

## 目录

Base Function .....	1
Item 1 .....	9
Item 2 .....	14
Item 3 .....	20

# Base Function

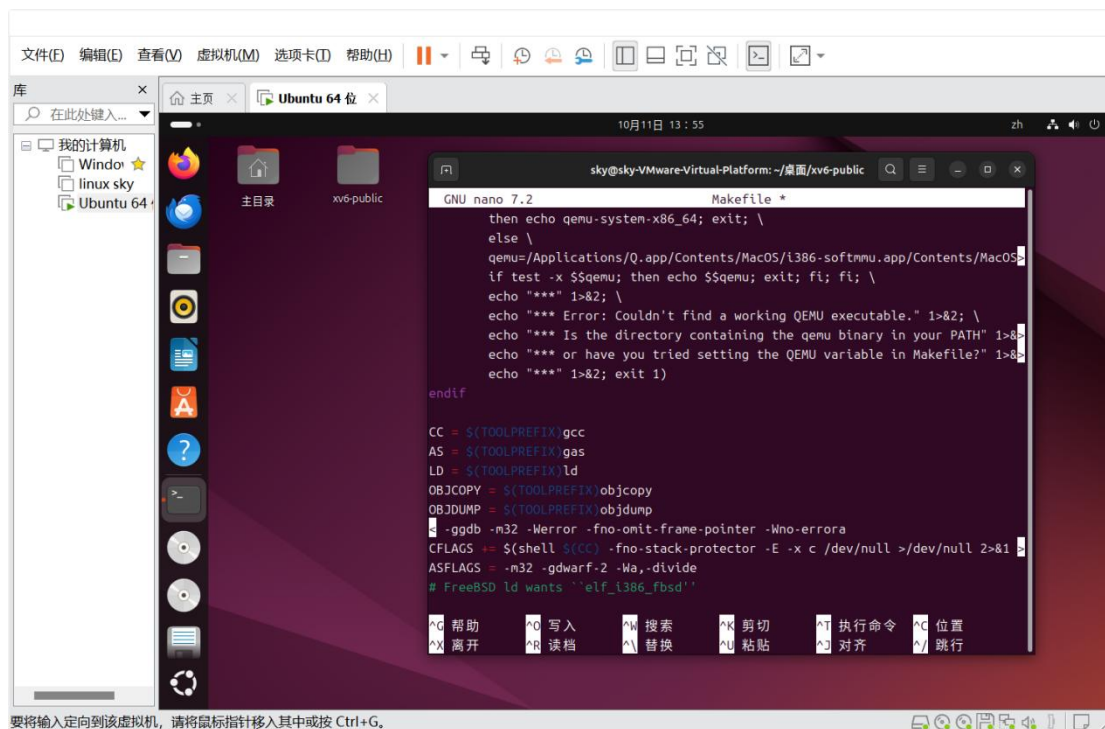
1. 打开终端，输入“`sudo apt-get install -y build-essential gdb git gcc-multilib`”安装工具链
2. 输入 `gcc --version`，出现以下图片则证明安装成功

```
sky@sky-VMware-Virtual-Platform:~/桌面$ gcc --version
gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

3. 输入命令“`git clone https://github.com/mit-pdos/xv6-public.git`”克隆 xv6 项目代码库
4. `cd xv6-public` 后输入 `make` 以编译项目代码
5. 发现有报错——回溯源码，在 [https://www.reddit.com/r/osdev/comments/16g4mg3/xv6\\_make\\_issue/?rdt=34243](https://www.reddit.com/r/osdev/comments/16g4mg3/xv6_make_issue/?rdt=34243) 这里发现答案：The x86 version of xv6 is no longer maintained. Try following the cross-compiler instructions to build an older version of GCC.——该版本不再维护，请根据说明构建旧版。故搭建旧版。
6. 输入指令 `sudo apt remove gcc-13` 来移除 gcc 13.2.0
7. 输入指令 `sudo apt install gcc-12 g++-12` 来安装 gcc12.3.0

```
sky@sky-VMware-Virtual-Platform:~/桌面$ gcc --version
gcc (Ubuntu 12.3.0-17ubuntu1) 12.3.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

8. 接下来 `cd` 过去后 `make`，但是发现会出现报错。发现是 warning 的问题
9. 输入 `nano makefile`，找到 `CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie` 这一行，在末尾添加 `-Wno-error`，修改后即为 `CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -Wno-error`，然后按 `ctrl+O` 写入



10. 然后再 cd 过去输入 make，发现编译成功。

11. 阅读 proc.c 文件，理解当前调度器的运行原理：当前算法为简单的轮转调度算法（Round-Robin Scheduling），其基本思想是将处理器的时间划分为固定长度的时间片，然后按照先来先服务的顺序循环地分配给每个就绪进程，使所有进程都能公平地获得 CPU 的执行机会。对其代码的详解如下：

该函数是一个死循环，持续执行，直到系统关闭。每个 CPU 都有自己的调度器，负责管理该 CPU 上的进程调度：

```

1.  void
2.  scheduler(void)
3.  {
4.      struct proc *p;
5.      struct cpu *c = mycpu(); // 获取当前 CPU 对象
6.      c->proc = 0; // 当前 CPU 尚未运行任何进程
7.
8.      for(;;){ // 死循环，调度器不断地运行
9.          sti(); // 使能中断，以便处理外部中断请求
10.
11.         // 通过 acquire(&ptable.lock) 锁住进程表 (ptable)，以确保在多核 CPU 上
            // 操作进程表时，不会出现并发访问的问题
12.         acquire(&ptable.lock); // 获取进程表的锁，保证在操作进程表时不会发生
            // 竞态条件
13.
14.         // 遍历进程表，寻找处于 RUNNABLE 状态的进程
15.         // 它通过一个 for 循环遍历进程表 ptable.proc，寻找状态为 RUNNABLE 的进
            // 程。每个进程的状态 (state) 决定了它当前所处的状态
16.         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

