp = USTACKTOP;//需要将esp设置为用户栈的栈顶，直接使用之前建立用户栈时的参数USTACKTOP即可**

**tf->tf\_eip = elf->e\_entry;//eip是程序的入口，elf类的e\_entry函数直接声明了**

****

**tf->tf\_eflags = FL\_IF;//FL\_IF打开中断**

**/\*tf是一个中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。\*/**

**ret = 0;**

**out:**

**return ret;**

**bad\_cleanup\_mmap:**

**exit\_mmap(mm);**

**bad\_elf\_cleanup\_pgdir:**

**put\_pgdir(mm);**

**bad\_pgdir\_cleanup\_mm:**

**mm\_destroy(mm);**

**bad\_mm:**

**goto out;**

**}**

###### 练习2: 父进程复制自己的内存空间给子进程(需要编码)

创建子进程的函数do\_fork在执行中将拷贝当前进程(即父进程)的用户内存地址空间中的合法内容到新进程中(子进程),完成内存资源的复制。具体是通过copy\_range函数(位于kern/mm/pmm.c中)实现的,请补充copy\_range的实现,确保能够正确执行。

copy\_range部分代码如下：

**/\* copy\_range - copy content of memory (start, end) of one process A to another process B**

**\* @to: the addr of process B's Page Directory**

**\* @from: the addr of process A's Page Directory**

**\* @share: flags to indicate to dup OR share. We just use dup method, so it didn't be used.**

**\***

**\* CALL GRAPH: copy\_mm-->dup\_mmap-->copy\_range**

**//复制是先在do\_fork函数中调用copy\_mm，copy\_mm调用dup\_mmap，dup\_mmap调用copy\_range**

**\*/**

**/\* LAB5:EXERCISE2 YOUR CODE**

**\* replicate content of page to npage, build the map of phy addr of nage with the linear addr start**

**\***

**\* Some Useful MACROs and DEFINEs, you can use them in below implementation.**

**\* MACROs or Functions:**

**\* page2kva(struct Page \*page): return the kernel vritual addr of memory which page managed (SEE pmm.h)**

**\* page\_insert: build the map of phy addr of an Page with the linear addr la**

**\* memcpy: typical memory copy function**

**\***

**\* (1) find src\_kvaddr: the kernel virtual address of page**

**\* (2) find dst\_kvaddr: the kernel virtual address of npage**

**\* (3) memory copy from src\_kvaddr to dst\_kvaddr, size is PGSIZE**

**\* (4) build the map of phy addr of nage with the linear addr start**

**\*/**

**void \*kva\_src = page2kva(page);//获取老页表的值**

**void \*kva\_dst =page2kva(npage);//获取新页表的值**

**memcpy(kva\_dst,kva\_src,PGSIZE);//复制操作**

**ret =page\_insert(to,npage,start,perm);//建立子进程页地址起始位置与物理地址的映射关系，perm是权限**

**assert(ret == 0);**

**}**

**start += PGSIZE;**

**} while (start != 0 && start < end);**

**return 0;**

**}**

###### 练习3: 阅读分析源代码,理解进程执行 fork/exec/wait/exit 的实现,以及系统调用的实现(不需要编码)

请在实验报告中简要说明你对fork/exec/wait/exit函数的分析。

并回答如下问题:

1、请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的?

2、请给出ucore中一个用户态进程的执行状态生命周期图(包执行状态,执行状态之间的变换关系,以及产生变换的事件或函数调用)。(字符方式画即可)

3、执行:make grade。如果所显示的应用程序检测都输出ok,则基本正确。(使用的是qemu-1.0.1)

