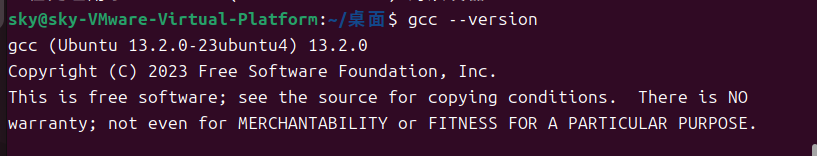
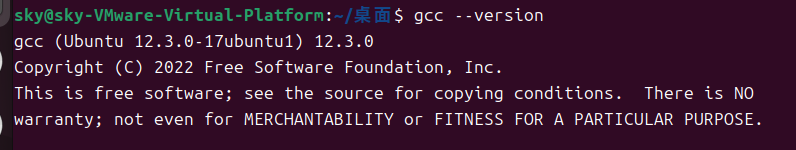
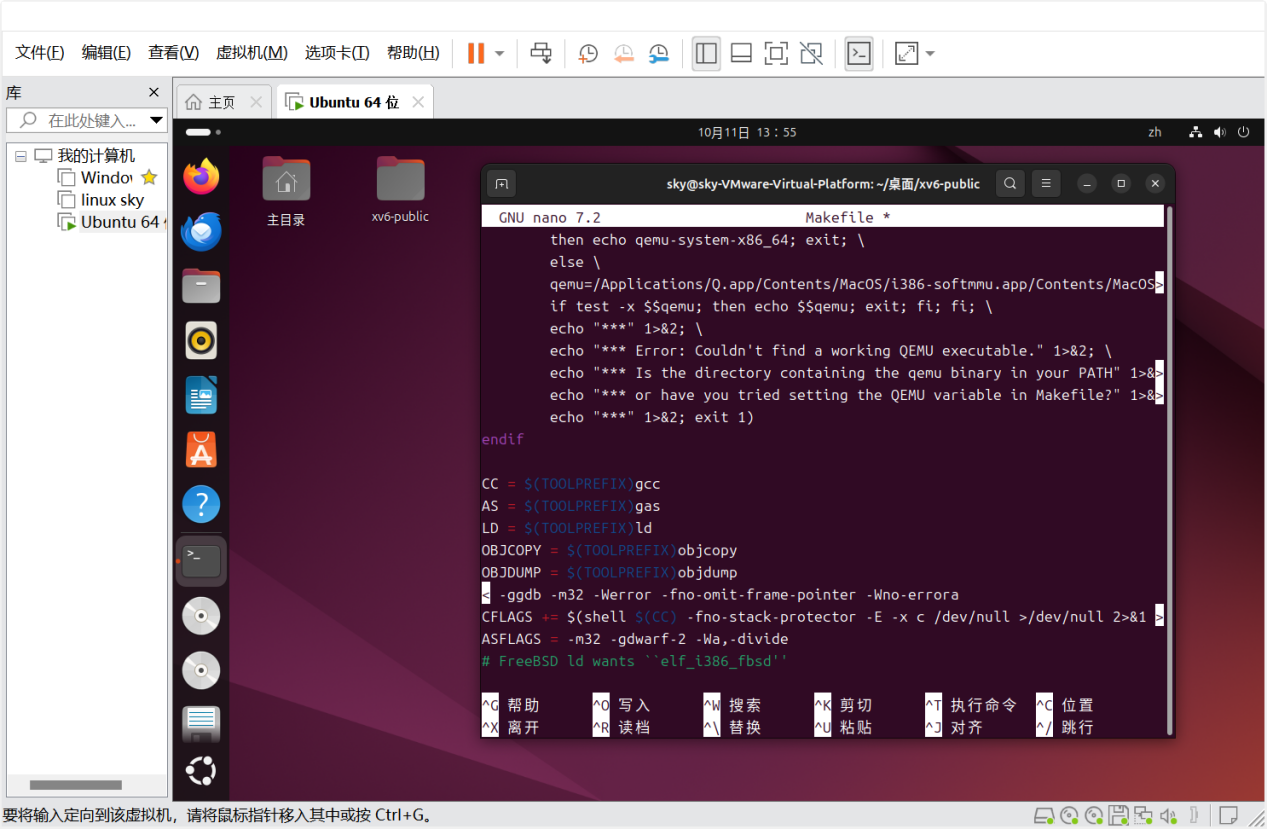
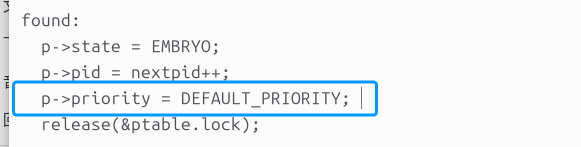
# Base Function

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  
   
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   https://www.reddit.com/r/osdev/comments/16g4mg3/xv6\_make\_issue/?rdt=34243这里发现答案：The x86 version of xv6 is no longer maintained.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  
   