```

```

17.         if(p->state != RUNNABLE) // 只调度处于 RUNNABLE 状态的进程
18.             continue;
19.
20.         // 找到 RUNNABLE 进程后, 准备调度该进程
21.         c->proc = p; // 将当前 CPU 的进程指针指向该进程
22.
23.         switchvm(p); // 目标进程运行完毕或被其他条件打断后, 控制将返回调度
           器, 调度器通过 switchkvm() 恢复内核的虚拟内存空间, 为下一个进程调度做好准备。
24.         p->state = RUNNING; // 将该进程的状态设置为 RUNNING
25.
26.         // 保存当前调度器的 CPU 状态, 并将控制权交给目标进程的状态(即上下文),
           这样目标进程就可以开始执行。
27.         swtch(&(c->scheduler), p->context);
28.         switchkvm(); // 恢复内核虚拟内存空间
29.
30.         // 进程运行结束, 重置当前 CPU 的进程指针
31.         c->proc = 0;
32.     }
33.
34.     release(&ptable.lock); // 调度器会在每次遍历完进程表后通
           过 release(&ptable.lock) 释放进程表的锁, 使其他 CPU 或内核线程可以访问进程
           表。随后, 调度器返回到主循环, 继续寻找下一个可以运行的进程
35. }
36. }

```

12. 然后阅读 proc.h 文件, 其定义了许多结构体, 具体解释如下:

```

1.  struct cpu {
2.     uchar apicid; // 本地 APIC ID, 用于标识每个 CPU 的唯一编号
3.     struct context *scheduler; // 用于调度器上下文切换的指针
4.     struct taskstate ts; // 任务状态段, 用于 x86 系统中断时找到栈指针
5.     struct segdesc gdt[NSEGS]; // 全局描述符表, 用于内存分段
6.     volatile uint started; // CPU 是否已经启动的标志
7.     int ncli; // pushcli 嵌套的深度, 用于管理中断
8.     int intena; // 在调用 pushcli 之前中断是否启用的标志
9.     struct proc *proc; // 当前正在该 CPU 上运行的进程, 如果没有则为 NULL
10. };
11.
12. struct context {
13.     uint edi; // edi 是通用寄存器之一, 用于存储数据或指针。
14.     uint esi; // esi 也是一个通用寄存器, 常用于数据操作。
15.     uint ebx; // ebx 是一个基址寄存器, 用于存储基地址或指针。
16.     uint ebp; // ebp 通常用作栈帧指针, 指向当前函数调用的栈帧。
17.     uint eip; // eip 是指令指针, 指向下一条将要执行的指令。
18. };
19.

```

```

20.  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE }; // U
    NUSED、EMBRYO、SLEEPING、RUNNABLE、RUNNING、ZOMBIE：进程未使用状态、进程正在被创建、
    进程处于睡眠状态、进程已经准备好运行、进程正在运行中、
21.  struct proc {
22.      uint sz; // 进程内存的大小（以字节为单位）
23.      pde_t* pgdir; // 页表地址
24.      char *kstack; // 内核栈的底部地址
25.      enum procstate state; // 进程的状态
26.      int pid; // 进程的唯一标识符
27.      struct proc *parent; // 父进程
28.      struct trapframe *tf; // 当前系统调用的陷阱帧
29.      struct context *context; // 用于运行进程的上下文（用于调度器切换）
30.      void *chan; // 如果非零，进程正在该通道上休眠
31.      int killed; // 如果非零，表示该进程已被杀死
32.      struct file *ofile[NOFILE]; // 进程打开的文件数组
33.      struct inode *cwd; // 当前工作目录的指针
34.      char name[16]; // 进程的名称（用于调试）
35.  };

```

13. 实验开始：首先，在 `proc.h` 上添加优先级表示：在 `struct proc` 中添加字段 `int priority;` // 优先级，范围为 0-31

14. 在 `proc.c` 文件里的 `allocproc(void)` 函数里，初始化每个进程的优先级，通过在函数里添加 `p->priority = DEFAULT_PRIORITY;` 来设置默认优先级。并且在文件头通过代码 `#define DEFAULT_PRIORITY 12` 来定义初始值为 12。

```

found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->priority = DEFAULT_PRIORITY; |
    release(&ptable.lock);

```

15. 现在我们修改 `scheduler`，具体思路为：遍历进程表时，除了检查进程是否处于 `RUNNABLE` 状态，还要比较进程的优先级，选择优先级最高的进程执行。具体修改后的函数为：

```

1.  void
2.  scheduler(void)
3.  {
4.      struct proc *p;
5.      struct proc *highest_p; // 用来记录优先级最高的进程
6.      struct cpu *c = mycpu();
7.      c->proc = 0;
8.
9.      for(;;){
10.         // 启用中断
11.         sti();
12.
13.         // 锁住进程表，确保遍历时不受干扰
14.         acquire(&ptable.lock);

```

```

15.      // 初始化为 NULL，表示尚未找到优先级最高的进程
16.      highest_p = 0;
17.
18.      // 遍历进程表，寻找优先级最高的进程
19.      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
20.          // 跳过非 RUNNABLE 状态的进程
21.          if(p->state != RUNNABLE)
22.              continue;
23.
24.          // 如果目前尚未找到可运行进程，或者当前进程的优先级更高
25.          if (highest_p == 0 || p->priority > highest_p->priority) {
26.              // 更新为优先级最高的进程
27.              highest_p = p;
28.          }
29.      }
30.
31.      // 如果找到了优先级最高的进程，进行进程切换
32.      if (highest_p != 0) {
33.          // 切换到优先级最高的进程
34.          // 设置当前 CPU 正在运行的进程
35.          c->proc = highest_p;
36.          // 切换到该进程的用户内存空间
37.          switchvm(highest_p);
38.          // 将进程状态设置为 RUNNING
39.          highest_p->state = RUNNING;
40.
41.          // 进行上下文切换，保存调度器的状态并切换到进程
42.          swtch(&(c->scheduler), highest_p->context);
43.          // 切换回内核虚拟内存
44.          switchkvm();
45.
46.          // 当该进程时间片结束或其他情况导致其暂停时，重置当前 CPU 运行的
           进程
47.          c->proc = 0;
48.      }
49.
50.      // 释放进程表锁
51.      release(&ptable.lock);
52.  }
53. }

```

16. 然后我们需要修改系统调用，保证我们的优先级函数能够被成功调用。

17. 在 syscall.h 末增加对于优先级调用的定义:#define SYS\_setpriority 22

18. 接着在 sysproc.c 中实现 sys\_setpriority 代码，来实现进程调用：

```

1.      int

```

```

2.  sys_setpriority(void)
3.  {
4.      int pid, priority;
5.
6.      // 获取传递给系统调用的参数
7.      if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
8.          return -1;
9.
10.     struct proc *p;
11.
12.     // 获取进程表锁，保证进程表的安全访问
13.     acquire(&ptable.lock);
14.
15.     // 遍历进程表，找到目标进程
16.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
17.         if(p->pid == pid){
18.             p->priority = priority; // 更新进程的优先级
19.             break;
20.         }
21.     }
22.
23.     // 释放进程表锁
24.     release(&ptable.lock);
25.
26.     return 0;
27. }