**do\_fork：**

**/\* do\_fork - parent process for a new child process**

**\* @clone\_flags: used to guide how to clone the child process**

**\* @stack: the parent's user stack pointer. if stack==0, It means to fork a kernel thread(内核线程).**

**\* @tf: the trapframe info, which will be copied to child process's proc->tf**

**\*/**

**int do\_fork(uint32\_t clone\_flags, uintptr\_t stack, struct trapframe \*tf) {**

**int ret = -E\_NO\_FREE\_PROC;**

**struct proc\_struct \*proc; //声明新进程，若进程数不超过最大值则申请成功**

**if (nr\_process >= MAX\_PROCESS) {**

**goto fork\_out;**

**}**

**ret = -E\_NO\_MEM; //由于内存空间不足而分配失败**

**//LAB4:EXERCISE2 YOUR CODE**

**/\***

**\* Some Useful MACROs（宏）, Functions and DEFINEs, you can use them in below implementation.**

**\* MACROs or Functions:**

**\* alloc\_proc: create a proc struct and init fields (lab4:exercise1)**

**\* setup\_kstack: alloc pages with size KSTACKPAGE as process kernel stack**

**\* copy\_mm: process "proc" duplicate OR share process "current"'s mm according clone\_flags**

**\* if clone\_flags & CLONE\_VM, then "share" ; else "duplicate"**

**\* copy\_thread: setup the trapframe on the process's kernel stack top and**

**\* setup the kernel entry point and stack of process**

**\* hash\_proc: add proc into proc hash\_list**

**\* get\_pid: alloc a unique pid for process**

**\* wakeup\_proc: set proc->state = PROC\_RUNNABLE**

**\* VARIABLES:**

**\* proc\_list: the process set's list**

**\* nr\_process: the number of process set**

**\*/**

**// 1. call alloc\_proc to allocate a proc\_struct**

**// 2. call setup\_kstack to allocate a kernel stack for child process**

**// 3. call copy\_mm to dup OR share mm according clone\_flag**

**// 4. call copy\_thread to setup tf & context in proc\_struct**

**// 5. insert proc\_struct into hash\_list && proc\_list**

**// 6. call wakeup\_proc to make the new child process RUNNABLE**

**// 7. set ret vaule using child proc's pid**

**if ((proc = alloc\_proc()) == NULL) { //分配进程控制块**

**goto fork\_out;**

**}**

**proc->parent = current; //设置父进程为当前进程**

**assert(current->wait\_state == 0); //assert()判断参数是否为零，保证当前进程在等待状态**

**if (setup\_kstack(proc) != 0) { //建立内核栈**

**goto bad\_fork\_cleanup\_proc;**

**}**

**if (copy\_mm(clone\_flags, proc) != 0) { //根据clone\_flags复制或共享内存空间**

**goto bad\_fork\_cleanup\_kstack;**

**}**

**copy\_thread(proc, stack, tf); //根据栈和中断帧设置进程的中断帧和上下文**

**bool intr\_flag;**

**local\_intr\_save(intr\_flag); //获得进程pid，将进程控制块放入哈希表和进程序列中。**

**{**

**proc->pid = get\_pid();**

**hash\_proc(proc);**

**set\_links(proc);**

**}**

**local\_intr\_restore(intr\_flag);**

**wakeup\_proc(proc); //唤醒子进程**

**ret = proc->pid; //返回子进程的pid**

**fork\_out:**

**return ret;**

**bad\_fork\_cleanup\_kstack:**

**put\_kstack(proc);**

**bad\_fork\_cleanup\_proc:**

**kfree(proc);**

**goto fork\_out;**

**}**

**do\_fork：**

首先分配并初始化进程控制块，分配初始化内核栈，根据clone\_flag标志复制或共享进程内存管理结构，设置进程在内核正常运行和调度所需要的中断帧和执行上下文。获取子进程的pid，将进程控制块放入哈希表和进程序列表中，唤醒子进程，把进程状态设置为就绪态。返回值设置为子进程的pid。

**do\_exit:**

**// do\_exit - called by sys\_exit**

**// 1. call exit\_mmap & put\_pgdir & mm\_destroy to free the almost all memory space of process**

**// 2. set process' state as PROC\_ZOMBIE, then call wakeup\_proc(parent) to ask parent reclaim itself.**

**// 3. call scheduler to switch to other process**

**int do\_exit(int error\_code) {**

**if (current == idleproc) { //判断当前进程是否是idleproc或initproc，若是则直接退出，不需要回**

**panic("idleproc exit.\n"); //收资源**

**}**

**if (current == initproc) {**

**panic("initproc exit.\n");**

**}**

**struct mm\_struct \*mm = current->mm;**

**if (mm != NULL) { //若当前内存不为空，则回收内存**

**lcr3(boot\_cr3); //从用户模式切换回内核模式**

**if (mm\_count\_dec(mm) == 0) {**

**exit\_mmap(mm);**

**put\_pgdir(mm);**

**mm\_destroy(mm);**

**}**

**current->mm = NULL; //将内存指向空，回收内存完成**

**}**

**current->state = PROC\_ZOMBIE; //将当前进程状态设置为僵尸态，等待父进程回收**

**current->exit\_code = error\_code; //当前程序的退出**

**bool intr\_flag;**

**struct proc\_struct \*proc;**

**local\_intr\_save(intr\_flag);**

**{**

**proc = current->parent;**

**if (proc->wait\_state == WT\_CHILD) { //如果父进程在等待子进程，唤醒父进程**

**wakeup\_proc(proc);**

**}**

**while (current->cptr != NULL) {**

**//如果当前进程还有子进程（孤儿进程），则需要把这些子进程的父进程指针设置为内核线程initproc，如果某个子进程的执行状态是僵尸态，则需要唤醒initproc来完成对此子进程的回收**

**proc = current->cptr;**

**current->cptr = proc->optr;**

**proc->yptr = NULL;**

**if ((proc->optr = initproc->cptr) != NULL) {**

**initproc->cptr->yptr = proc;**

**}**

**proc->parent = initproc;**

**initproc->cptr = proc;**

**if (proc->state == PROC\_ZOMBIE) {**

**if (initproc->wait\_state == WT\_CHILD) {**

**wakeup\_proc(initproc);**

**}**

**}**

**}**

**}**

**local\_intr\_restore(intr\_flag);**

**schedule(); //调度其他进程运行**

**panic("do\_exit will not return!! %d.\n", current->pid);**

**}**

**do\_exit:**

先判断进程是否为用户进程，如果是，则开始回收此用户进程所占用的内存空间。设置当前进程状态为僵尸态，退出码为error\_code。此时这个进程已经无法再被调度了，只能等待父进程完成最后回收。若当前进程父进程处于等待子进程的状态，可以唤醒父进程帮助子进程进行最后资源回收。如果当前进程还有子进程，则将子进程的父进程指针指向内核线程init，且各个子进程指针需要插入到init子进程链表中，如果某个子进程的执行状态是僵尸态，则要唤醒init对子进程进行最后回收。最后执行schedule函数调度其他进程执行。

