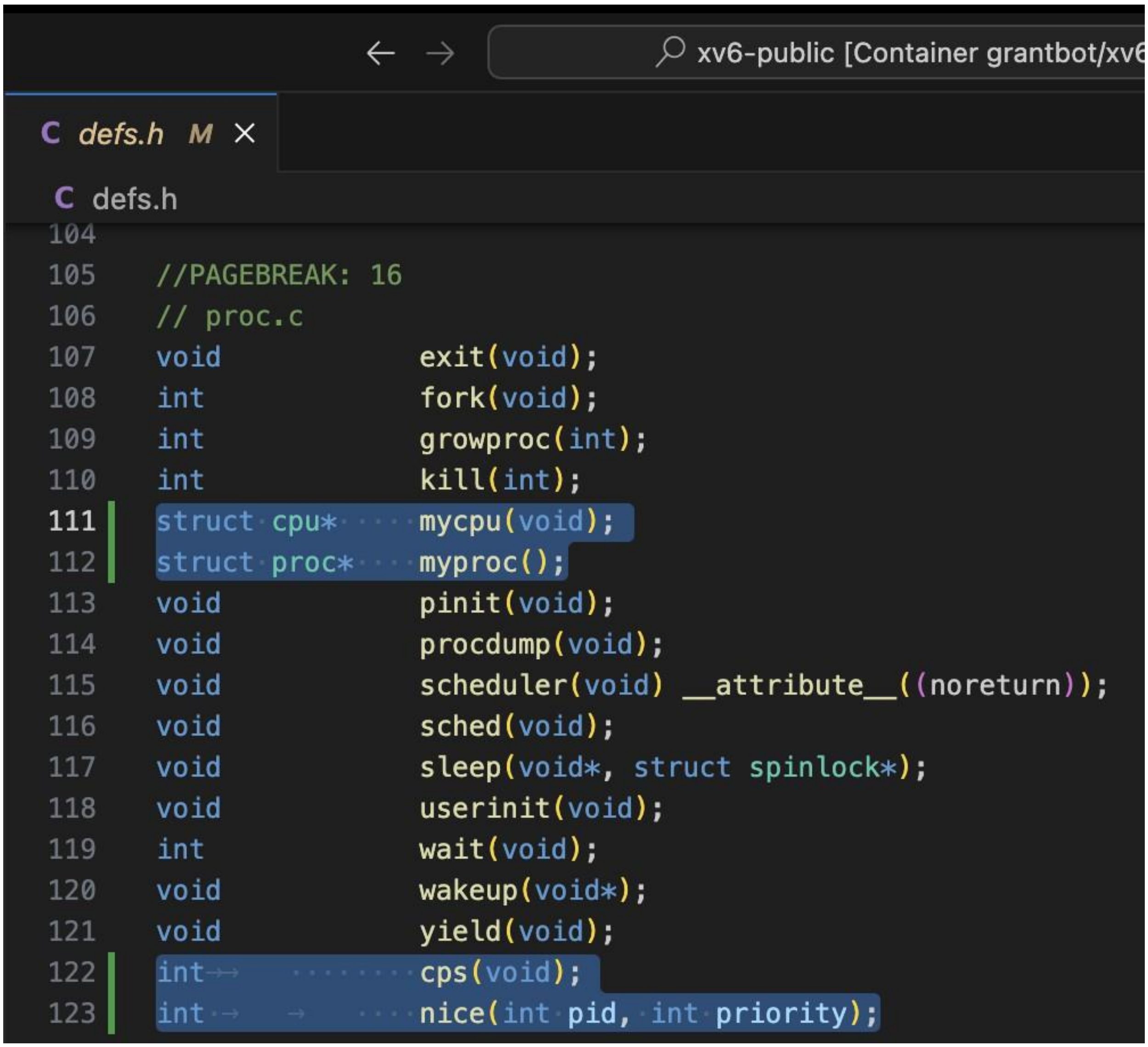
Explanation

Task 1 – Nice System Call Implementation

Changes in files:

Defs.h




```c
//PAGEBREAK: 16
// proc.c
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             cps(void);
int             nice(int pid, int priority);
```

```c
int             lapicid(void);
```

Added function definitons in defs.h

lapic.c

```c
#define ERROR    (0x0370/4)   // Local Vector Table 3 (ERROR)
  #define MASKED      0x00010000   // Interrupt masked
#define TICR     (0x0380/4)   // Timer Initial Count
#define TCCR     (0x0390/4)   // Timer Current Count
#define TDCR     (0x03E0/4)   // Timer Divide Configuration

volatile uint *lapic;  // Initialized in mp.c

static void
lapicw(int index, int value)
{
  lapic[index] = value;
  lapic[ID];  // wait for write to finish, by reading
}
//PAGEBREAK!

int
lapicid(void)
{
  if (!lapic)
    return 0;
  return lapic[ID] >> 24;
}

```

The above function is used in mycpu() function.

