目录

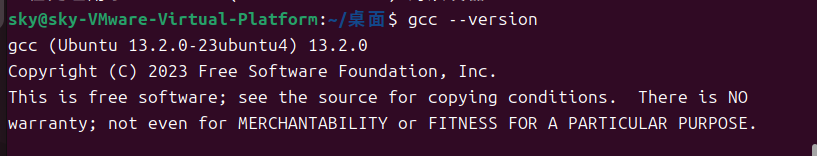
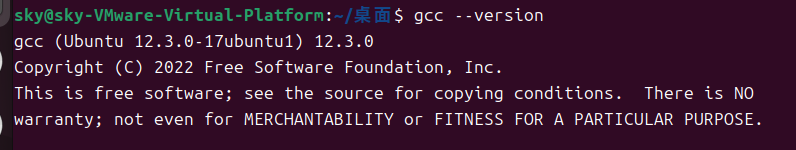
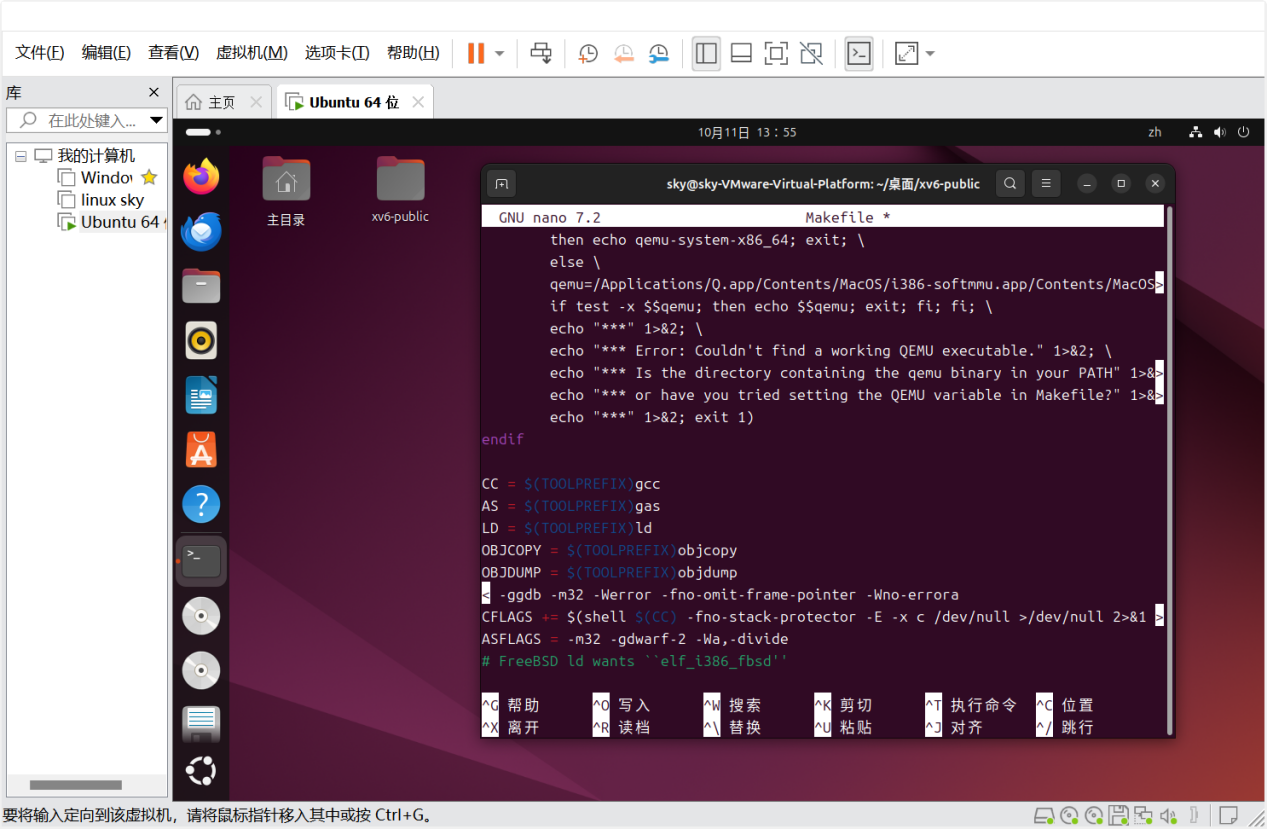
[Base Function 1](#_Toc12362)