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. for(;;){  *// 死循环，调度器不断地运行* 8. sti();  *// 使能中断，以便处理外部中断请求* 9. *// 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* 10. 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* 11. *// 遍历进程表，寻找处于 RUNNABLE 状态的进程* 12. *// 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* 13. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 14. if(p->state != RUNNABLE)  *// 只调度处于 RUNNABLE 状态的进程* 15. continue; 16. *// 找到 RUNNABLE 进程后，准备调度该进程* 17. c->proc = p;  *// 将当前 CPU 的进程指针指向该进程* 18. 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。* 19. p->state = RUNNING;  *// 将该进程的状态设置为 RUNNING* 20. *// 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。* 21. swtch(&(c->scheduler), p->context); 22. switchkvm();  *// 恢复内核虚拟内存空间* 23. *// 进程运行结束，重置当前 CPU 的进程指针* 24. c->proc = 0; 25. } 26. 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* 27. } 28. } |

1. 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. struct context { 12. uint edi; *//edi 是通用寄存器之一，用于存储数据或指针。* 13. uint esi; *// esi 也是一个通用寄存器，常用于数据操作。* 14. uint ebx; *// ebx 是一个基址寄存器，用于存储基地址或指针。* 15. uint ebp; *// ebp 通常用作栈帧指针，指向当前函数调用的栈帧。* 16. uint eip; *// eip 是指令指针，指向下一条将要执行的指令。* 17. }; 18. enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE }; *// UNUSED、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、* 19. struct proc { 20. uint sz;                     *// 进程内存的大小（以字节为单位）* 21. pde\_t\* pgdir;                *// 页表地址* 22. char \*kstack;                *// 内核栈的底部地址* 23. enum procstate state;        *// 进程的状态* 24. int pid;                     *// 进程的唯一标识符* 25. struct proc \*parent;         *// 父进程* 26. struct trapframe \*tf;        *// 当前系统调用的陷阱帧* 27. struct context \*context;     *// 用于运行进程的上下文（用于调度器切换）* 28. void \*chan;                  *// 如果非零，进程正在该通道上休眠* 29. int killed;                  *// 如果非零，表示该进程已被杀死* 30. struct file \*ofile[NOFILE];  *// 进程打开的文件数组* 31. struct inode \*cwd;           *// 当前工作目录的指针* 32. char name[16];               *// 进程的名称（用于调试）* 33. }; |

1. 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
2. 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。  
   
3. 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：

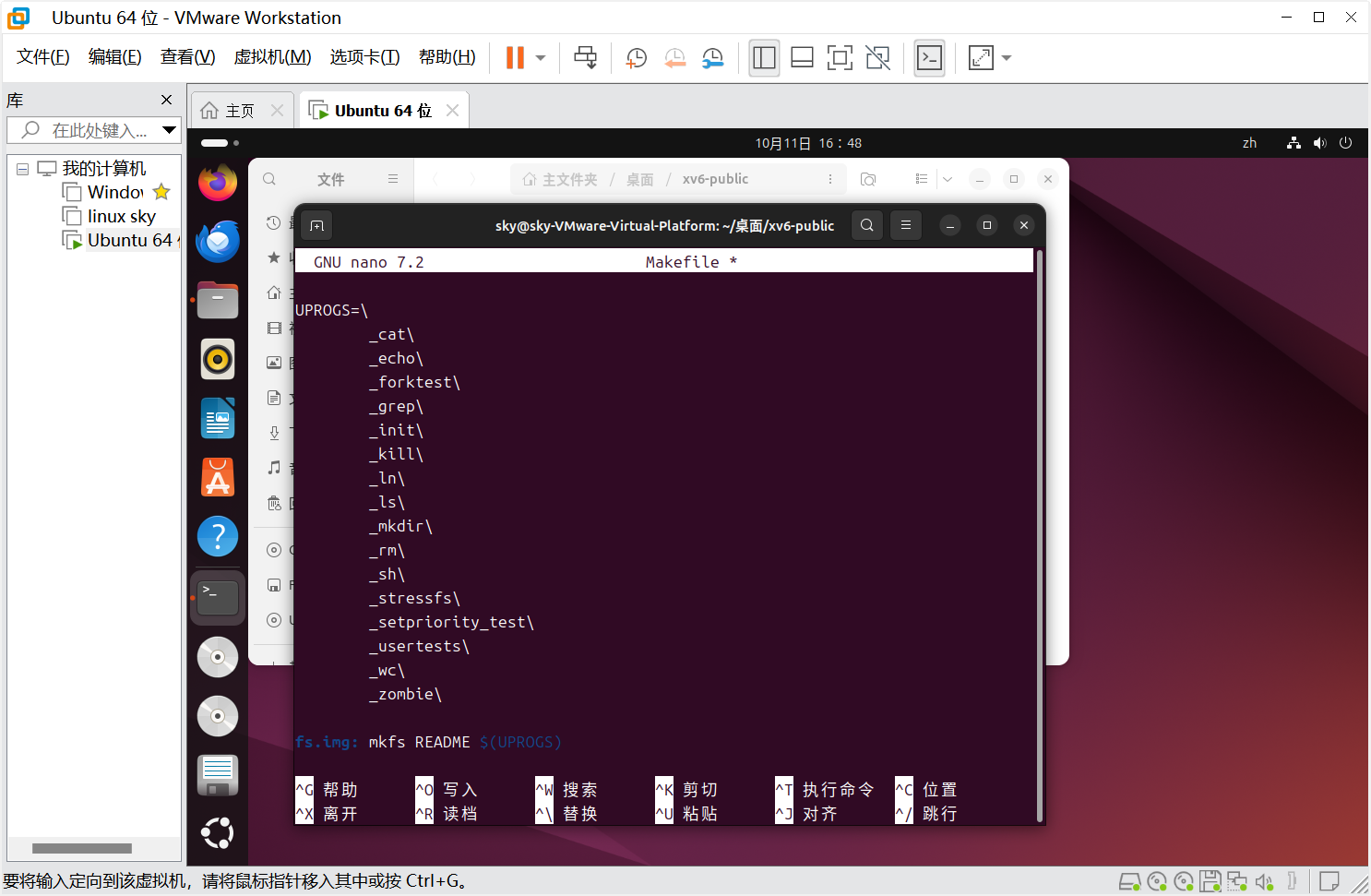
|  |
| --- |
| 1. void 2. scheduler(void) 3. { 4. struct proc \*p; 5. struct proc \*highest\_p;    *// 用来记录优先级最高的进程* 6. struct cpu \*c = mycpu(); 7. c->proc = 0; 8. for(;;){ 9. *// 启用中断* 10. sti(); 11. *// 锁住进程表，确保遍历时不受干扰* 12. acquire(&ptable.lock); 13. *// 初始化为 NULL，表示尚未找到优先级最高的进程* 14. highest\_p = 0; 15. *// 遍历进程表，寻找优先级最高的进程* 16. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 17. *// 跳过非 RUNNABLE 状态的进程* 18. if(p->state != RUNNABLE) 19. continue; 20. *// 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* 21. if (highest\_p == 0 || p->priority > highest\_p->priority) { 22. *// 更新为优先级最高的进程* 23. highest\_p = p; 24. } 25. } 26. *// 如果找到了优先级最高的进程，进行进程切换* 27. if (highest\_p != 0) { 28. *// 切换到优先级最高的进程* 29. *// 设置当前 CPU 正在运行的进程* 30. c->proc = highest\_p; 31. *// 切换到该进程的用户内存空间* 32. switchuvm(highest\_p); 33. *// 将进程状态设置为 RUNNING* 34. highest\_p->state = RUNNING; 35. *// 进行上下文切换，保存调度器的状态并切换到进程* 36. swtch(&(c->scheduler), highest\_p->context); 37. *// 切换回内核虚拟内存* 38. switchkvm(); 39. *// 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* 40. c->proc = 0; 41. } 42. *// 释放进程表锁* 43. release(&ptable.lock); 44. } 45. } |