MAKEFILE

Makefile M ×

Makefile

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_ps\
	_nice\
	_test1\
	_test2\
	_test3\
	_test1_nice\
	_test2_nice\
	_test3_nice\
```

Makefile M ×

Makefile

```
qemu-nox-gdb: fs.img xv6.img .gdbinit
```

```
# CUT HERE
# prepare dist for students
# after running make dist, probably want to
# rename it to rev0 or rev1 or so on and then
# check in that version.

EXTRA=\
	mkfs Click to collapse the range. c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
	ps.c nice.c test1.c test2.c test3.c test1_nice.c test2_nice.c test3_nice.c\
```

Added files under UPRODGS and EXTRAS to get them into simulation env.

nice.c

nice.c

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
#include "param.h"

int
main(int argc, char *argv[])
{
  int priority, pid=-1;
  // old_priority=-1;
  if(argc < 2 || argc > 3){
    printf(2,"Usage: nice pid priority\n");
    exit();
  }
  if(argc==2){
    //change the priority of the current process to the priority given in prompt
    priority = atoi(argv[1]);
    if (priority < 0 || priority > 5){
      printf(2,"Invalid priority (0-5)!\n");
      exit();
    }
    int current_pid=getpid();
    int res=nice(current_pid,priority);
    printf(1,"%d %d\n",res/MOD,res%MOD);
    exit();
  }
  else{
    //if three arguments are given
    priority = atoi(argv[2]);
    if (priority < 0 || priority > 5){
      printf(2,"Invalid priority (0-5)!\n");
      exit();
    }
    pid = atoi(argv[1]);
    int res=nice(pid,priority);
    printf(1,"%d %d\n",res/MOD,res%MOD);
    exit();
  }
}
```

The program modifies the priority of a process in a Unix-like system. If only a priority is given, it changes the current process's priority. If a process ID (pid) and priority are provided, it changes the priority of that specified process. It validates inputs to ensure the priority is within the acceptable range (0-5) and calls a nice() function to perform the change.

params.h

param.h

```c
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
#define MAXARG       32  // max exec arguments
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       1000  // size of file system in blocks
#define PLEVELS      5   // Levels of priority
#define MOD          65  // to extract pid and priority
```

PLEVELS is the priority levels and MOD is used to get PID and old value using dividend quotient method.

proc.c

```c
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 3;
```

When a process is created it is by default given a priority of 3.

```
int cps()
{
  struct proc *p;
  //Enables interrupts on this processor.
  sti();

  //Loop over process table looking for process with pid.
  acquire(&ptable.lock);
  cprintf("name \t pid \t state \t priority \n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == SLEEPING)
      cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
    else if(p->state == RUNNING)
      cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
    else if(p->state == RUNNABLE)
      cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
  }
  release(&ptable.lock);
  return 22;
}
```

Created a ps system call to list SLEEPING, RUNNING and RUNNABLE process.

```c
int
nice(int pid, int priority)
{
  if(priority <1 || priority>5){
    cprintf("Error: Priority not within range(1-5)\n");
    exit();
  }
  struct proc *p;
  int found=0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      found=1;
      break;
    }
  }
  if(!found){
      release(&ptable.lock);
      cprintf("No process with Pid=%d\n",pid);
      exit();
  }
  release(&ptable.lock);
  int old_priority = p->priority;
  p->priority=priority;

  remove_process_from_queue(p,old_priority);
  add_process_to_queue(p,priority);
  return (pid*MOD+old_priority);
}
```

The nice() function changes the priority of a process identified by pid to a new priority (1-5). It checks if the priority is valid, searches for the process in the process table, updates its priority, and adjusts the process's position in the scheduling queues accordingly. If the process is not found, it prints an error and exits.

```c
struct node priority_q[PLEVELS][NPROC]; // A two dimensional array with 5 priority rows and 64 process columns
int marker[PLEVELS] = {0,0,0,0,0};           // 0 because initially no process inside the priority queue

int add_process_to_queue(struct proc *p, int pr)
{
    int i;
    for(i=0;i<marker[pr];i++){
      if(p->pid == priority_q[pr][i].p->pid){
        return -1;
      }
    }

    priority_q[pr][marker[pr]].p=p;
    marker[pr]++;
    return 1;
}
```