[Item 1 9](#_Toc31973)

[Item 2 14](#_Toc15811)

[Item 3 20](#_Toc24036)

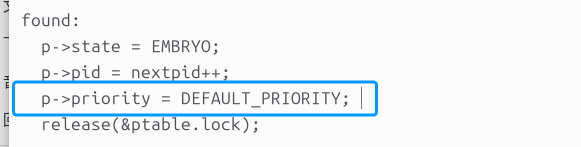
# Base Function

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   https://www.reddit.com/r/osdev/comments/16g4mg3/xv6\_make\_issue/?rdt=34243这里发现答案：The x86 version of xv6 is no longer maintained.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：

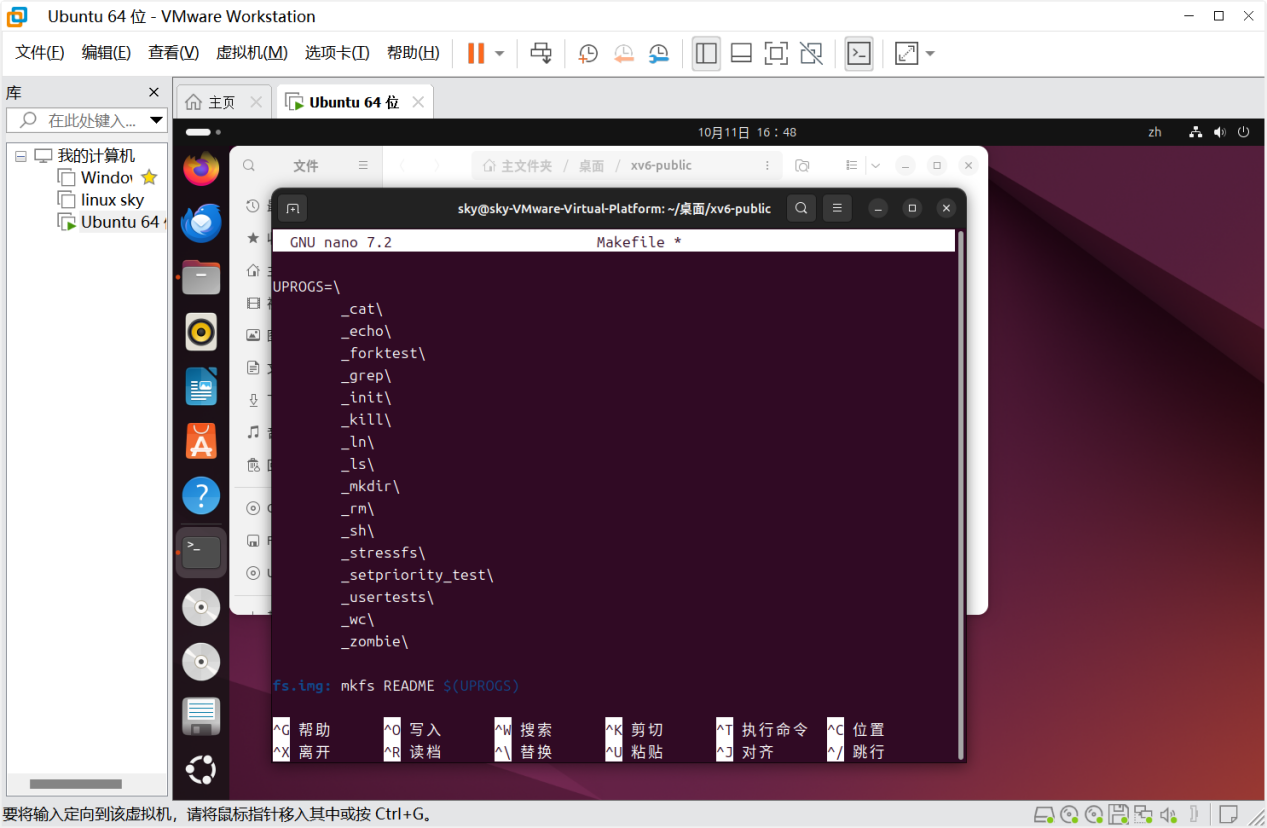
|  |
| --- |
| 1. void 2. scheduler(void) 3. { 4. struct proc \*p; 5. struct proc \*highest\_p;    *// 用来记录优先级最高的进程* 6. struct cpu \*c = mycpu(); 7. c->proc = 0; 8. for(;;){ 9. *// 启用中断* 10. sti(); 11. *// 锁住进程表，确保遍历时不受干扰* 12. acquire(&ptable.lock); 13. *// 初始化为 NULL，表示尚未找到优先级最高的进程* 14. highest\_p =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