**do\_execve:**

**// do\_execve - call exit\_mmap(mm)&put\_pgdir(mm) to reclaim memory space of current process**

**// - call load\_icode to setup new memory space accroding binary prog.**

**int do\_execve(const char \*name, size\_t len, unsigned char \*binary, size\_t size) {**

**struct mm\_struct \*mm = current->mm;**

**if (!user\_mem\_check(mm, (uintptr\_t)name, len, 0)) { //查看用户内存空间，若为空或有其他异常**

**return -E\_INVAL; //返回无效参数**

**}**

**if (len > PROC\_NAME\_LEN) { //若长度超过进程名规定长度，长度改为规定长度**

**len = PROC\_NAME\_LEN;**

**}**

**char local\_name[PROC\_NAME\_LEN + 1]; //命名**

**memset(local\_name, 0, sizeof(local\_name)); //设置内存空间为特定值**

**memcpy(local\_name, name, len); //复制内存空间local\_name中的len个字节内容到//name中**

**if (mm != NULL) { //如果当前内存空间不空**

**lcr3(boot\_cr3); //从用户模式切换回内核模式**

**if (mm\_count\_dec(mm) == 0) { //当共享内存空间的进程为零时**

**exit\_mmap(mm);**

**put\_pgdir(mm); //释放PDT的内存空间**

**mm\_destroy(mm); //释放内存**

**}**

**current->mm = NULL;**

**}**

**int ret;**

**if ((ret = load\_icode(binary, size)) != 0) { //加载进程，若失败则退出**

**goto execve\_exit;**

**}**

**set\_proc\_name(current, local\_name);**

**return 0;**

**execve\_exit:**

**do\_exit(ret);**

**panic("already exit: %e.\n", ret);**

**}**

**do\_execve:**

首先准备清空用户态内存，若内存不空，将内存中数据复制到内核中，加载新进程应用程序执行码到当前进程新创建的用户态内存中。设置进程名。

**do\_wait:**

**// do\_wait - wait one OR any children with PROC\_ZOMBIE state, and free memory space of kernel stack**

**// - proc struct of this child.**

**// NOTE: only after do\_wait function, all resources of the child proces are free.**

**int do\_wait(int pid, int \*code\_store) {**

**struct mm\_struct \*mm = current->mm;**

**if (code\_store != NULL) { //检测参数是否有效**

**if (!user\_mem\_check(mm, (uintptr\_t)code\_store, sizeof(int), 1)) {**

**return -E\_INVAL;**

**}**

**}**

**struct proc\_struct \*proc;**

**bool intr\_flag, haskid;**

**repeat:**

**haskid = 0; //先记录已有孩子为0**

**if (pid != 0) {**

**proc = find\_proc(pid); //若pid不为0，从进程哈希表中找到子进程**

**if (proc != NULL && proc->parent == current) { //子若进程不为空，且当前进程为其父进程，//孩子数加一**

**haskid = 1;**

**if (proc->state == PROC\_ZOMBIE) {**

**goto found;**

**}**

**}**

**}**

**else {**

**proc = current->cptr; //若pid==0，则找其他子进程**

**for (; proc != NULL; proc = proc->optr) {**