The add_process_to_queue() function adds a process p to a priority queue priority_q at the specified priority level pr. It first checks if the process is already in the queue, returning -1 if found. If not, it adds the process to the queue, increments the marker for that priority level, and returns 1 to indicate success.

```c
int remove_process_from_queue(struct proc *p, int pr)
{
  int found = -1, i;
  for(i=0; i < marker[pr]; i++)
  {
    //if process is found
    if(priority_q[pr][i].p->pid == p->pid)
    {
      found = 1;
      break;
    }
  }
  if(found==-1)
  {
    return found;
  }
  int j;
  //shift process after removing
  for(j = i; j<marker[pr]-1; j++)
  {
    priority_q[pr][j] = priority_q[pr][j+1];
  }

  marker[pr]-=1;
  return found;
}
```

The remove_process_from_queue() function removes a process p from the priority queue at the specified priority level pr. It searches for the process in the queue, and if found, it shifts all subsequent processes one position to the left to maintain the queue order and decrements the marker for that priority level. It returns 1 if the process was found and removed, or -1 if the process was not found.

proc.h

proc.h

```c
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?

  // Cpu-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc;           // The currently-running process.
};

struct node {
  struct proc* p;
  struct node* prev;
  struct node* next;
};

extern struct cpu cpus[NCPU];
extern int ncpu;
extern int add_process_to_queue(struct proc *p, int priority);
extern int remove_remove_process_from_queueproc_from_q(struct proc *p, int priority);
```

Added definitions in proc.h

ps.c

ps.c

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

int main(void){
  cps();
  exit();
}
```

Code for ps system call

syscall.c

Added lines

syscall.c

```c
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_cps(void);
extern int sys_nice(void);
```

```c
[SYS_cps]     sys_cps,
[SYS_nice]    sys_nice,
};
```

Added definitions in syscall.c

Syscall.h

syscall.h M

syscall.h

```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_cps    22
#define SYS_nice   23
```

Changed lines

Gave system call numbers to the ps and nice calls

sysproc.c

C sysproc.c M

C sysproc.c

```c
{
}

// return how many clock tick inte
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

int
sys_cps(void)
{
  return cps();
}

int
sys_nice(void)
{
  int pid, pr;
  if(argint(0, &pid) < 0)
    return -1;
  if(argint(1, &pr) < 0)
    return -1;

  return nice(pid, pr);
}
```

Added process in sysproc

user.h

```
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int cps(void);
int nice(int pid, int priority);
```

Added processes definitions in user.h

usys.S

usys.S

```
#define SYSCALL(name) \
  name: \

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(cps)
SYSCALL(nice)
```

Added lines

Added syscalls.

TEST CASES

1. **test1_nice.c**

M Makefile M   C test1_nice.c U

C test1_nice.c

```c
#include "types.h"
#include "user.h"
#include "param.h"

int main() {
    printf(1, "Test 1: Nice Value\n");
    int i;
    for(i=4;i>1;i=i-2){
      int value = i;
      int res = nice(getpid(),value);
      if (res < 0) {
        printf(1, "Failed: Unable to Change Value\n");
      } else {
        printf(1, "Current PID: %d, Old_Value: %d, New Value: %d\n", res/MOD, res%MOD, value);
      }
    }
    exit();
}
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
init: starting sh
$ test1_nice
Test 1: Nice Value
Current PID: 3, Old_Value: 3, New Value: 4
Current PID: 3, Old_Value: 4, New Value: 2
```

The program tests changing the priority of the current process twice using nice(), and it prints the process ID, old priority, and new priority value or an error if the change fails.

2. test2_nice.c

test2_nice.c

```c
#include "types.h"
#include "user.h"
#include "param.h"

int main() {
    printf(1, "Test 2: Passing value as 8 in nice which is out of bounds.\n");
    int pid=2;
    int value=8;
    nice(pid,value);
    exit();
}
```

```
$ test2_nice
Test 2: Passing value as 8 in nice which is out of bounds.
Error: Priority not within range(1-5)
```

The program attempts to change the priority of a process with pid = 2 to 8, which is out of bounds, likely triggering an error due to invalid priority.

3. test3_nice.c

Makefile M   test3_nice.c U

test3_nice.c