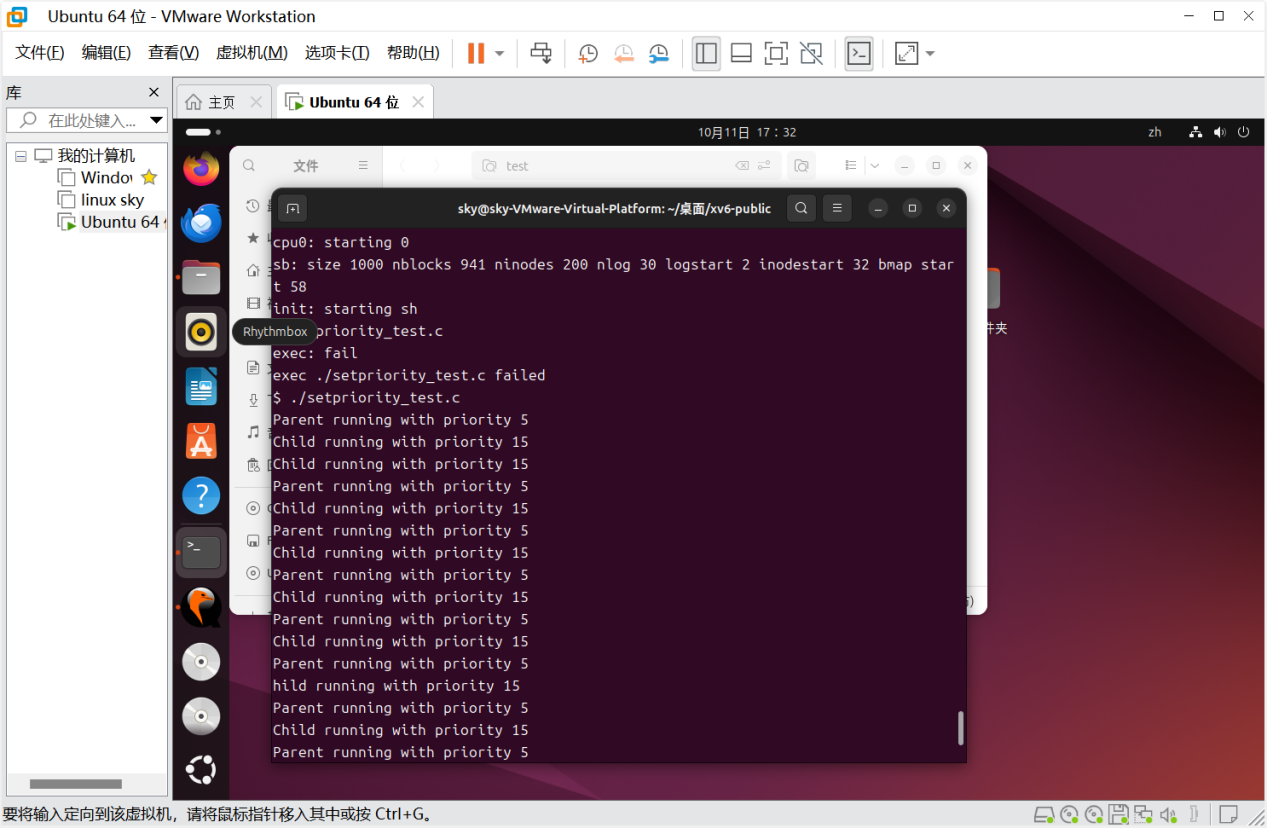
|  |
| --- |
| 1. #include "types.h" 2. #include "stat.h" 3. #include "user.h" 4. int main() { 5. int pid = fork();  *// 创建子进程* 6. if (pid == 0) { 7. *// 子进程，设置优先级为 15（较高的优先级）* 8. setpriority(getpid(), 15); 9. while (1) { 10. printf(1, "Child running with priority 15\n"); 11. sleep(100); 12. } 13. } 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./setpriority\_test.c  
   