**haskid = 1;**

**if (proc->state == PROC\_ZOMBIE) { //直到找到僵尸态子进程，进行回收**

**goto found;**

**}**

**}**

**}**

**if (haskid) { //如果当前有子进程不是僵尸态**

**current->state = PROC\_SLEEPING; //设置当前进程状态为等待状态**

**current->wait\_state = WT\_CHILD; //等待状态为等待子进程**

**schedule(); //进行调度其他进程执行**

**if (current->flags & PF\_EXITING) {**

**do\_exit(-E\_KILLED);**

**}**

**goto repeat;**

**}**

**return -E\_BAD\_PROC;**

**found: //子进程处于僵尸态执行**

**if (proc == idleproc || proc == initproc) { //检验进程是否为内核进程**

**panic("wait idleproc or initproc.\n");**

**}**

**if (code\_store != NULL) {**

**\*code\_store = proc->exit\_code;**

**}**

**local\_intr\_save(intr\_flag); //将进程从哈希表和列表中移除**

**{**

**unhash\_proc(proc);**

**remove\_links(proc);**

**}**

**local\_intr\_restore(intr\_flag);**

**put\_kstack(proc);**

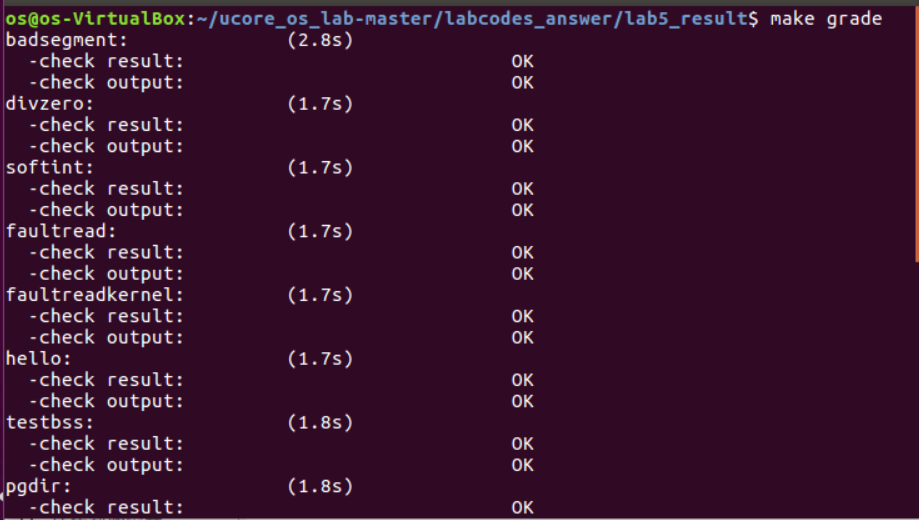
**kfree(proc); //释放子进程**

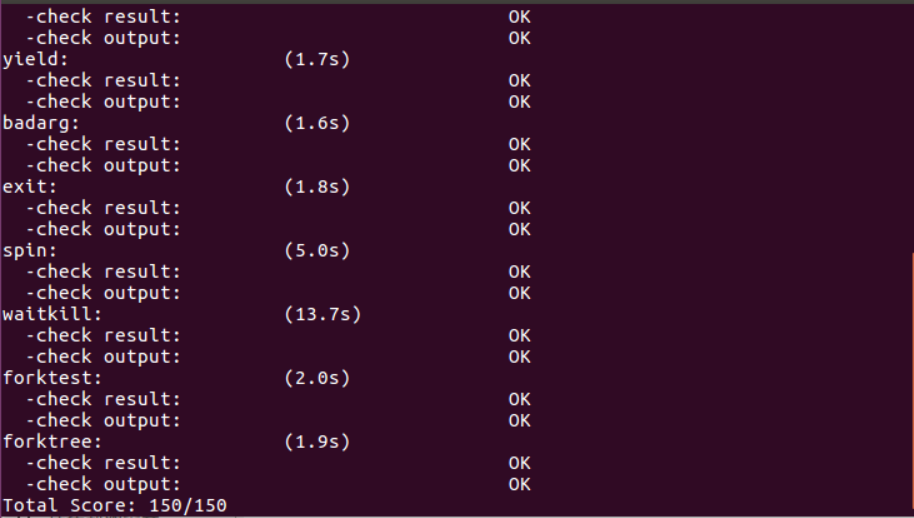
**return 0;**

**}**

**do\_wait:**

完成的是父进程最终释放子进程的过程。如果pid不为零，则找一个处于僵尸态的子进程进行回收。若pid为零，则找其他子进程，直到找到僵尸态子进程，将其回收。如果当前有子进程不是僵尸态，则父进程设为等待子进程的睡眠态，调度其他进程执行，直到唤醒或被杀死终止。释放僵尸态子进程时，检验其是否为内核进程，将code\_store设为退出代码，从哈希表和进程序列中移除子进程，并释放内核栈和进程控制块。子进程资源回收完毕。

在用**make grade** 运行的时候，会有错误。注释掉tools/grade.sh的221行到233行后，不会报错。



执行状态生命周期图

lab6

练习1: 使用Round Robin 调度算法(不需要编码**)**

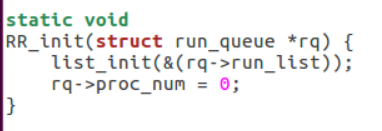
分析了解lab6采用RR调度算法后的执行过程。执行make grade,大部分测试用例应该通过。但执行priority.c应该过不去。请理解并分析sched\_class中各个函数指针的用法,并接合Round Robin调度算法描ucore的调度执行过程

请在简要说明如何设计实现”多级反馈队列调度算法“,给出概要设计。

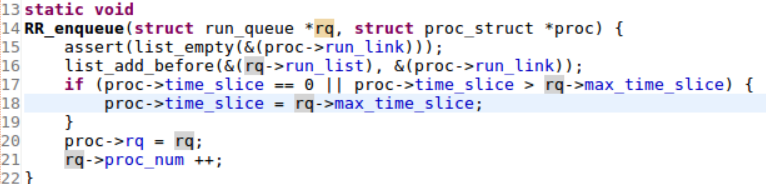
Round Robin调度算法（RR，轮转调度）的调度思想是让所有就绪态的进程分时轮流使用CPU时间。调度器维护当前就绪进程的有序运行队列。当前进程的时间片用完之后，调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进行调度。

RR算法分析：

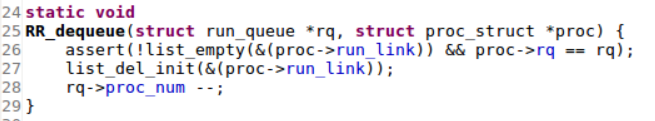
1、先初始化一个列表，初始化的节点前一个后一个都先指向自身，进程数也设为零。



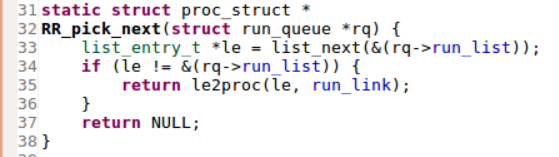
2、进程入队操作。先判断序列是否为空，在进程序列最后添加节点。若进程时间片为零，或进程时间片大于队列最大时间片，则将该进程时间片设置为队列中最大时间片。进程进入到运行序列后，队列中的进程数目加一。



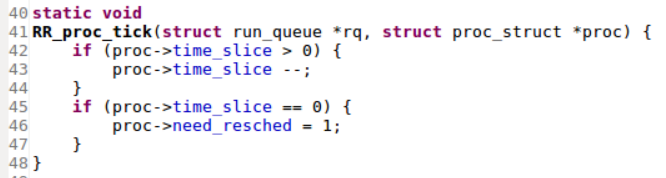
3、出队时判断进程序列是否为空，将进程从队列中移除，运行队列进程数减一。



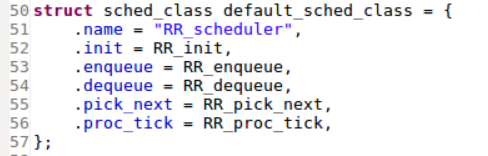
4、从队尾选一个进程去执行。如果选不出处于就绪态的进程，则返回NULL，把执行权交给内核线程idle，idle的功能是不断调用schedule，直到系统出现一个可以执行的进程。



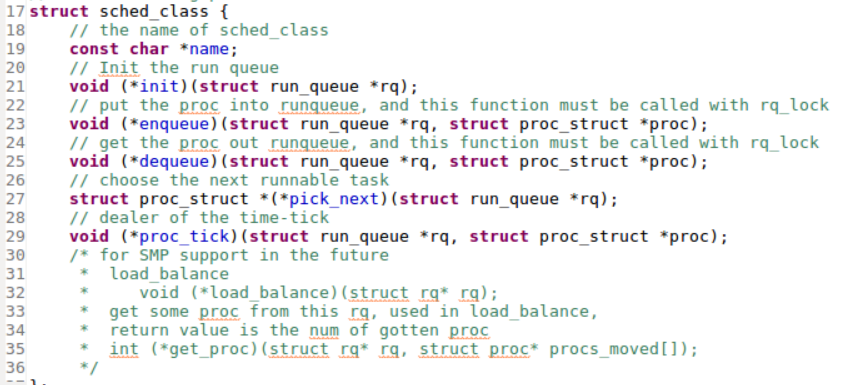
5、产生时钟中断时，会触发函数调用。每次产生时钟中断，时间片减一，一旦时间片用完，将进程控制块的need\_resched置为1，代表它放弃对cpu的占有，需要将别的进程调度进来执行，当前进程需要等待。