```c
#include "types.h"
#include "user.h"
#include "param.h"

int main() {
    printf(1, "Test 3: Changing the value of a process that does not exist.\n");
    nice(70,3);
    exit();
}
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
init: starting sh
$ ps
name     pid     state    priority
init     1       SLEEPING        3
 sh      2       SLEEPING        3
 ps      3       RUNNING         3
 $ test3_nice
Test 3: Changing the value of a process that does not exist.
No process with Pid=70
```

The program tries to change the priority of a non-existent process with pid = 70 to 3, which will likely result in an error message indicating that the process was not found.

Task 2 Round Robin Priority Scheduler

MAKEFILE

Makefile M × | defs.h M

Makefile

```
clean:
	initcode initcode.out kernel xv6.img fs.img kernelmemfs mkfs \
	.gdbinit \
	$(UPROGS)

# make a printout
FILES = $(shell grep -v '^\#' runoff.list)
PRINT = runoff.list runoff.spec README toc.hdr toc.ftr $(FILES)

xv6.pdf: $(PRINT)
	./runoff
	ls -l xv6.pdf

print: xv6.pdf

# run in emulators

bochs : fs.img xv6.img
	if [ ! -e .bochsrc ]; then ln -s dot-bochsrc .bochsrc; fi
	bochs -q

# try to generate a unique GDB port
GDBPORT = $(shell expr `id -u` % 5000 + 25000)
# QEMU's gdb stub command line changed in 0.11
QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \
	then echo "-gdb tcp::$(GDBPORT)"; \
	else echo "-s -p $(GDBPORT)"; fi)
ifndef CPUS
CPUS := 1
endif
QEMUOPTS = -drive file=fs.img,index=1,media=disk,format=raw -drive
```

Make CPU count to 1

Makefile M × defs.h M

Makefile

```
# Try to infer the correct QEMU
ifndef QEMU
QEMU = $(shell if which qemu > /dev/null; \
    then echo qemu; exit; \
    elif which qemu-system-i386 > /dev/null; \
    then echo qemu-system-i386; exit; \
    else \
    qemu=/Applications/Q.app/Contents/MacOS/i386-
    if test -x $$qemu; then echo $$qemu; exit; fi
    echo "***" 1>&2; \
    echo "*** Error: Couldn't find a working QEMU
    echo "*** Is the directory containing the qem
    echo "*** or have you tried setting the QEMU
    echo "***" 1>&2; exit 1)
endif

CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
OBJCOPY = $(TOOLPREFIX)objcopy
OBJDUMP = $(TOOLPREFIX)objdump
CFLAGS = -fno-pic -static -fno-builtin -fno-stric
#CFLAGS = -fno-pic -static -fno-builtin -fno-stri
CFLAGS += $(shell $(CC) -fno-stack-protector -E -
ASFLAGS = -m32 -gdwarf-2 -Wa,-divide
# FreeBSD ld wants ``elf_i386_fbsd''
LDFLAGS += -m $(shell $(LD) -V | grep elf_i386 2>
ifndef PRIORITY
    PRIORITY := 1
endif
CFLAGS += -DCUSTOM_SCHEDULER=$(PRIORITY)
```

Add FLAG PRIORITY so that user has option to compile with normal RR and RR with priority.

proc.c

Makefile M | proc.c M

proc.c

```c
found:

  return p;
}

//PAGEBREAK: 32
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  #ifndef CUSTOM_SCHEDULER
    int res;
    res=add_process_to_queue(p,p->priority-1);
    if(res==-1){
      cprintf("Unable to create process with pid=%d\n",p->pid);
      release(&ptable.lock);
      exit();
    }
  #endif
  release(&ptable.lock);
}
```

The userinit() function initializes the first user process, setting up its memory, registers, and state as RUNNABLE, and, if using a custom scheduler, adds it to the appropriate priority queue. It handles synchronization with ptable.lock to ensure safe process state updates.

Makefile M   proc.c M

proc.c

```c
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  safestrcpy(np->name, proc->name, sizeof(proc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;
  #ifndef CUSTOM_SCHEDULER
    int res;
    res=add_process_to_queue(np,np->priority-1);
    if(res==-1){
      cprintf("Unable to create process with pid=%d\n",np->pid);
      release(&ptable.lock);
      exit();
    }
  #endif
  release(&ptable.lock);
  return pid;
}
```

This section checks if a custom scheduler is not being used. It then tries to add the new process np to the appropriate priority queue. If adding fails, it prints an error message and exits, ensuring proper handling of process creation failures.

Makefile M | proc.c M

proc.c

```c
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        #ifndef CUSTOM_SCHEDULER
          remove_process_from_queue(p, p->priority-1);
        #endif
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