1. 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。
2. 在syscall.h末增加对于优先级调用的定义:#define SYS\_setpriority 22
3. 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; 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. struct proc \*p; 10. *// 获取进程表锁，保证进程表的安全访问* 11. acquire(&ptable.lock); 13. *// 遍历进程表，找到目标进程* 14. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 15. if(p->pid == pid){ 16. p->priority = priority;  *// 更新进程的优先级* 17. break; 18. } 19. } 21. *// 释放进程表锁* 22. release(&ptable.lock); 23. return 0; 24. } |

1. 在usys.S文件末尾添加SYSCALL(setpriority)来添加系统调用的条目。
2. 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。
3. 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);
4. 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,
5. 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. int main() { 5. int pid = fork();  *// 创建子进程* 6. if (pid == 0) { 7. *// 子进程，设置优先级为 15（较高的优先级）* 8. setpriority(getpid(), 15); 9. while (1) { 10. printf(1, "Child running with priority 15\n"); 11. sleep(100); 12. } 13. } else { 14. *// 父进程，设置优先级为 5（较低的优先级）* 15. setpriority(getpid(), 5); 16. while (1) { 17. printf(1, "Parent running with priority 5\n"); 18. sleep(100); 19. } 20. } 21. return 0; 22. } |

1. 在文件中编辑Makefile，在UPROGS中添加setpriority\_test.c进程  
   
2. 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* 9. struct context \*context;     *// swtch() here to run process* 10. void \*chan;                  *// If non-zero, sleeping on chan* 11. int killed;                  *// If non-zero, have been killed* 12. struct file \*ofile[NOFILE];  *// Open files* 13. struct inode \*cwd;           *// Current directory* 14. char name[16];               *// Process name (debugging)* 16. *// Add the priority field* 17. int priority;                *// Process priority for scheduling* 18. }; |

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

|  |
| --- |
| 1. #include "types.h"  *// 确保定义了基础数据类型* 2. #include "spinlock.h" 3. *// 提前声明 struct proc* 4. struct proc; 5. extern struct { 6. struct spinlock lock; 7. struct proc proc[NPROC]; 8. } ptable; |

1. 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。
2. 输入sudo apt-get install qemu-system-i386来安装QEMU。
3. 然后输入make qemu来运行该操作系统
4. 接下来输入./set\_priority.c  
   