Base Function

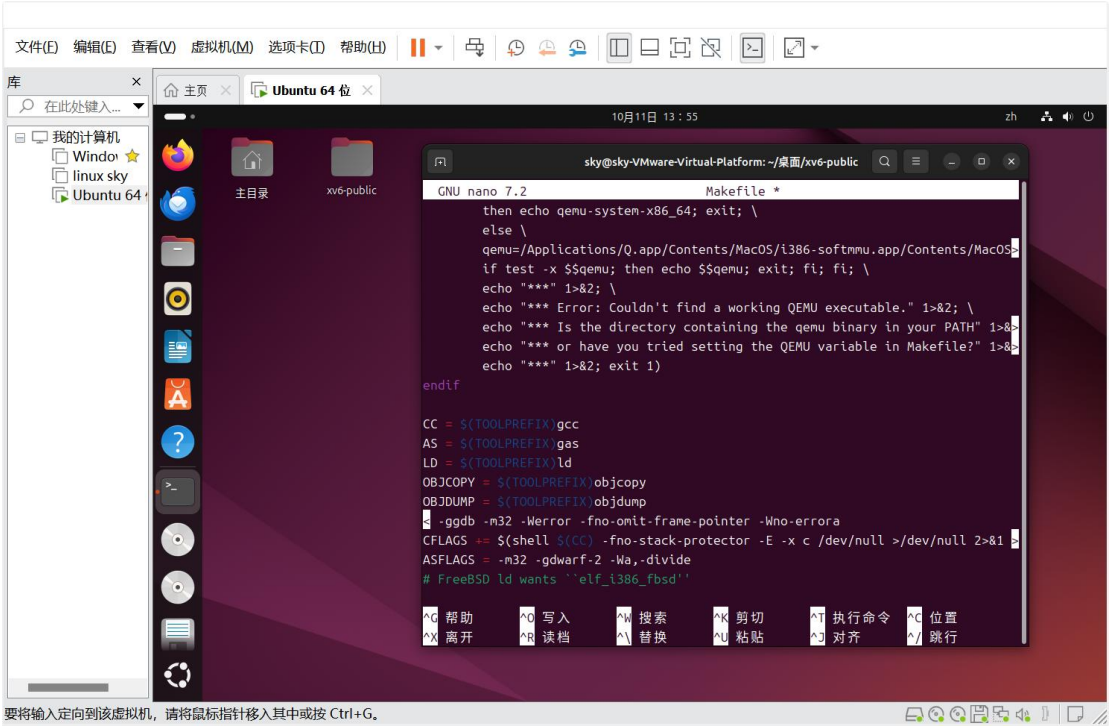
1. 打开终端，输入“sudo apt-get install -y build-essential gdb git gcc-multilib”安装工具链
2. 输入 gcc --version，出现以下图片则证明安装成功

```
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https://www.reddit.com/r/osdev/comments/16g4mg3/xv6_make_issue/?rdt=34243 这里发现答案：The x86 version of xv6 is no longer maintained.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，然后安 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++){
```

```
        if(p->state != RUNNABLE)  // 只调度处于 RUNNABLE 状态的进程
          continue;

        // 找到 RUNNABLE 进程后，准备调度该进程
        c->proc = p;  // 将当前 CPU 的进程指针指向该进程

        switchuvm(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。
        p->state = RUNNING;  // 将该进程的状态设置为 RUNNING

        // 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。
        swtch(&(c->scheduler), p->context);
        switchkvm();  // 恢复内核虚拟内存空间

        // 进程运行结束，重置当前 CPU 的进程指针
        c->proc = 0;
      }

      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
    }
  }
```

12. 然后阅读 proc.h 文件，其定义了许多结构体，具体解释如下：

```
  struct cpu {
    uchar apicid;                // 本地 APIC ID，用于标识每个 CPU 的唯一编号
    struct context *scheduler;   // 用于调度器上下文切换的指针
    struct taskstate ts;         // 任务状态段，用于 x86 系统中断时找到栈指针
    struct segdesc gdt[NSEGS];   // 全局描述符表，用于内存分段
    volatile uint started;       // CPU 是否已经启动的标志
    int ncli;                    // pushcli 嵌套的深度，用于管理中断
    int intena;                  // 在调用 pushcli 之前中断是否启用的标志
    struct proc *proc;           // 当前正在该 CPU 上运行的进程，如果没有则为 NULL
  };

  struct context {
    uint edi; //edi 是通用寄存器之一，用于存储数据或指针。
    uint esi; // esi 也是一个通用寄存器，常用于数据操作。
    uint ebx; // ebx 是一个基址寄存器，用于存储基地址或指针。
    uint ebp; // ebp 通常用作栈帧指针，指向当前函数调用的栈帧。
    uint eip; // eip 是指令指针，指向下一条将要执行的指令。
  };