In this part of the wait() function, if a custom scheduler is not in use, the code removes the process p from the priority queue when the process transitions to the UNUSED state after being freed. This ensures that terminated (ZOMBIE) processes are properly removed from scheduling queues before they are cleaned up.

```c
struct cpu*
mycpu(void)
{
  int apicid, i;

  if(readeflags()&FL_IF)
    panic("mycpu called with interrupts enabled\n");

  apicid = lapicid();
  // APIC IDs are not guaranteed to be contiguous. Maybe we should have
  // a reverse map, or reserve a register to store &cpus[i].
  for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
      return &cpus[i];
  }
  panic("unknown apicid\n");
}

struct proc*
myproc(void) {
  struct cpu *c;
  struct proc *p;
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
  return p;
}
```

mycpu(): This function returns a pointer to the cpu structure corresponding to the current CPU, identified by its APIC ID. It ensures interrupts are disabled before running and panics if the APIC ID is not recognized.

myproc(): This function returns a pointer to the currently running process on the current CPU. It temporarily disables interrupts to safely access the cpu and proc structures, then restores interrupt settings.

M Makefile M   C proc.c M ×

C proc.c

```c
//      via swtch back to the scheduler.
#ifndef CUSTOM_SCHEDULER
void
scheduler(void)
{
  struct proc *p=0;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    int i;
    for(i=0;i<PLEVELS;i++){
      if(marker[i]>0){
        p=priority_q[i][0].p;
        remove_process_from_queue(p,i);
        break;
      }
    }

    if(p!=0 && p->state==RUNNABLE){
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      swtch(&(c->scheduler), p->context);
      switchkvm();
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
#else
```

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
#endif
```

The code defines two different versions of the scheduler() function, based on whether CUSTOM_SCHEDULER is defined:

1. **Without CUSTOM_SCHEDULER**:
   - Implements a custom scheduling mechanism using priority queues.
   - It iterates over the priority levels (PLEVELS), selects a process from the highest non-empty queue, removes it from the queue, and runs it if it is in the RUNNABLE state.

2. **With CUSTOM_SCHEDULER**:
   - Implements a simpler round-robin scheduler.
   - It iterates over all processes in the process table, selects a RUNNABLE process, and runs it.

In both versions, the scheduler enables interrupts, switches to the chosen process, and manages the process state transitions while holding the ptable.lock for safe access.

```c
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  #ifndef CUSTOM_SCHEDULER
    int res=add_process_to_queue(myproc(),myproc()->priority-1);
    if(res==-1){
      cprintf("Unable to create process with pid=%d\n",myproc()->pid);
    }
  #endif
  sched();
  release(&ptable.lock);
}
```

The yield() function makes the current process give up the CPU voluntarily, setting its state to RUNNABLE. If CUSTOM_SCHEDULER is not defined, it adds the process back to the appropriate priority queue. The sched() function is then called to perform a context switch, and finally, the ptable.lock is released.

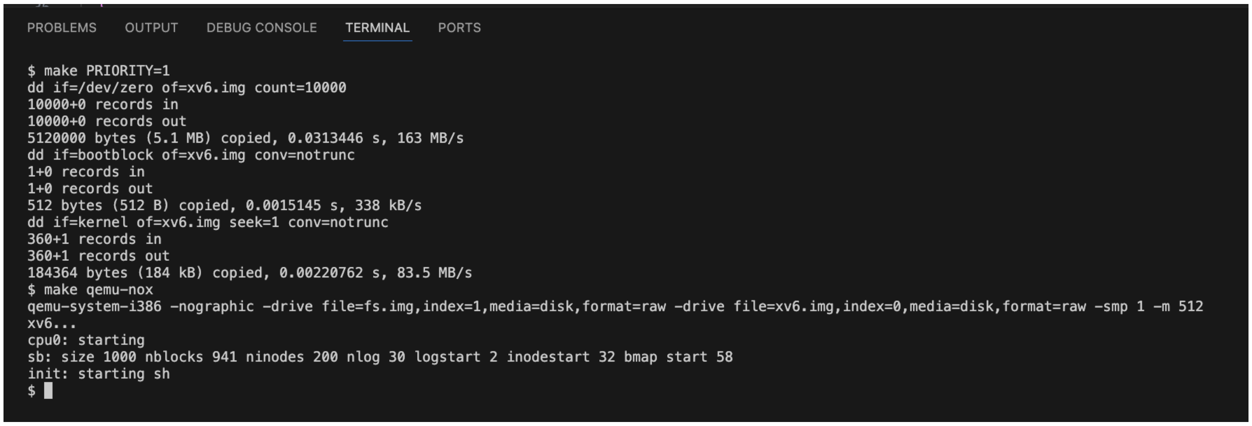
```
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      #ifndef CUSTOM_SCHEDULER
    int res=add_process_to_queue(p,p->priority-1);
    if(res==-1){
      cprintf("Unable to create process with pid=%d\n",p->pid);
    }
      #endif
    }
  }
}
```

The wakeup1() function iterates over all processes in the process table and wakes up any process that is in the SLEEPINGstate and waiting on the specified chan. It changes the process state to RUNNABLE and, if CUSTOM_SCHEDULER is not defined, adds the process to the appropriate priority queue. If adding to the queue fails, it prints an error message.