```

19. 在 `usys.S` 文件末尾添加 `SYSCALL(setpriority)` 来添加系统调用的条目。

20. 接着我们需要在 `syscall.c` 文件中添加对 `setpriority` 系统调用的支持，首先声明 `sys_setpriority` 函数，并在系统调用处理函数数组 `syscalls[]` 中为 `setpriority` 预留一个位置。

21. 在 `syscall.c` 文件的顶部，我们需要添加 `sys_setpriority` 的声明。即在其顶部添加代码 `extern int sys_setpriority(void);`

22. 然后添加 `sys_setpriority` 到 `syscalls[]` 数组，具体为：在 `static int (*syscalls[])(void)` 中为 `SYS_setpriority` 分配一个位置。找到数组定义的位置，并将 `SYS_setpriority` 和 `sys_setpriority` 对应起来。即在该数组末尾添加 `[SYS_setpriority] sys_setpriority,`

23. 接着新建一个测试代码，命名为：`setpriority_test.c`。

```

1.  #include "types.h"
2.  #include "stat.h"
3.  #include "user.h"
4.
5.  int main() {
6.      int pid = fork(); // 创建子进程
7.
8.      if (pid == 0) {
9.          // 子进程，设置优先级为 15（较高的优先级）

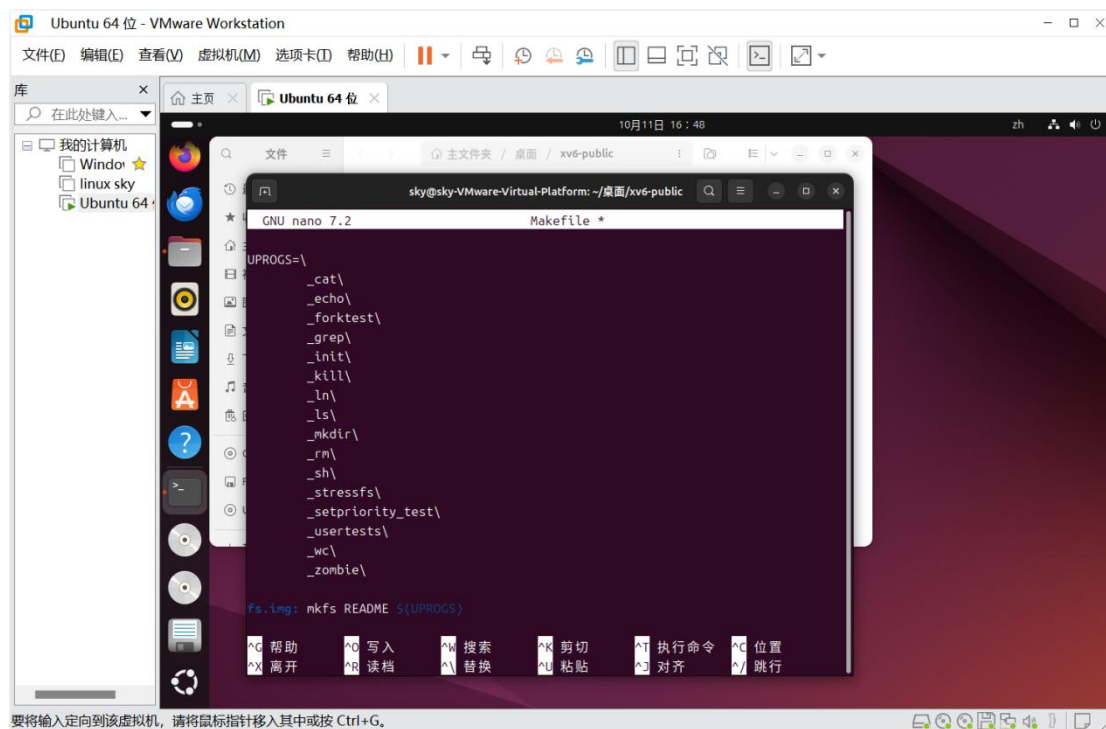
```

```

10.     setpriority(getpid(), 15);
11.     while (1) {
12.         printf(1, "Child running with priority 15\n");
13.         sleep(100);
14.     }
15. } else {
16.     // 父进程, 设置优先级为 5 (较低的优先级)
17.     setpriority(getpid(), 5);
18.     while (1) {
19.         printf(1, "Parent running with priority 5\n");
20.         sleep(100);
21.     }
22. }
23.
24.     return 0;
25. }

```

24. 在文件中编辑 Makefile, 在 UPROGS 中添加 setpriority\_test.c 进程



25. 接着, 在 proc.h 的结构体中添加如下代码:

```

1.     struct proc {
2.         uint sz;                // Size of process memory (bytes)
3.         pde_t* pgdir;          // Page table
4.         char *kstack;           // Bottom of kernel stack for this process
5.         enum procstate state;    // Process state
6.         int pid;                // Process ID
7.         struct proc *parent;    // Parent process
8.         struct trapframe *tf;   // Trap frame for current syscall

```



26. 在 `proc.h` 中添加如下声明，以便 `ptable` 能在其他文件中访问：

27. 接下来：先输入 `make clean` 以清楚之前编译的版本，然后再输入 `make` 以编译现在的版本。

29. 然后输入 `make qemu` 来运行该操作系统

The screenshot shows the VMware Workstation application window titled "Ubuntu 64 位 - VMware Workstation". The main view is a terminal window for a virtual machine named "sky". The terminal prompt is "sky@sky-VMware-Virtual-Platform: ~/桌面/xv6-public". The terminal output shows the following sequence of commands and their results:

```

cpu0: starting 0
fsb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Rhythmbox > priority_test.c
exec: fall
exec ./setpriority_test.c failed
$ ./setpriority_test.c
Parent running with priority 5
Child running with priority 15
Child running with priority 15
Parent running with priority 5
Child running with priority 15
Parent running with priority 5
Child running with priority 15
Parent running with priority 5
Child running with priority 15
Parent running with priority 5
Child running with priority 15
Parent running with priority 5
Child running with priority 15
Parent running with priority 5
Child running with priority 15
Parent running with priority 5
Child running with priority 15
Parent running with priority 5

```

At the bottom of the screen, there is a caption in Chinese: "要将输入定向到该虚拟机，请将鼠标指针移入其中或按 Ctrl+G。"