6、在schedule初始化时，需要填写初始化信息。填上以上函数，系统就会按照这个方式去执行。



**sched\_class:**

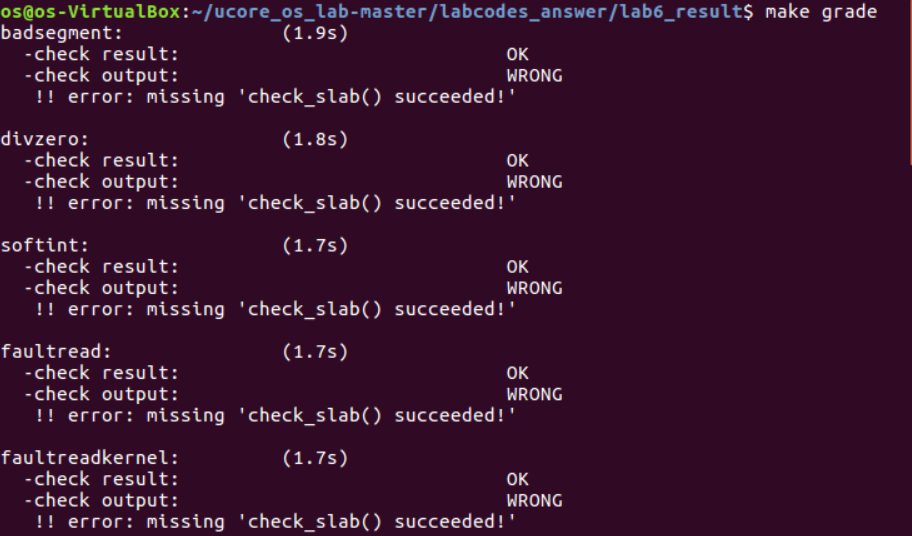


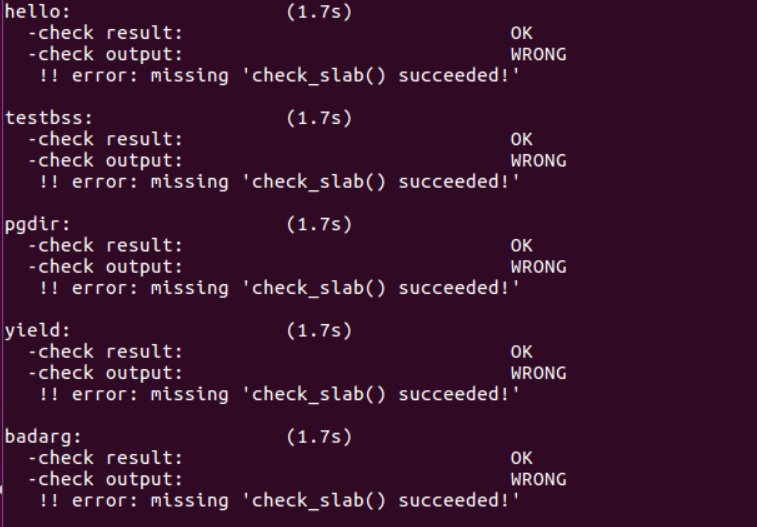
sched\_class 中运行需要初始化一些函数，赋给sched\_class，如上述RR算法，将相应的函数写出，赋给sched\_class，形成RR\_schedule

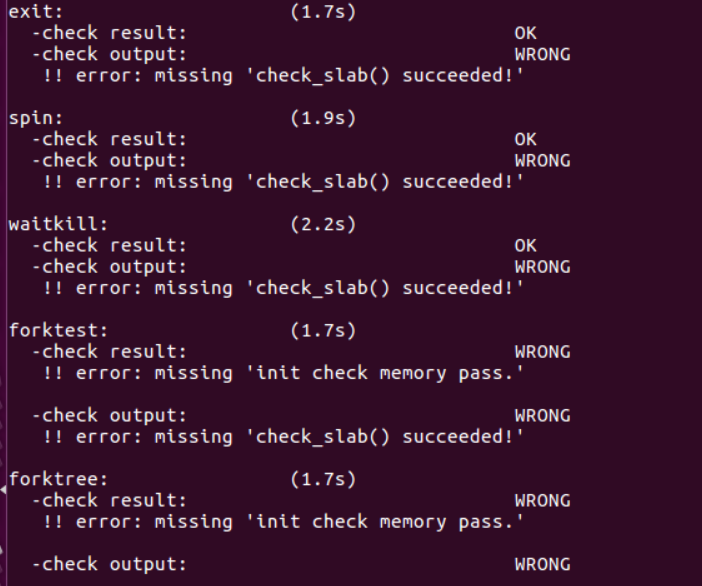
多级调度：

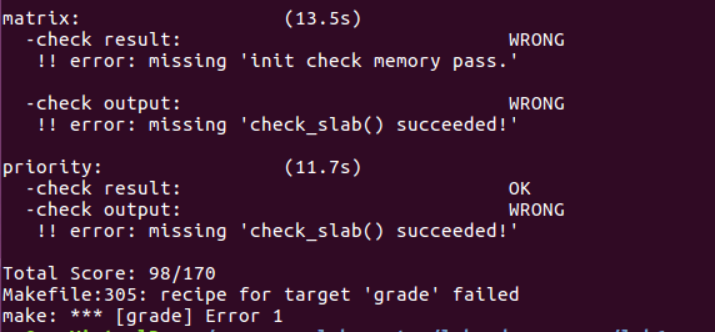
需要根据优先级个数设置进程队列个数。高优先级进程时间片耗尽后，加入低优先级队列中，排队等待下一次运行。