# 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* |

1. 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 |

1. 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. for(;;){ 7. *// 使当前处理器支持中断* 8. sti(); 9. *// 遍历进程表，查找可运行的进程* 10. acquire(&ptable.lock); 11. highest\_p = 0; 12. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) { 13. if(p->state != RUNNABLE) 14. continue; 15. *// 对等待的进程进行老化，增加优先级* 16. if (p != highest\_p) { 17. p->aging\_ticks++; 18. if (p->aging\_ticks >= AGING\_THRESHOLD) { 19. if (p->priority < MAX\_PRIORITY) { 20. p->priority++; 21. printf(1, "Process %d: aging - priority increased to %d\n", p->pid, p->priority); 22. } 23. p->aging\_ticks = 0; 24. } 25. } 26. *// 找到当前优先级最高的进程* 27. if(highest\_p == 0 || p->priority > highest\_p->priority) { 28. highest\_p = p; 29. } 30. } 31. if(highest\_p != 0) { 32. *// 切换到优先级最高的进程* 33. c->proc = highest\_p; 34. switchuvm(highest\_p); 35. highest\_p->state = RUNNING; 36. printf(1, "Process %d is running with priority %d\n", highest\_p->pid, highest\_p->priority); 37. *// 运行时降低优先级，避免过度占用CPU资源* 38. if (highest\_p->priority > MIN\_PRIORITY) { 39. highest\_p->priority--; 40. printf(1, "Process %d: running - priority decreased to %d\n", highest\_p->pid, highest\_p->priority); 41. } 42. swtch(&(c->scheduler), highest\_p->context); 43. switchkvm(); 44. *// 当前进程完成一个调度周期后，将其从处理器释放* 45. c->proc = 0; 46. } 47. release(&ptable.lock); 48. } 49. } |

1. 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. if(lk == 0) 6. panic("sleep without lk"); 7. *// 获取 ptable.lock 并释放当前锁* 8. if(lk != &ptable.lock) { 9. acquire(&ptable.lock); 10. release(lk); 11. } 12. *// 设置进程状态为 SLEEPING 并进入调度* 13. p->chan = chan; 14. p->state = SLEEPING; 15. p->aging\_ticks = 0;  *// 进入睡眠时重置老化计时器* 16. sched(); 18. *// 清空 chan 并重新获取原始锁* 19. p->chan = 0; 20. if(lk != &ptable.lock) { 21. release(&ptable.lock); 22. acquire(lk); 23. } 24. } 25. void wakeup(void \*chan) { 26. struct proc \*p; 27. acquire(&ptable.lock); 28. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) 29. if(p->state == SLEEPING && p->chan == chan) { 30. p->state = RUNNABLE; 31. p->aging\_ticks = 0;  *// 唤醒时重置老化计时器* 32. } 33. release(&ptable.lock); 34. } |

1. 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; 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. struct proc \*p; 10. acquire(&ptable.lock); 11. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 12. if(p->pid == pid) { 13. p->priority = priority; 14. break; 15. } 16. } 17. release(&ptable.lock); 18. return 0; 19. } |

1. 在sysproc.c 中声明 sys\_setpriority——新增extern int sys\_setpriority(void);
2. 在syscall.h 中添加对setpriority 分配系统调用的编号：#define SYS\_setpriority 22
3. 在user.h处声明setpriority:int setpriority(int pid, int priority);
4. 接下来，更新setpriority\_test.c函数为:

|  |
| --- |
| 1. #include "types.h" 2. #include "stat.h" 3. #include "user.h" 4. #define NUM\_PROCS 5  *// 定义子进程数量* 5. #define RUN\_TIME 100 *// 每个子进程运行的时间片* 6. void do\_work(int priority) { 7. int i; 8. setpriority(getpid(), priority);  *// 设置进程优先级* 9. printf(1, "Process %d started with priority %d\n", getpid(), priority); 11. for (i = 0; i < RUN\_TIME; i++) { 12. printf(1, "Process %d is running with priority %d, iteration %d\n", getpid(), priority, i); 13. sleep(10);  *// 模拟一些工作，避免独占CPU* 14. } 15. printf(1, "Process %d finished with priority %d\n", getpid(), priority); 16. } 17. int main(void) { 18. int pids[NUM\_PROCS];  *// 保存子进程的pid* 19. int initial\_priority = 5; 20. *// 创建多个子进程，每个子进程设置不同的初始优先级* 21. for (int i = 0; i < NUM\_PROCS; i++) { 22. pids[i] = fork(); 23. if (pids[i] == 0) {  *// 子进程* 24. do\_work(initial\_priority + i \* 5);  *// 设置初始优先级，逐渐增高* 25. exit();  *// 结束子进程* 26. } 27. } 28. *// 父进程负责监控每个子进程* 29. for (int i = 0; i < NUM\_PROCS; i++) { 30. wait();  *// 等待每个子进程结束* 31. printf(1, "Process %d exited\n", pids[i]); 32. } 33. *// 父进程结束* 34. printf(1, "All child processes have exited.\n"); 35. exit(); 36. } |

1. 先输入make,然后输入输入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：

|  |
| --- |
| IMG_256 **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* |

1. 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. if(holding(lk)) 4. panic("acquire"); 5. *// 获取当前持有锁的进程* 6. struct proc \*holder = myproc()->lock\_holding; 7. *// 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* 8. if (lk->locked && holder && holder->priority < myproc()->priority) { 9. *// 记录持有者的原始优先级，并捐献优先级* 10. holder->donated\_priority = myproc()->priority; 11. cprintf("Priority donation: process %d donates priority to process %d\n", myproc()->pid, holder->pid); 12. } 13. *// 获取锁* 14. while(xchg(&lk->locked, 1) != 0) 15. ; 16. \_\_sync\_synchronize(); 17. *// 当前进程持有锁* 18. myproc()->lock\_holding = lk; 19. } |

1. 接着修改release函数

|  |
| --- |
| 1. void release(struct spinlock \*lk) { 2. if(!holding(lk)) 3. panic("release"); 4. struct proc \*holder = myproc(); 5. *// 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* 6. if (holder->donated\_priority != 0) { 7. holder->priority = holder->donated\_priority; 8. holder->donated\_priority = 0;  *// 清除捐献的优先级* 9. cprintf("Priority restored: process %d restores its original priority\n", holder->pid); 10. } 11. *// 清除锁的持有状态* 12. holder->lock\_holding = 0; 13. \_\_sync\_synchronize(); 14. xchg(&lk->locked, 0); 15. popcli(); 16. } |

1. 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. for(;;){ 6. sti(); 7. acquire(&ptable.lock); 8. struct proc \*highest\_p = 0; 9. *// 查找具有最高优先级的进程* 10. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) { 11. if(p->state != RUNNABLE) 12. continue; 13. *// 使用捐献的优先级（如果有）* 14. int effective\_priority = (p->donated\_priority != 0) ? p->donated\_priority : p->priority; 15. if(highest\_p == 0 || effective\_priority > highest\_p->priority) { 16. highest\_p = p; 17. } 18. } 19. *// 切换到具有最高优先级的进程* 20. if(highest\_p != 0) { 21. c->proc = highest\_p; 22. switchuvm(highest\_p); 23. highest\_p->state = RUNNING; 24. 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); 25. swtch(&(c->scheduler), highest\_p->context); 26. switchkvm(); 27. c->proc = 0; 28. } 29. release(&ptable.lock); 30. } 31. } |