要将输入定向到该虚拟机，请将鼠标指针移入其中或按 Ctrl+G。



# Item 1

1. 将第一个完善的版本命名为 `xv6-public_first_edition`，复制一个新版，`cd` 进去后输入 `make clean` 清除之前的编译

2. 在 `param.h` 中新增定义：

```
1.  #define AGING_THRESHOLD 10 // 等待10个时钟周期后提高优先级
2.  #define AGING_STEP 1      // 每次提高优先级的步长
3.  #define MAX_PRIORITY 31    // 最大优先级
4.  #define MIN_PRIORITY 0     // 最小优先级
```

3. 在 `proc.h` 的 `struct proc{}` 中新增对累积等待时间的定义：

```
int aging_ticks; // 累计等待的时钟周期
```

4. 修改 `proc.c` 函数里的 `scheduler` 函数，具体思路为为等待的进程增加老化机制。若进程在 `RUNNABLE` 状态但未被调度，则提升其优先级。若进程处于 `RUNNING` 状态，则降低优先级：

```
1.  void scheduler(void) {
2.      struct proc *p;
3.      struct proc *highest_p;
4.      struct cpu *c = mycpu();
5.      c->proc = 0;
6.
7.      for(;;){
8.          // 使当前处理器支持中断
9.          sti();
10.
11.         // 遍历进程表，查找可运行的进程
12.         acquire(&ptable.lock);
13.
14.         highest_p = 0;
15.         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
16.             if(p->state != RUNNABLE)
17.                 continue;
18.
19.             // 对等待的进程进行老化，增加优先级
20.             if (p != highest_p) {
21.                 p->aging_ticks++;
22.                 if (p->aging_ticks >= AGING_THRESHOLD) {
23.                     if (p->priority < MAX_PRIORITY) {
24.                         p->priority++;
25.                         printf(1, "Process %d: aging - priority increased to %d\n", p->pid, p->priority);
26.                     }
27.                     p->aging_ticks = 0;
28.                 }
29.             }
```

```

30.
31.     // 找到当前优先级最高的进程
32.     if(highest_p == 0 || p->priority > highest_p->priority) {
33.         highest_p = p;
34.     }
35. }
36.
37. if(highest_p != 0) {
38.     // 切换到优先级最高的进程
39.     c->proc = highest_p;
40.     switchvm(highest_p);
41.     highest_p->state = RUNNING;
42.
43.     printf(1, "Process %d is running with priority %d\n", highest_
        p->pid, highest_p->priority);
44.
45.     // 运行时降低优先级，避免过度占用 CPU 资源
46.     if (highest_p->priority > MIN_PRIORITY) {
47.         highest_p->priority--;
48.         printf(1, "Process %d: running - priority decreased to %d\n"
            , highest_p->pid, highest_p->priority);
49.     }
50.
51.     swtch(&(c->scheduler), highest_p->context);
52.     switchkvm();
53.
54.     // 当前进程完成一个调度周期后，将其从处理器释放
55.     c->proc = 0;
56. }
57.
58. release(&ptable.lock);
59. }
60. }

```

5. 修改 sleep 和 wake 函数——修改 sleep 函数，确保进程进入睡眠状态时重置老化计时器。修改 wakeup 函数，确保进程从睡眠中恢复为 RUNNABLE 状态时重置老化计时器：

```

1. void sleep(void *chan, struct spinlock *lk) {
2.     struct proc *p = myproc();
3.     if(p == 0)
4.         panic("sleep");
5.
6.     if(lk == 0)
7.         panic("sleep without lk");
8.
9.     // 获取 ptable.Lock 并释放当前锁

```

```

10.     if(lk != &ptable.lock) {
11.         acquire(&ptable.lock);
12.         release(lk);
13.     }
14.
15.     // 设置进程状态为 SLEEPING 并进入调度
16.     p->chan = chan;
17.     p->state = SLEEPING;
18.     p->aging_ticks = 0; // 进入睡眠时重置老化计时器
19.
20.     sched();
21.
22.     // 清空 chan 并重新获取原始锁
23.     p->chan = 0;
24.     if(lk != &ptable.lock) {
25.         release(&ptable.lock);
26.         acquire(lk);
27.     }
28. }
29.
30. void wakeup(void *chan) {
31.     struct proc *p;
32.
33.     acquire(&ptable.lock);
34.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
35.         if(p->state == SLEEPING && p->chan == chan) {
36.             p->state = RUNNABLE;
37.             p->aging_ticks = 0; // 唤醒时重置老化计时器
38.         }
39.     release(&ptable.lock);
40. }

```

6. 在 proc.c 中新增 sys\_setpriority 函数，用于动态调整进程的优先级:

```

1.  int sys_setpriority(void)
2.  {
3.      int pid, priority;
4.
5.      if(argint(0, &pid) < 0)
6.          return -1;
7.      if(argint(1, &priority) < 0 || priority < MIN_PRIORITY || priority
        > MAX_PRIORITY)
8.          return -1;
9.
10.     struct proc *p;
11.     acquire(&ptable.lock);

```

```

12.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
13.         if(p->pid == pid) {
14.             p->priority = priority;
15.             break;
16.         }
17.     }
18.     release(&ptable.lock);
19.     return 0;
20. }