**make grade**

****

****

****

****

###### 练习2: 实现 Stride Scheduling 调度算法（需要编码）---经资料查询

首先需要换掉RR调度器的实现，即用default\_sched\_stride\_c覆盖default\_sched.c。然后根据此文件和后续文档对Stride度器 的相关描述，完成Stride调度算法的实现。执行：make grade。如果所显示的应用程序检测都输出ok，则基本正确。如果只是priority.c过不去，可执行 make runpriority 命令来单独调试它。大致执行结果可看附录。（ 使用的是 qemu-1.0.1 ）。 请在实验报告中简要说明你的设计实现过程。

**思路：**

考察 round-robin 调度器，在假设所有进程都充分使用了其拥有的 CPU 时间资源的情况下，所有进程得到的 CPU 时间应该是相等的。但是有时候我们希望调度器能够更智能地为每个进程分配合理的 CPU 资源。假设我们为不同的进程分配不同的 优先级，则我们有可能希望每个进程得到的时间资源与他们的优先级成正比关系。Stride调度是基于这种想法的一个较为典型和简单的算法。

除了简单易于实现以外，它还有如下的特点：

**可控性**：如我们之前所希望的，可以证明 Stride Scheduling对进程的调度次数正比于其优先级。

**确定性**：在不考虑计时器事件的情况下，整个调度机制都是可预知和重现的。该算法的基本思想可以考虑如下：

1. 为每个runnable的进程设置一个当前状态stride，表示该进程当前的调度权。另外定义其对应的pass值，表示对应进 程在调度后，stride 需要进行的累加值。

2. 每次需要调度时，从当前 runnable 态的进程中选择 stride最小的进程调度。

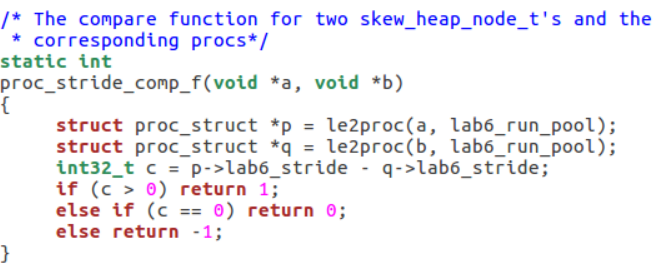
3. 对于获得调度的进程P，将对应的stride加上其对应的步长pass（只与进程的优先权有关系）。

4. 在一段固定的时间之后，回到 2.步骤，重新调度当前stride最小的进程。 可以证明，如果令 P.pass =BigStride / P.priority 其中 P.priority 表示进程的优先权（大于 1），而 BigStride 表示一 个预先定义的大常数，则该调度方案为每个进程分配的时间将与其优先级成正比。