```
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
        #ifndef CUSTOM_SCHEDULER
          int res;
          res=add_process_to_queue(p,p->priority-1);
          if(res==-1){
            cprintf("Unable to create process with pid=%d\n",p->pid);
            release(&ptable.lock);
            exit();
          }
        #endif
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

The kill() function sets the killed flag for a process with the specified pid and wakes it up if it is SLEEPING by setting its state to RUNNABLE. If CUSTOM_SCHEDULER is not defined, it adds the process to the priority queue. If adding fails, it prints an error and exits. The function returns 0 on success or -1 if no process with the given pid is found.

HOW TO RUN SCHEDULER WITH PRIORITY RR

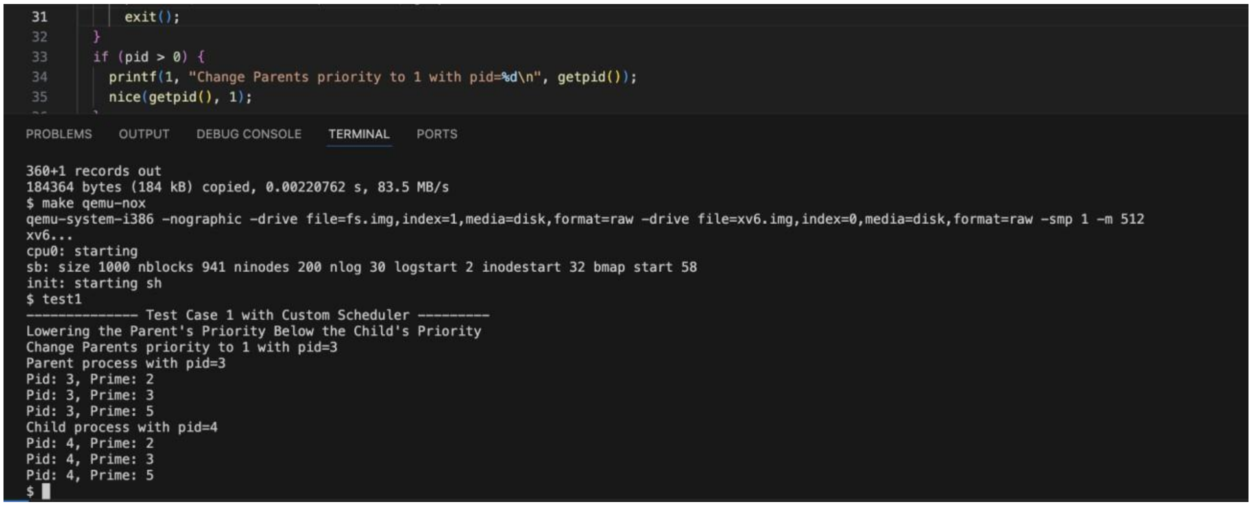



```
$ make PRIORITY=1
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0313446 s, 163 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.0015145 s, 338 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
360+1 records in
360+1 records out
184364 bytes (184 kB) copied, 0.00220762 s, 83.5 MB/s
$ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 512
xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ 
```

TEST CASES

1. test1.c

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

void prime() {
  int num = 2;  // Start from 2, since 1 is not a prime number
  while (num < 7) {
    int prime = 1;
    int i;
    for (i = 2; i * i <= num; i++) {  // Corrected condition to i * i <= num
      if (num % i == 0) {
        prime = 0;
        break;
      }
    }
    if (prime) {
      printf(1, "Pid: %d, Prime: %d\n", getpid(), num);
    }
    num++;
  }
}


int main() {
  printf(1,"-------------- Test Case 1 with Custom Scheduler ---------\n");
  printf(1,"Lowering the Parent's Priority Below the Child's Priority\n");
  int pid = fork();
  sleep(200); // to process changing the pid
  if (pid < 0) {
      printf(2, "Fork fail in pid: %d\n", getpid());
      exit();
  }
  if (pid > 0) {
    printf(1, "Change Parents priority to 1 with pid=%d\n", getpid());
    nice(getpid(), 1);
  }
  if (pid == 0) {
    printf(1, "Child process with pid=%d\n", getpid());
    // nice(getpid(),5);
    prime();
    exit();
  } else {
    printf(1, "Parent process with pid=%d\n", getpid());
    prime();
    wait();
    exit();
  }
  exit();
}

```