```

7. 在 sysproc.c 中声明 sys\_setpriority——新增 `extern int sys_setpriority(void);`
8. 在 syscall.h 中添加对 setpriority 分配系统调用的编号: `#define SYS_setpriority 22`
9. 在 user.h 处声明 setpriority: `int setpriority(int pid, int priority);`
10. 接下来, 更新 setpriority\_test.c 函数为:

```

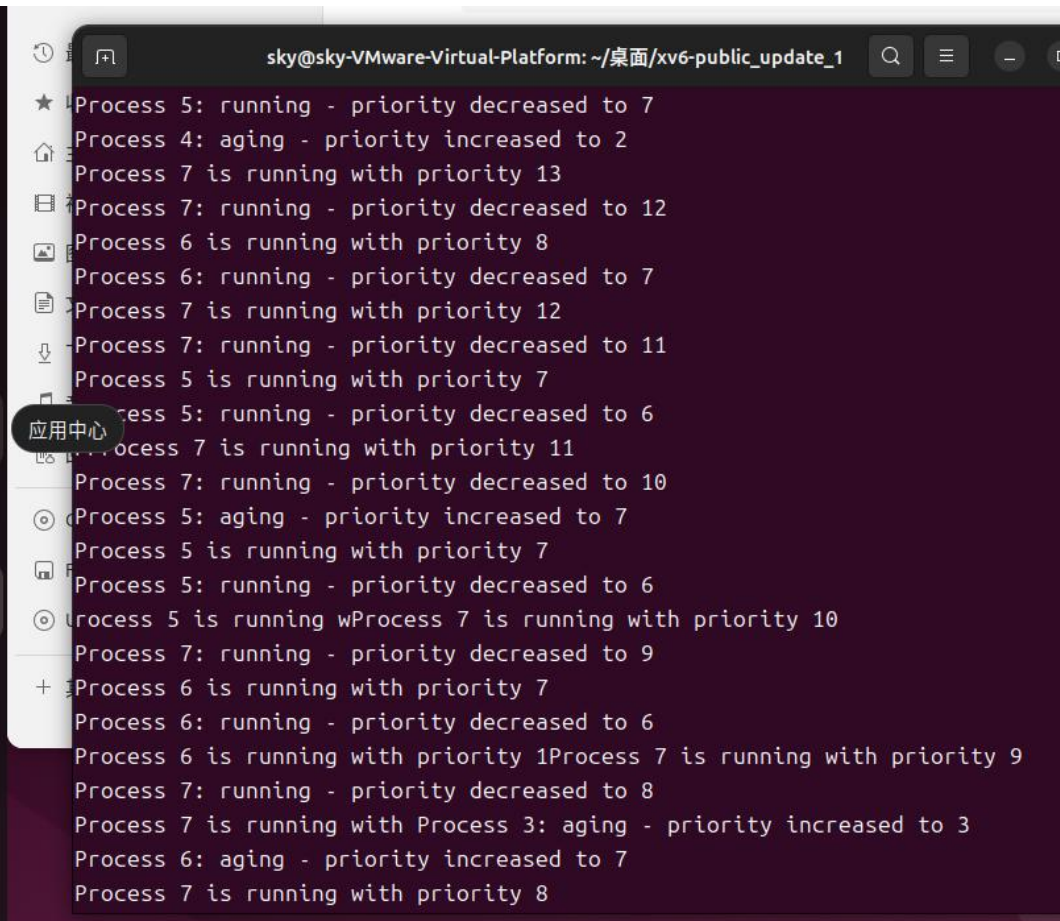
1.  #include "types.h"
2.  #include "stat.h"
3.  #include "user.h"
4.
5.  #define NUM_PROCS 5 // 定义子进程数量
6.  #define RUN_TIME 100 // 每个子进程运行的时间片
7.
8.  void do_work(int priority) {
9.      int i;
10.     setpriority(getpid(), priority); // 设置进程优先级
11.     printf(1, "Process %d started with priority %d\n", getpid(), pri
        ority);
12.
13.     for (i = 0; i < RUN_TIME; i++) {
14.         printf(1, "Process %d is running with priority %d, iteration
            %d\n", getpid(), priority, i);
15.         sleep(10); // 模拟一些工作, 避免独占 CPU
16.     }
17.     printf(1, "Process %d finished with priority %d\n", getpid(), pr
        iority);
18. }
19.
20. int main(void) {
21.     int pids[NUM_PROCS]; // 保存子进程的 pid
22.     int initial_priority = 5;
23.
24.     // 创建多个子进程, 每个子进程设置不同的初始优先级
25.     for (int i = 0; i < NUM_PROCS; i++) {
26.         pids[i] = fork();
27.         if (pids[i] == 0) { // 子进程
28.             do_work(initial_priority + i * 5); // 设置初始优先级, 逐渐

```

增高

```
29.         exit(); // 结束子进程
30.     }
31. }
32.
33. // 父进程负责监控每个子进程
34. for (int i = 0; i < NUM_PROCS; i++) {
35.     wait(); // 等待每个子进程结束
36.     printf(1, "Process %d exited\n", pids[i]);
37. }
38.
39. // 父进程结束
40. printf(1, "All child processes have exited.\n");
41. exit();
42. }
```

11. 先输入 make,然后输入 make qemu 来运行该操作系统,最后输入 ./setpriority\_test.c,发现达到要求,具体体现在:

A terminal window titled 'sky@sky-VMware-Virtual-Platform: ~/桌面/xv6-public\_update\_1' displays the output of the ./setpriority\_test.c program. The output consists of multiple lines of status reports for various processes, showing their current state and priority. For example, 'Process 5: running - priority decreased to 7' and 'Process 7 is running with priority 13'. The terminal has a dark background with light-colored text. On the left side of the terminal window, there is a sidebar with icons for file management and a '应用中心' (Application Center) button.

```
sky@sky-VMware-Virtual-Platform: ~/桌面/xv6-public_update_1
★ Process 5: running - priority decreased to 7
Process 4: aging - priority increased to 2
Process 7 is running with priority 13
Process 7: running - priority decreased to 12
Process 6 is running with priority 8
Process 6: running - priority decreased to 7
Process 7 is running with priority 12
Process 7: running - priority decreased to 11
Process 5 is running with priority 7
Process 5: running - priority decreased to 6
Process 7 is running with priority 11
Process 7: running - priority decreased to 10
Process 5: aging - priority increased to 7
Process 5 is running with priority 7
Process 5: running - priority decreased to 6
Process 5 is running wProcess 7 is running with priority 10
Process 7: running - priority decreased to 9
Process 6 is running with priority 7
Process 6: running - priority decreased to 6
Process 6 is running with priority 1Process 7 is running with priority 9
Process 7: running - priority decreased to 8
Process 7 is running with Process 3: aging - priority increased to 3
Process 6: aging - priority increased to 7
Process 7 is running with priority 8
```

优先级的动态调整:

- 可以看到一些进程的优先级在运行时降低,例如:

- Process 5: running - priority decreased to 6
- Process 7: running - priority decreased to 12
- 其他进程在等待时会增加优先级，例如：
  - Process 4: aging - priority increased to 2
  - Process 5: aging - priority increased to 7

## Item 2

1. 为了实现优先级捐献/优先级继承，我们需要在现有的进程调度机制和锁管理机制上进行改进。具体的做法是：当一个高优先级的进程请求被低优先级进程持有的锁时，我们要将高优先级进程的优先级“捐献”给锁的持有者，以防止 优先级反转 问题。等到锁被释放后，恢复持有者的原始优先级。

2. 首先，在 `proc.h` 中修改 `struct proc` 结构体，增加以下内容：

1. `int donated_priority;` // 被捐献的优先级
2. `struct spinlock *lock_holding;` // 持有的锁