在Stride Scheduling中，优先级与步长成反比。因为谁的步长越小，调度上优先级就会越高。数据结构上，使用斜堆（一种二叉树，根节点最小）。

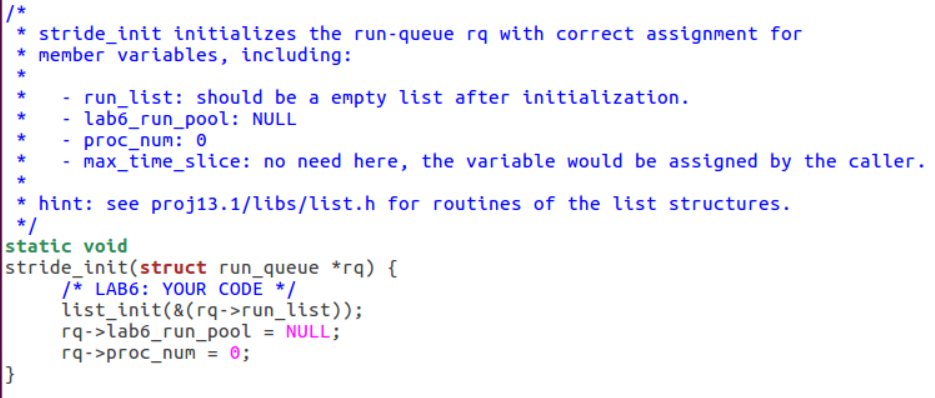
**实现：**

1、实现比较函数，找出stride最小的进程去执行

****

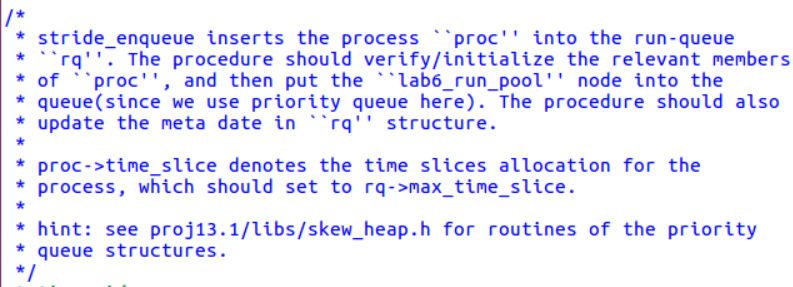
比较stride值，并根据返回值调整斜堆。

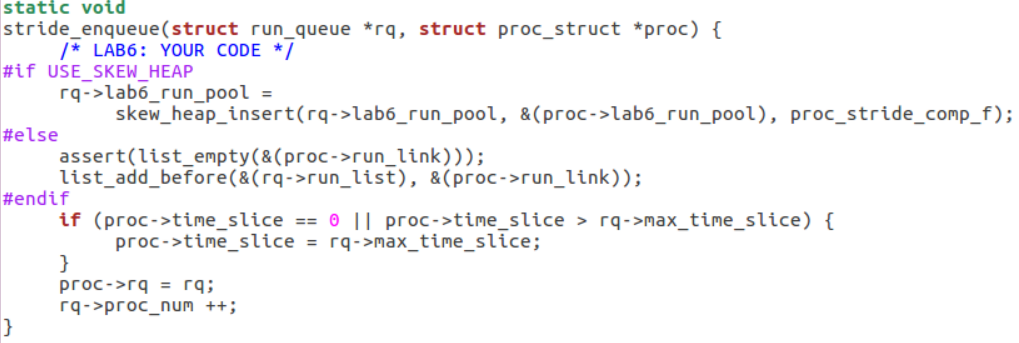
2、初始化



初始化进程序列，斜堆和进程数目。

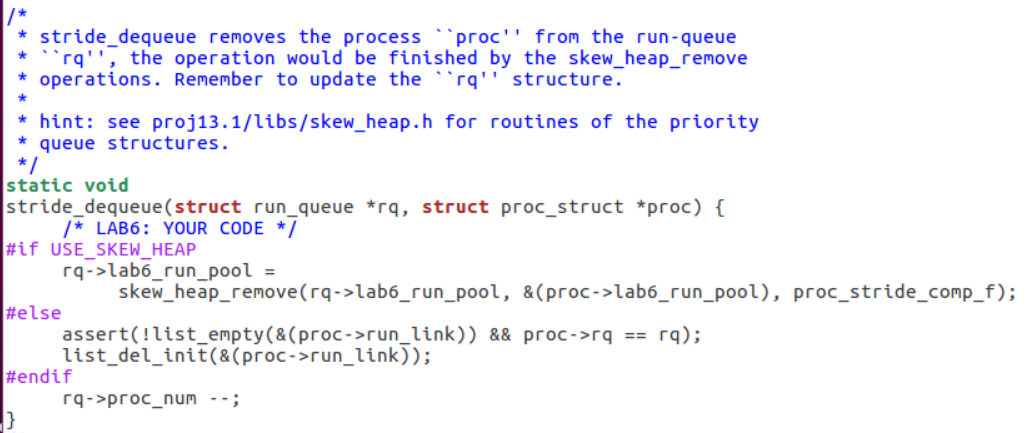
3、入队





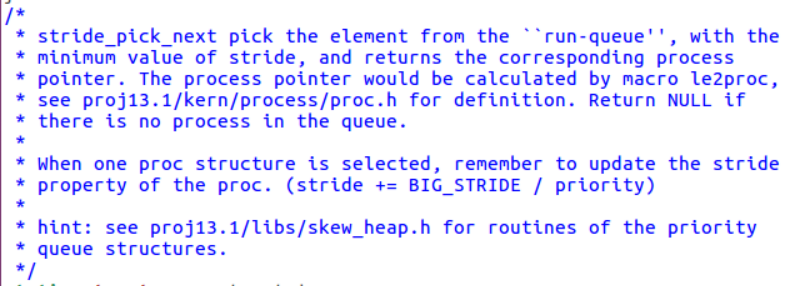
如果使用斜堆数据结构，调用skew\_heap\_insert 函数插入进程，并对斜堆自身进行排序。将进程插入到进程序列中。判断时间片是否有问题，若有问题，设置时间片。