```

```
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);
```

```
    // 初始化为 NULL，表示尚未找到优先级最高的进程
    highest_p = 0;

    // 遍历进程表，寻找优先级最高的进程
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  // 跳过非 RUNNABLE 状态的进程
      if(p->state != RUNNABLE)
        continue;

      // 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
      if (highest_p == 0 || p->priority > highest_p->priority) {
      // 更新为优先级最高的进程
        highest_p = p;
      }
    }

    // 如果找到了优先级最高的进程，进行进程切换
    if (highest_p != 0) {
      // 切换到优先级最高的进程
      // 设置当前 CPU 正在运行的进程
      c->proc = highest_p;
      // 切换到该进程的用户内存空间
      switchuvm(highest_p);
     // 将进程状态设置为 RUNNING
      highest_p->state = RUNNING;

      // 进行上下文切换，保存调度器的状态并切换到进程
      swtch(&(c->scheduler), highest_p->context);
      // 切换回内核虚拟内存
      switchkvm();

      // 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
      c->proc = 0;
    }

    // 释放进程表锁
    release(&ptable.lock);
  }
}
```

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：

```
int
```

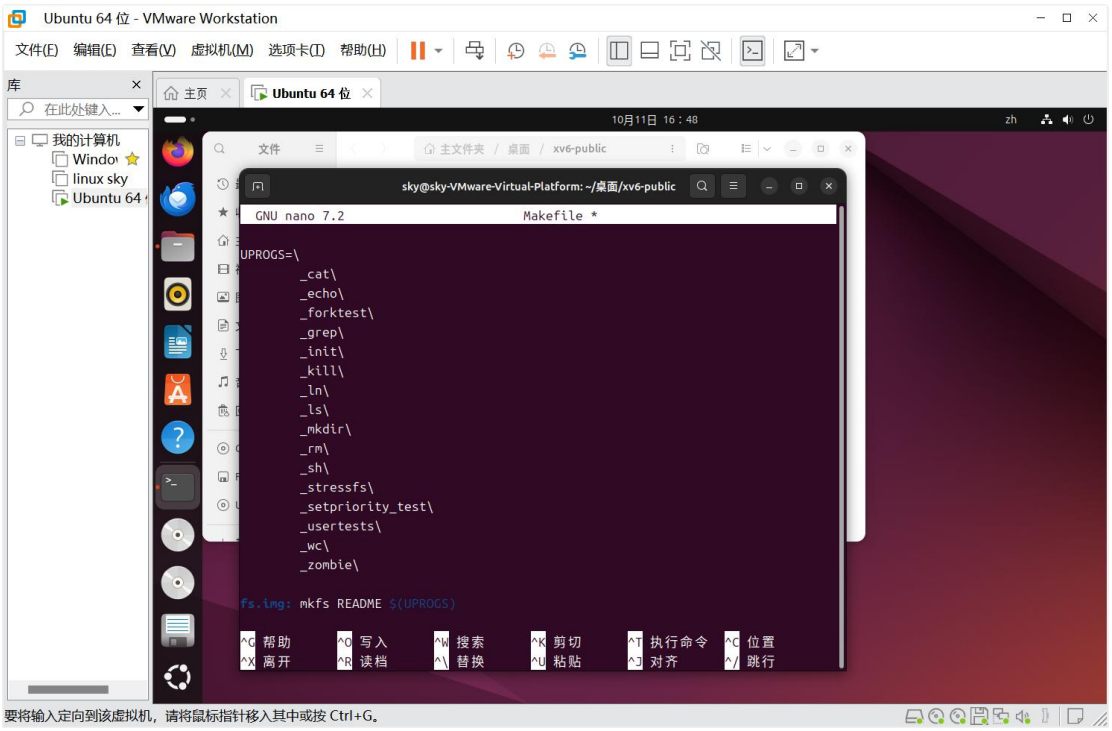
```
sys_setpriority(void)
{
  int pid, priority;

  // 获取传递给系统调用的参数
  if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
    return -1;

  struct proc *p;

  // 获取进程表锁，保证进程表的安全访问
  acquire(&ptable.lock);

  // 遍历进程表，找到目标进程
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->priority = priority;  // 更新进程的优先级
      break;
    }
  }

  // 释放进程表锁
  release(&ptable.lock);

  return 0;
}
```