3. 接着，修改 `spinlock.c` 中锁的获取和释放逻辑，首先，修改 `acquire` 函数：

```
1. void acquire(struct spinlock *lk) {
2.     pushcli(); // 禁用中断
3.
4.     if(holding(lk))
5.         panic("acquire");
6.
7.     // 获取当前持有锁的进程
8.     struct proc *holder = myproc()->lock_holding;
9.
10.    // 如果锁已被持有，并且持有者的优先级低于当前进程，则进行优先级捐献
11.    if (lk->locked && holder && holder->priority < myproc()->priority) {
12.        // 记录持有者的原始优先级，并捐献优先级
13.        holder->donated_priority = myproc()->priority;
14.        cprintf("Priority donation: process %d donates priority to process %d\n", myproc()->pid, holder->pid);
15.    }
16.
17.    // 获取锁
18.    while(xchg(&lk->locked, 1) != 0)
19.        ;
20.    __sync_synchronize();
21.
22.    // 当前进程持有锁
```



```
23.     myproc()->lock_holding = lk;
24. }
```

4.

5. 接着修改 release 函数

```
1.  void release(struct spinlock *lk) {
2.     if(!holding(lk))
3.         panic("release");
4.
5.     struct proc *holder = myproc();
6.
7.     // 如果持有锁的进程曾经获得过优先级捐献, 在释放锁时恢复原始优先级
8.     if (holder->donated_priority != 0) {
9.         holder->priority = holder->donated_priority;
10.        holder->donated_priority = 0; // 清除捐献的优先级
11.        cprintf("Priority restored: process %d restores its original priorit
y\n", holder->pid);
12.    }
13.
14.    // 清除锁的持有状态
15.    holder->lock_holding = 0;
16.
17.    __sync_synchronize();
18.    xchg(&lk->locked, 0);
19.    popcli();
20. }
```

6. 然后修改 proc.c 函数中的 scheduler 函数以实现支持使用进程的捐献优先级制度:

```
1.  void scheduler(void) {
2.     struct proc *p;
3.     struct cpu *c = mycpu();
4.     c->proc = 0;
5.
6.     for(;;){
7.         sti();
8.         acquire(&ptable.lock);
9.
10.        struct proc *highest_p = 0;
11.
12.        // 查找具有最高优先级的进程
13.        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
14.            if(p->state != RUNNABLE)
15.                continue;
16.
17.            // 使用捐献的优先级 (如果有)
18.            int effective_priority = (p->donated_priority != 0) ? p-
```

```

    >donated_priority : p->priority;
19.
20.         if(highest_p == 0 || effective_priority > highest_p->priority) {
21.             highest_p = p;
22.         }
23.     }
24.
25.         // 切换到具有最高优先级的进程
26.         if(highest_p != 0) {
27.             c->proc = highest_p;
28.             switchvm(highest_p);
29.             highest_p->state = RUNNING;
30.             cprintf("Process %d is running with effective priority %d\n", highest_p->pid, (highest_p->donated_priority != 0) ? highest_p->donated_priority : highest_p->priority);
31.
32.             swtch(&(c->scheduler), highest_p->context);
33.             switchkvm();
34.             c->proc = 0;
35.         }
36.
37.         release(&ptable.lock);
38.     }
39. }

```

7. 接着修改 sleep 和 wakeup 函数

8. sleep 函数修改如下:

```

1. void sleep(void *chan, struct spinlock *lk) {
2.     struct proc *p = myproc();
3.     if(p == 0)
4.         panic("sleep");
5.
6.     if(lk == 0)
7.         panic("sleep without lk");
8.
9.     // 获取 ptable.lock 并释放当前锁
10.    if(lk != &ptable.lock) {
11.        acquire(&ptable.lock);
12.        release(lk);
13.    }
14.
15.    // 设置进程状态为 SLEEPING
16.    p->chan = chan;
17.    p->state = SLEEPING;

```

```

18.     p->aging_ticks = 0; // 重置老化计时器
19.
20.     sched();
21.
22.     // 清空 chan 并重新获取原始锁
23.     p->chan = 0;
24.     if(lk != &ptable.lock) {
25.         release(&ptable.lock);
26.         acquire(lk);
27.     }
28.
29.     // 如果进程有捐献的优先级, 恢复原始优先级
30.     if (p->donated_priority != 0) {
31.         p->priority = p->donated_priority;
32.         p->donated_priority = 0;
33.     }
34. }

```

9. wakeup 函数修改如下:

```

1.     void wakeup(void *chan) {
2.         struct proc *p;
3.
4.         acquire(&ptable.lock);
5.         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
6.             if(p->state == SLEEPING && p->chan == chan) {
7.                 p->state = RUNNABLE;
8.
9.                 // 唤醒时恢复进程的原始优先级
10.                if (p->donated_priority != 0) {
11.                    p->priority = p->donated_priority;
12.                    p->donated_priority = 0;
13.                }
14.            }
15.        }
16.        release(&ptable.lock);
17.    }

```

10. 然后编辑测试函数 setpriority\_test.c:

```

1.     #include "types.h"
2.     #include "stat.h"
3.     #include "user.h"
4.
5.     int lock = 0; // 锁状态, 0 表示未锁定, 1 表示锁定
6.
7.     // 模拟获取锁

```

```
8.  void acquire_lock(int *lk) {
9.      while (*lk == 1) {
10.         // 自旋等待锁释放
11.     }
12.     *lk = 1; // 获取锁
13. }
14.
15. // 模拟释放锁
16. void release_lock(int *lk) {
17.     *lk = 0; // 释放锁
18. }
19.
20. // 模拟低优先级任务
21. void low_priority_task() {
22.     acquire_lock(&lock); // 获取锁
23.     printf(1, "Low priority task running...\n");
24.
25.     // 模拟延迟操作
26.     for (int i = 0; i < 100000000; i++) {
27.         asm volatile("nop"); // 占用 CPU
28.     }
29.
30.     printf(1, "Low priority task finished.\n");
31.     release_lock(&lock); // 释放锁
32. }
33.
34. // 模拟高优先级任务
35. void high_priority_task() {
36.     printf(1, "High priority task trying to acquire lock...\n");
37.
38.     acquire_lock(&lock); // 尝试获取锁
39.
40.     printf(1, "High priority task running...\n");
41.
42.     release_lock(&lock); // 释放锁
43. }
44.
45. int main() {
46.     int pid_low = fork();
47.     if (pid_low == 0) {
48.         setpriority(getpid(), 5); // 低优先级进程
49.         low_priority_task();
50.         exit();
51.     }
```

```

52.
53.     // 等待低优先级进程开始执行
54.     sleep(50);
55.
56.     int pid_high = fork();
57.     if (pid_high == 0) {
58.         setpriority(getpid(), 20); // 高优先级进程
59.         high_priority_task();
60.         exit();
61.     }
62.
63.     wait(); // 等待子进程
64.     wait(); // 等待子进程
65.     exit();
66. }

```

12. 先输入 `make`, 然后输入 `make qemu` 来运行该操作系统, 最后输入 `./setpriority_test.c`, 发现达到要求, 具体体现在:

```

11. Low priority task running...
    High priority task trying to acquire lock...
    Priority donation: process 2 donates priority to process 1
    Low priority task finished.
    Priority restored: process 1 restores its original priority
    High priority task running...