```c
    exit();
  }
  if (pid > 0) {
    printf(1, "Change Parents priority to 1 with pid=%d\n", getpid());
    nice(getpid(), 1);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
360+1 records out
184364 bytes (184 kB) copied, 0.00220762 s, 83.5 MB/s
$ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 512
xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test1
-------------- Test Case 1 with Custom Scheduler ---------
Lowering the Parent's Priority Below the Child's Priority
Change Parents priority to 1 with pid=3
Parent process with pid=3
Pid: 3, Prime: 2
Pid: 3, Prime: 3
Pid: 3, Prime: 5
Child process with pid=4
Pid: 4, Prime: 2
Pid: 4, Prime: 3
Pid: 4, Prime: 5
$
```

The code tests a custom scheduler by creating a parent and child process. By default, the child process has a higher priority, but the parent's priority is lowered and executes first.

2.

test2.c U ×

test2.c

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

// Function to calculate and print prime numbers
void find_primes(int priority, int t) {
    int pid = getpid();
    // Set the process priority using the nice system call
    nice(pid, priority);

    int start_time = uptime();
    int i = 1;

    printf(1, "Started Current PID: %d, Priority: %d\n", pid, priority);

    while (1) {
        int  j;

        // Check if the number is a prime
        for (j = 2; j * j <= i; j++) {
            if (i % j == 0) {
                break;
            }
        }

        int current_time = uptime();
        if (current_time - start_time >= t) {
            int end_time = uptime();
            int execution_time = end_time - start_time;
            printf(1, "Finished Current PID: %d, Priority: %d, Execution Time: %d\n", pid, priority, execution_time);
            exit();
        }
        i++;
    }
}

int main() {
    int max_time = 1000;

    // Creating 3 children with same priority
    int p1 = fork();
    if (p1 == 0) {
        find_primes(2, max_time);
        exit();
    }

    int p2 = fork();
    if (p2 == 0) {
        find_primes(2, max_time);
        exit();
    }

    int p3 = fork();
    if (p3 == 0) {
        find_primes(2, max_time);
        exit();
    }
    while(wait()!=-1);
    exit();
}
```

```c
            break;
        }
    }

    int current_time = uptime();
    if (current_time - start_time >= t) {
        int end_time = uptime();
        int execution_time = end_time - start_time;
        printf(1, "Finished Current PID: %d, Priority: %d, Execution Time: %d\n", pid, priority, execution_time);
        exit();
    }
    i++;
  }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

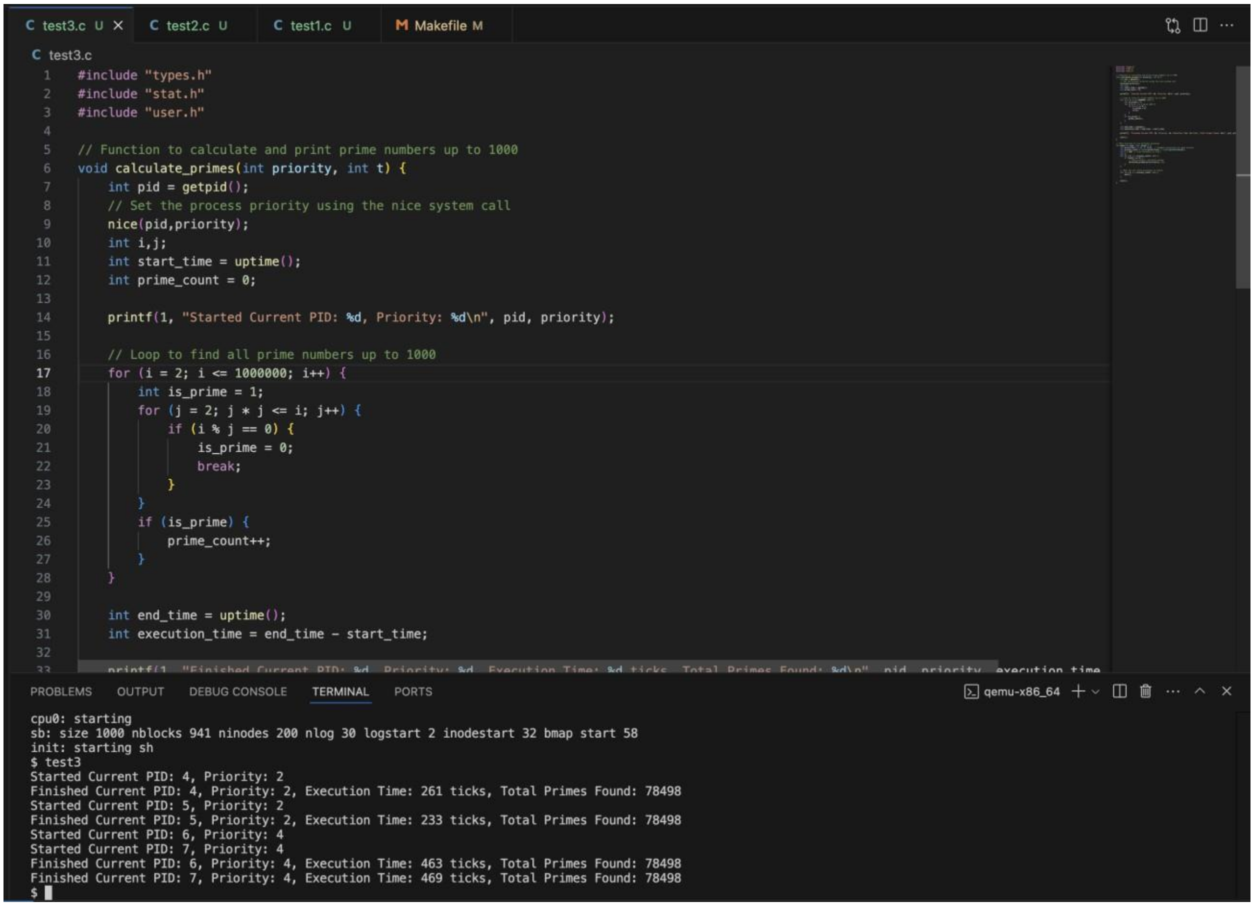
```
$ test2
Started Current PID: 25, Priority: 2