1. 接着修改sleep和wakeup函数
2. sleep函数修改如下：

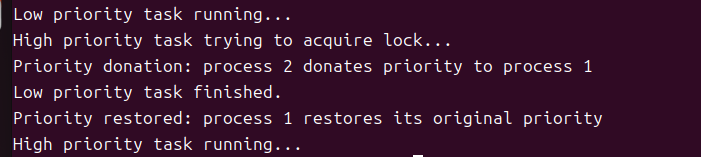
|  |
| --- |
| 1. void sleep(void \*chan, struct spinlock \*lk) { 2. struct proc \*p = myproc(); 3. if(p == 0) 4. panic("sleep"); 5. if(lk == 0) 6. panic("sleep without lk"); 7. *// 获取 ptable.lock 并释放当前锁* 8. if(lk != &ptable.lock) { 9. acquire(&ptable.lock); 10. release(lk); 11. } 12. *// 设置进程状态为 SLEEPING* 13. p->chan = chan; 14. p->state = SLEEPING; 15. p->aging\_ticks = 0;  *// 重置老化计时器* 16. sched(); 17. *// 清空 chan 并重新获取原始锁* 18. p->chan = 0; 19. if(lk != &ptable.lock) { 20. release(&ptable.lock); 21. acquire(lk); 22. } 23. *// 如果进程有捐献的优先级，恢复原始优先级* 24. if (p->donated\_priority != 0) { 25. p->priority = p->donated\_priority; 26. p->donated\_priority = 0; 27. } 28. } |

1. wakeup函数修改如下：

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

1. 然后编辑测试函数setpriority\_test.c：

|  |
| --- |
| 1. #include "types.h" 2. #include "stat.h" 3. #include "user.h" 4. int lock = 0; *// 锁状态，0表示未锁定，1表示锁定* 5. *// 模拟获取锁* 6. void acquire\_lock(int \*lk) { 7. while (\*lk == 1) { 8. *// 自旋等待锁释放* 9. } 10. \*lk = 1;  *// 获取锁* 11. } 12. *// 模拟释放锁* 13. void release\_lock(int \*lk) { 14. \*lk = 0;  *// 释放锁* 15. } 16. *// 模拟低优先级任务* 17. void low\_priority\_task() { 18. acquire\_lock(&lock);  *// 获取锁* 19. printf(1, "Low priority task running...\n"); 20. *// 模拟延迟操作* 21. for (int i = 0; i < 100000000; i++) { 22. asm volatile("nop");  *// 占用 CPU* 23. } 24. printf(1, "Low priority task finished.\n"); 25. release\_lock(&lock);  *// 释放锁* 26. } 27. *// 模拟高优先级任务* 28. void high\_priority\_task() { 29. printf(1, "High priority task trying to acquire lock...\n"); 30. acquire\_lock(&lock);  *// 尝试获取锁* 31. printf(1, "High priority task running...\n"); 32. release\_lock(&lock);  *// 释放锁* 33. } 34. int main() { 35. int pid\_low = fork(); 36. if (pid\_low == 0) { 37. setpriority(getpid(), 5);  *// 低优先级进程* 38. low\_priority\_task(); 39. exit(); 40. } 41. *// 等待低优先级进程开始执行* 42. sleep(50); 43. int pid\_high = fork(); 44. if (pid\_high == 0) { 45. setpriority(getpid(), 20);  *// 高优先级进程* 46. high\_priority\_task(); 47. exit(); 48. } 49. wait();  *// 等待子进程* 50. wait();  *// 等待子进程* 51. exit(); 52. } |

1. 先输入make,然后输入输入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：
2. 

|  |
| --- |
| **"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sturct 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* |

1. 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. acquire(&ptable.lock); 7. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) 8. if(p->state == UNUSED) 9. goto found; 10. release(&ptable.lock); 11. return 0; 12. found: 13. p->state = EMBRYO; 14. p->pid = nextpid++; 15. p->priority = DEFAULT\_PRIORITY;  *// 分配默认优先级* 17. *// 初始化调度性能字段* 18. p->start\_time = ticks;   *// 设置进程开始时间* 19. p->end\_time = 0; 20. p->wait\_time = 0; 21. p->run\_time = 0; 22. p->last\_scheduled = ticks;  *// 进程刚创建时的时间* 24. release(&ptable.lock); 25. *// Allocate kernel stack.* 26. if((p->kstack = kalloc()) == 0){ 27. p->state = UNUSED; 28. return 0; 29. } 30. sp = p->kstack + KSTACKSIZE; 31. *// Leave room for trap frame.* 32. sp -= sizeof \*p->tf; 33. p->tf = (struct trapframe\*)sp; 34. sp -= 4; 35. \*(uint\*)sp = (uint)trapret; 36. sp -= sizeof \*p->context; 37. p->context = (struct context\*)sp; 38. memset(p->context, 0, sizeof \*p->context); 39. p->context->eip = (uint)forkret; 40. return p; 41. } |