```

**"Low priority task running...":** 低优先级任务开始运行并获取锁, 模拟执行一些占用 CPU 的操作。

**"High priority task trying to acquire lock...":** 高优先级任务在低优先级任务持有锁的情况下尝试获取锁, 并进入等待状态。

**"Priority donation: process 2 donates priority to process 1":** 实现了优先级捐献机制, 高优先级任务 (进程 2) 将其优先级捐献给持有锁的低优先级任务 (进程 1), 使得低优先级任务能够以较高的优先级继续运行, 从而尽快释放锁。

**"Low priority task finished.":** 低优先级任务完成工作并释放锁。

**"Priority restored: process 1 restores its original priority":** 当低优先级任务释放锁时, 它的优先级恢复到最初的低优先级。

**"High priority task running...":** 高优先级任务获得锁并开始执行。

## Item 3

1. 在 proc.h 中的 struct proct 里添加如下字段:

```
1. // 新增调度性能字段
2. int start_time;           // 进程开始时间
3. int end_time;             // 进程结束时间
4. int wait_time;            // 累计等待时间
5. int run_time;             // 累计运行时间
6. int last_scheduled;       // 上次被调度时间, 用于计算等待时间
```

2. 在 proc.c 中, 先修改 allocproc 函数来初始化调度性能字段 start\_time、wait\_time、run\_time 和 last\_scheduled。当进程被调度运行时, 我们需要更新其 run\_time 和 wait\_time。在 allocproc 函数中, 修改如下:

```
1. static struct proc*
2. allocproc(void)
3. {
4.     struct proc *p;
5.     char *sp;
6.
7.     acquire(&ptable.lock);
8.
9.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
10.        if(p->state == UNUSED)
11.            goto found;
12.
13.     release(&ptable.lock);
14.     return 0;
15.
16. found:
17.     p->state = EMBRYO;
18.     p->pid = nextpid++;
19.     p->priority = DEFAULT_PRIORITY; // 分配默认优先级
20.
21.     // 初始化调度性能字段
22.     p->start_time = ticks; // 设置进程开始时间
23.     p->end_time = 0;
24.     p->wait_time = 0;
25.     p->run_time = 0;
26.     p->last_scheduled = ticks; // 进程刚创建时的时间
27.
28.     release(&ptable.lock);
29.
30.     // Allocate kernel stack.
31.     if((p->kstack = kalloc()) == 0){
```



```

32.     p->state = UNUSED;
33.     return 0;
34. }
35. sp = p->kstack + KSTACKSIZE;
36.
37. // Leave room for trap frame.
38. sp -= sizeof *p->tf;
39. p->tf = (struct trapframe*)sp;
40.
41. sp -= 4;
42. *(uint*)sp = (uint)trapret;
43.
44. sp -= sizeof *p->context;
45. p->context = (struct context*)sp;
46. memset(p->context, 0, sizeof *p->context);
47. p->context->eip = (uint)forkret;
48.
49. return p;
50. }

```

3. 更新 scheduler 函数，需要在进程运行时更新 run\_time，在进程等待时更新 wait\_time:

```

1. void
2. scheduler(void)
3. {
4.     struct proc *p;
5.     struct proc *highest_p;
6.     struct cpu *c = mycpu();
7.     c->proc = 0;
8.
9.     for(;;){
10.         sti();
11.
12.         acquire(&ptable.lock);
13.
14.         highest_p = 0;
15.
16.         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
17.             if(p->state != RUNNABLE)
18.                 continue;
19.
20.             // 选择优先级最高的进程运行
21.             if(highest_p == 0 || p->priority > highest_p->priority) {
22.                 highest_p = p;
23.             }
24.         }

```

```

25.
26.     if(highest_p != 0) {
27.         // 更新等待时间
28.         highest_p->wait_time += (ticks - highest_p->last_scheduled);
29.
30.         c->proc = highest_p;
31.         switchvm(highest_p);
32.         highest_p->state = RUNNING;
33.
34.         swtch(&(c->scheduler), highest_p->context);
35.         switchkvm();
36.
37.         // 更新运行时间
38.         highest_p->run_time += (ticks - highest_p->last_scheduled);
39.
40.         // 记录上次调度时间
41.         highest_p->last_scheduled = ticks;
42.
43.         c->proc = 0;
44.     }
45.
46.     release(&ptable.lock);
47. }
48. }

```

4. 修改 `exit` 函数以打印调度性能数据：我们的设计为周转时间 = `end_time - start_time`、等待时间 已经通过 `wait_time` 字段进行跟踪、运行时间 已经通过 `run_time` 字段进行跟踪。

```

1.  void
2.  exit(void)
3.  {
4.      struct proc *curproc = myproc();
5.      struct proc *p;
6.      int fd;
7.
8.      if(curproc == initproc)
9.          panic("init exiting");
10.
11.     // 关闭所有打开的文件
12.     for(fd = 0; fd < NOFILE; fd++){
13.         if(curproc->ofile[fd]){
14.             fileclose(curproc->ofile[fd]);
15.             curproc->ofile[fd] = 0;
16.         }
17.     }
18.