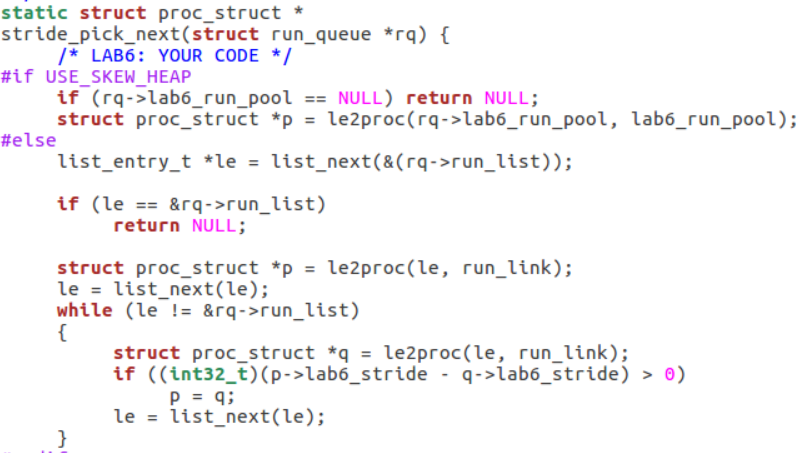
4、出队

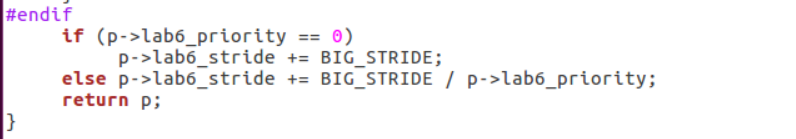


用斜堆结构，则使用skew\_heap\_remove函数从斜堆中将进程移出，判断进程队列是否为空，将对应进程从进程队列中移出，最后将进程数减一，出队完成。

5、选择下一个进程

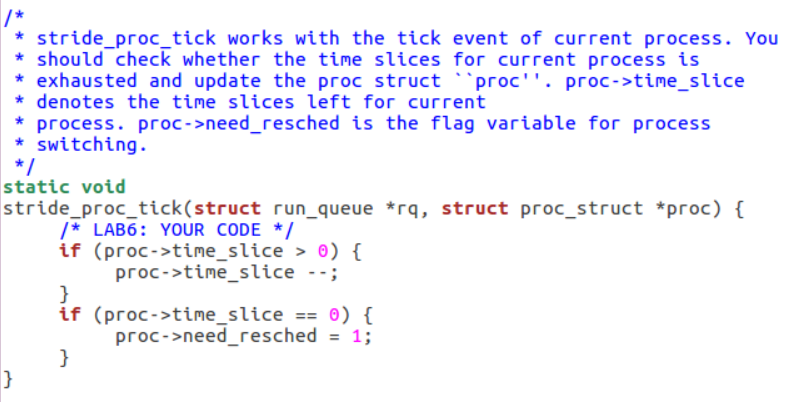






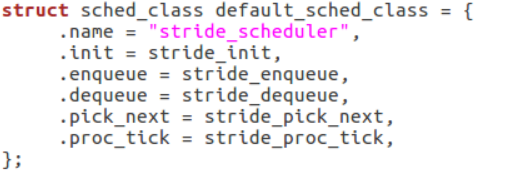
从进程队列中，挑选出stride最小的进程，返回其指针。

6、时钟中断函数



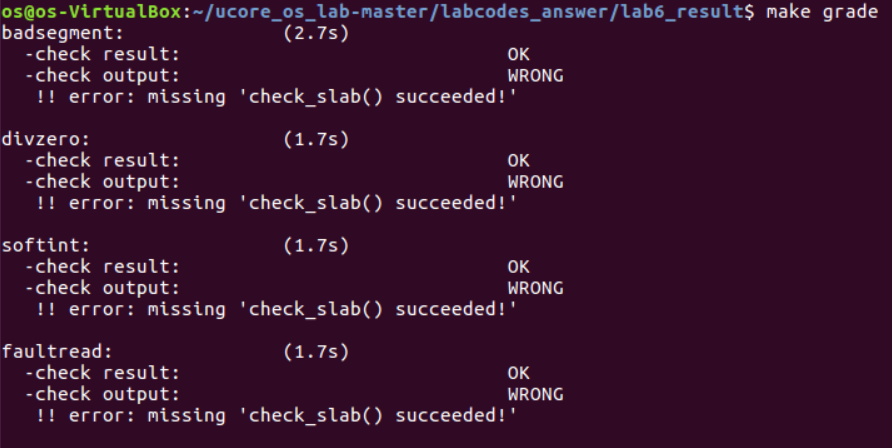
检查时间片是否为零，若不为零，则减一，若为零，则重新分配时间片。

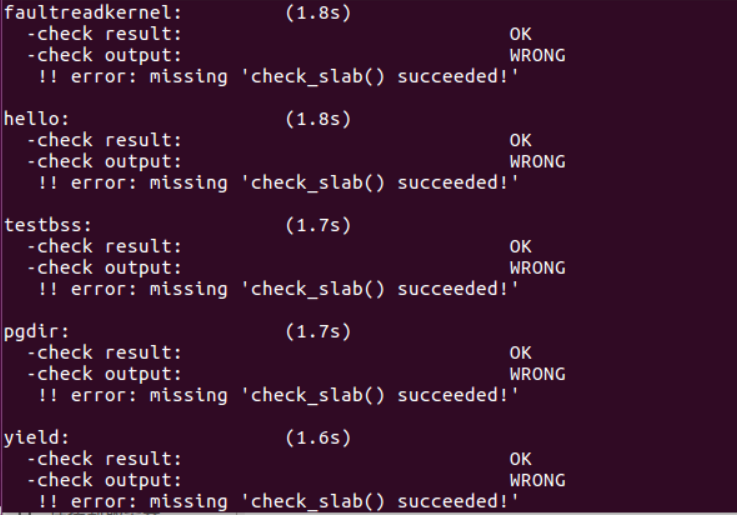
7、stride\_scheduler的初始化

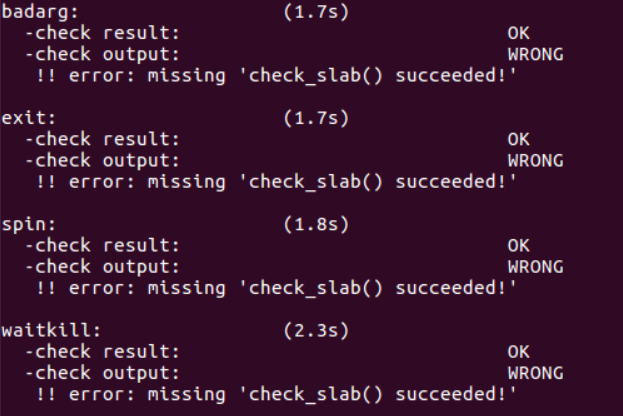


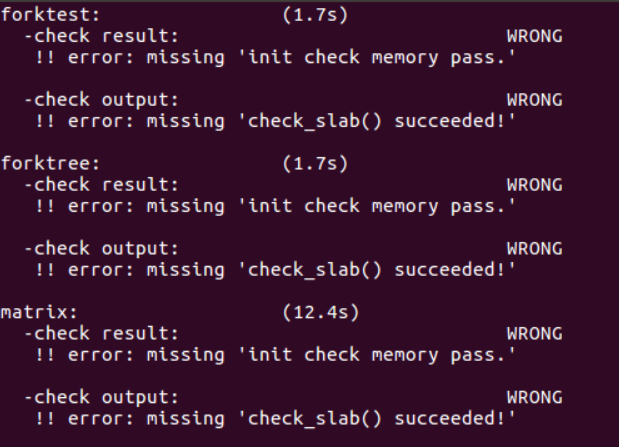
用以上函数给stride\_scheduler成员初始化

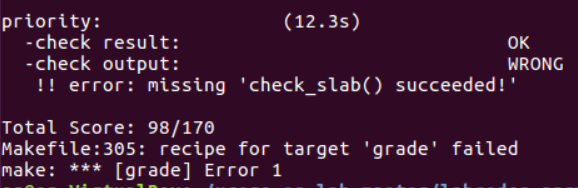
**MAKE GRADE:**

****

****

****

****

****

运行后结果中会有错误