Started Current PID: 26, Priority: 2
Started Current PID: 27, Priority: 2
Finished Current PID: 25, Priority: 2, Execution Time: 1000
Finished Current PID: 26, Priority: 2, Execution Time: 1001
Finished Current PID: 27, Priority: 2, Execution Time: 1000
$
```

The code creates three child processes, each with the same priority level (2). Each child process calculates prime numbers for a specified duration (max_time = 1000 ticks) and then exits. Since all three processes have identical priority, they share the CPU time equally, resulting in similar execution times for each process. This demonstrates fair scheduling when processes have the same priority.

3.

test3.c

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

// Function to calculate and print prime numbers up to 1000
void calculate_primes(int priority, int t) {
    int pid = getpid();
    // Set the process priority using the nice system call
    nice(pid,priority);
    int i,j;
    int start_time = uptime();
    int prime_count = 0;

    printf(1, "Started Current PID: %d, Priority: %d\n", pid, priority);

    // Loop to find all prime numbers up to 1000
    for (i = 2; i <= 1000000; i++) {
        int is_prime = 1;
        for (j = 2; j * j <= i; j++) {
            if (i % j == 0) {
                is_prime = 0;
                break;
            }
        }
        if (is_prime) {
            prime_count++;
        }
    }

    int end_time = uptime();
    int execution_time = end_time - start_time;

    printf(1, "Finished Current PID: %d, Priority: %d, Execution Time: %d ticks, Total Primes Found: %d\n", pid, priority, execution_time, prime_count);

    exit();
}

// Main function to run multiple processes
int main(int argc, char *argv[]) {
    int priorities[] = {2, 2, 4,4};  // Example priorities for each process
    int process_count = sizeof(priorities) / sizeof(priorities[0]);
    int t = 100; // Time threshold in ticks
    int i;
    for (i = 0; i < process_count; i++) {
        if (fork() == 0) {
            // Child process: calculate primes
            calculate_primes(priorities[i], t);
        }
    }

    // Wait for all child processes to finish
    for (i = 0; i < process_count; i++) {
        wait();
    }
```



The code creates multiple child processes, each with different priorities: two processes with priority 2 (higher priority) and two with priority 4 (lower priority). Because processes with lower numerical priority values (like 2) are given higher scheduling priority, they are executed first. As a result, these higher-priority processes complete their prime number calculations faster, leading to shorter execution times compared to the lower-priority processes.