1. 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. for(;;){ 9. sti(); 10. acquire(&ptable.lock); 11. highest\_p = 0; 12. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 13. if(p->state != RUNNABLE) 14. continue; 15. *// 选择优先级最高的进程运行* 16. if(highest\_p == 0 || p->priority > highest\_p->priority) { 17. highest\_p = p; 18. } 19. } 20. if(highest\_p != 0) { 21. *// 更新等待时间* 22. highest\_p->wait\_time += (ticks - highest\_p->last\_scheduled); 24. c->proc = highest\_p; 25. switchuvm(highest\_p); 26. highest\_p->state = RUNNING; 27. swtch(&(c->scheduler), highest\_p->context); 28. switchkvm(); 29. *// 更新运行时间* 30. highest\_p->run\_time += (ticks - highest\_p->last\_scheduled); 31. *// 记录上次调度时间* 32. highest\_p->last\_scheduled = ticks; 33. c->proc = 0; 34. } 35. release(&ptable.lock); 36. } 37. } |

1. 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. if(curproc == initproc) 8. panic("init exiting"); 9. *// 关闭所有打开的文件* 10. for(fd = 0; fd < NOFILE; fd++){ 11. if(curproc->ofile[fd]){ 12. fileclose(curproc->ofile[fd]); 13. curproc->ofile[fd] = 0; 14. } 15. } 16. begin\_op(); 17. iput(curproc->cwd); 18. end\_op(); 19. curproc->cwd = 0; 20. acquire(&ptable.lock); 21. wakeup1(curproc->parent); 22. *// 将遗弃的子进程传递给init* 23. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 24. if(p->parent == curproc){ 25. p->parent = initproc; 26. if(p->state == ZOMBIE) 27. wakeup1(initproc); 28. } 29. } 30. *// 记录进程的结束时间* 31. curproc->end\_time = ticks; 32. *// 计算并打印调度性能* 33. int turnaround\_time = curproc->end\_time - curproc->start\_time; 34. cprintf("Process %d Turnaround time: %d, Wait time: %d, Run time: %d\n", 35. curproc->pid, turnaround\_time, curproc->wait\_time, curproc->run\_time); 36. *// 进入调度器，永不返回* 37. curproc->state = ZOMBIE; 38. sched(); 39. panic("zombie exit"); 40. } |

1. 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. int 3. sys\_get\_sched\_performance(void) 4. { 5. int pid; 7. *// 获取传递的 pid 参数* 8. if(argint(0, &pid) < 0) 9. return -1; 11. struct proc \*p; 12. acquire(&ptable.lock); 14. *// 遍历进程表，找到匹配 pid 的进程* 15. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 16. if(p->pid == pid){ 17. int turnaround\_time = p->end\_time - p->start\_time; 18. int wait\_time = p->wait\_time; 19. int run\_time = p->run\_time; 20. *// 打印调度性能信息* 21. cprintf("Process %d - Turnaround time: %d, Wait time: %d, Run time: %d\n", 22. p->pid, turnaround\_time, wait\_time, run\_time); 23. release(&ptable.lock); 24. return 0; 25. } 26. } 28. release(&ptable.lock); 29. return -1;  *// 如果没有找到匹配的进程，返回 -1* 30. } |

1. 在 syscall.h 中定义系统调用编号：#define SYS\_get\_sched\_performance 23
2. 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. extern int sys\_get\_sched\_performance(void);  *// 声明新的系统调用* 6. static int (\*syscalls[])(void) = { 7. [SYS\_fork]    sys\_fork, 8. [SYS\_exit]    sys\_exit, 9. [SYS\_wait]    sys\_wait, 10. *// 其他系统调用的函数指针...* 11. [SYS\_get\_sched\_performance] sys\_get\_sched\_performance,  *// 注册新的系统调用* 12. }; |

1. 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)
2. 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);
3. 编写setpriority\_test.c:

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

1. 先输入make,然后输入输入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
2. 