```

```

19.     begin_op();
20.     iput(curproc->cwd);
21.     end_op();
22.     curproc->cwd = 0;
23.
24.     acquire(&ptable.lock);
25.
26.     wakeup1(curproc->parent);
27.
28.     // 将遗弃的子进程传递给 init
29.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
30.         if(p->parent == curproc){
31.             p->parent = initproc;
32.             if(p->state == ZOMBIE)
33.                 wakeup1(initproc);
34.         }
35.     }
36.
37.     // 记录进程的结束时间
38.     curproc->end_time = ticks;
39.
40.     // 计算并打印调度性能
41.     int turnaround_time = curproc->end_time - curproc->start_time;
42.     cprintf("Process %d Turnaround time: %d, Wait time: %d, Run time: %d\n",
43.            curproc->pid, turnaround_time, curproc->wait_time, curproc->run_time);
44.
45.     // 进入调度器, 永不返回
46.     curproc->state = ZOMBIE;
47.     sched();
48.     panic("zombie exit");
49. }

```

5. 然后，增加一个系统调用以获取调度性能在 `sysproc.c` 中添加 `sys_get_sched_performance()` 函数

```

1.     // sysproc.c
2.
3.     int
4.     sys_get_sched_performance(void)
5.     {
6.         int pid;
7.
8.         // 获取传递的 pid 参数
9.         if(argint(0, &pid) < 0)

```

```

10.     return -1;
11.
12.     struct proc *p;
13.     acquire(&ptable.lock);
14.
15.     // 遍历进程表, 找到匹配 pid 的进程
16.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
17.         if(p->pid == pid){
18.             int turnaround_time = p->end_time - p->start_time;
19.             int wait_time = p->wait_time;
20.             int run_time = p->run_time;
21.
22.             // 打印调度性能信息
23.             cprintf("Process %d - Turnaround time: %d, Wait time: %d, Run
                time: %d\n",
24.                    p->pid, turnaround_time, wait_time, run_time);
25.             release(&ptable.lock);
26.             return 0;
27.         }
28.     }
29.
30.     release(&ptable.lock);
31.     return -1; // 如果没有找到匹配的进程, 返回 -1
32. }

```

6. 在 syscall.h 中定义系统调用编号: `#define SYS_get_sched_performance 23`

7. 在 syscall.c 中注册系统调用, 具体注册方法为:

```

1.     extern int sys_fork(void);
2.     extern int sys_exit(void);
3.     extern int sys_wait(void);
4.     // 其他系统调用的 extern 声明...
5.
6.     extern int sys_get_sched_performance(void); // 声明新的系统调用
7.
8.     static int (*syscalls[])(void) = {
9.         [SYS_fork]    sys_fork,
10.        [SYS_exit]    sys_exit,
11.        [SYS_wait]    sys_wait,
12.        // 其他系统调用的函数指针...
13.        [SYS_get_sched_performance] sys_get_sched_performance, // 注册新的系
            统调用
14.    };

```

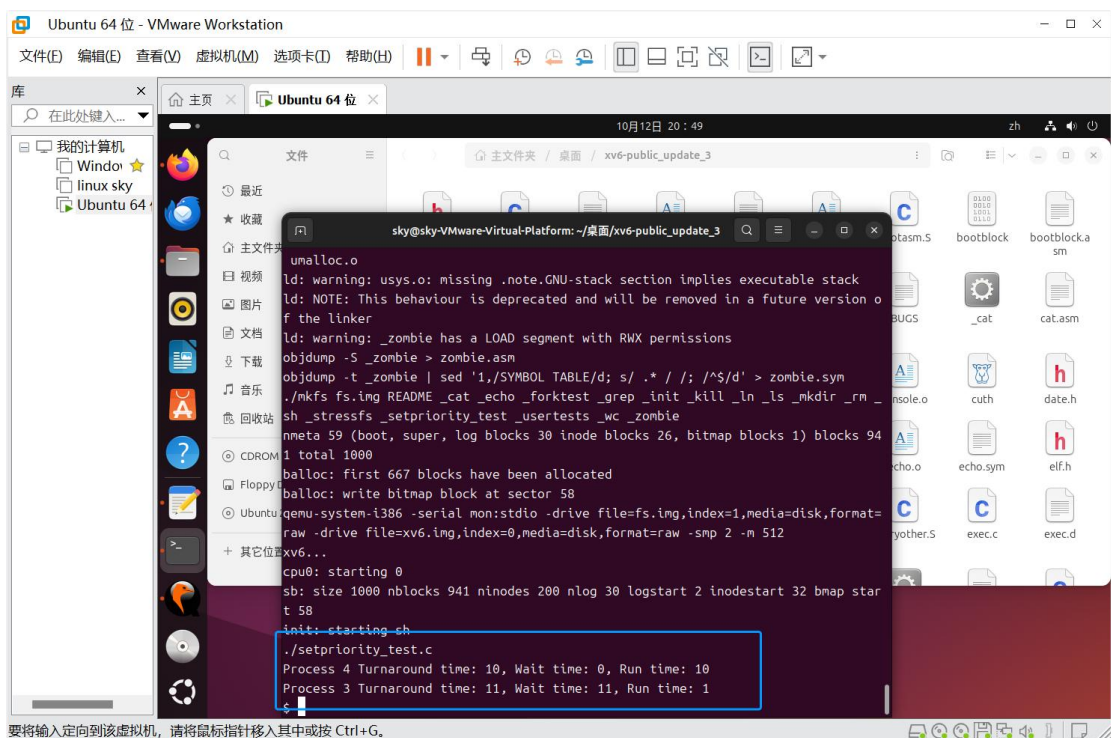
8. 在 usys.S 中添加用户态的系统调用接口。这会生成一个名为 `get_sched_performance` 的用户态函数, 它在调用时会通过中断触发内核态的系统调用。在最后一行添加 `SYSCALL(get_sched_performance)`

9. 在 user.h 中添加 get\_sched\_performance() 函数声明, 用户程序需要调用这个函数。int get\_sched\_performance(int pid);

10. 编写 setpriority\_test.c:

```
1. #include "types.h"
2. #include "user.h"
3.
4. int
5. main(void)
6. {
7.     int pid = fork(); // 创建一个子进程
8.
9.     if(pid == 0) {
10.        // 子进程
11.        for(int i = 0; i < 100000000; i++) {
12.            asm volatile("nop"); // 模拟一些工作
13.        }
14.        exit(); // 子进程退出
15.    } else {
16.        wait(); // 父进程等待子进程退出
17.        get_sched_performance(pid); // 调用新系统调用获取调度性能
18.    }
19.
20.    exit();
21. }
```

11. 先输入 make, 然后输入 make qemu 来运行该操作系统, 最后输入 ./setpriority\_test.c, 发现达到要求, 具体体现在



12. 要将输入定向到该虚拟机, 请将鼠标指针移入其中或按 Ctrl+G.