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。

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main() {
  int pid = fork();  // 创建子进程

  if (pid == 0) {
    // 子进程，设置优先级为 15（较高的优先级）
```

```
      setpriority(getpid(), 15);
      while (1) {
        printf(1, "Child running with priority 15\n");
        sleep(100);
      }
    } else {
      // 父进程，设置优先级为 5（较低的优先级）
      setpriority(getpid(), 5);
      while (1) {
        printf(1, "Parent running with priority 5\n");
        sleep(100);
      }
    }

    return 0;
  }
```

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：

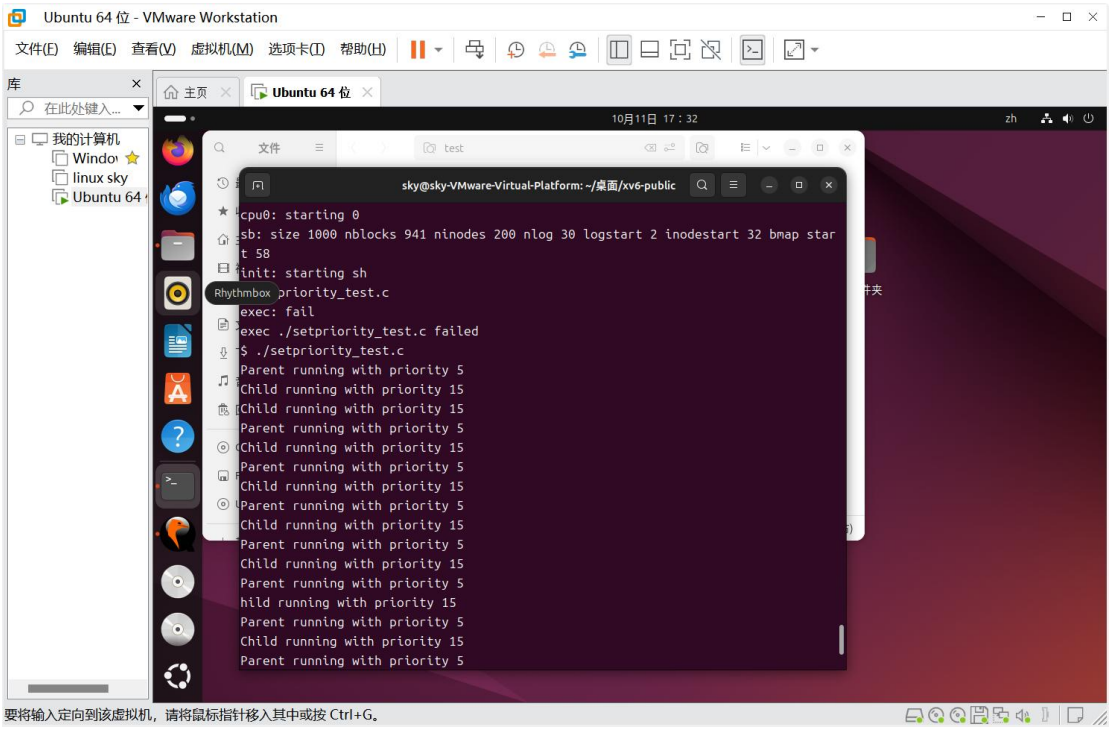
```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct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

```
    struct context *context;     // swtch() here to run process
    void *chan;                  // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;           // Current directory
    char name[16];               // Process name (debugging)

    // Add the priority field
    int priority;                // Process priority for scheduling
};
```

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

```
#include "types.h"  // 确保定义了基础数据类型
#include "spinlock.h"

// 提前声明 struct proc
struct proc;

extern struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

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。
28. 输入 sudo apt-get install qemu-system-i386 来安装 QEMU。
29. 然后输入 make qemu 来运行该操作系统
30. 接下来输入./set_priority.c



31.
32.