컴퓨터학부, 20202925, 강준현

1. 개요

xv6의 프로세스 관리 및 스케줄링 기법을 직접 코드를 분석해보며 이해해보고, 이를 바탕으로xv6에서 기존 스케줄러를 변경하고, 명령어를 추가하며 주어진 로직의 스케쥴러를 완성해본다. 기존 xv6의 스케줄링 기법은 다음 실행할 프로세스를 ptable을 순회하며 RUNNABLE 상태인 프로세스를 순차 선택한다. 이 과제에서는 우선순위가 가장 작은 프로세스를 선택하는 것이고 우선순위는 프로세스가 만들어지는 순서인 weight를 이용해 new priority = old priority + (time slice/ weight)로 한다.

2. 상세설계

xv6 scheduler 분석

xv6의 main.c 의 main()에서 지정된 초기 작업 후 mpmain()이 호출 되면 mpmain함수에서 scheduler()가 호출된다. proc.c에 가서 scheduler()를 살펴보면 for(;;)형태로 무한루프를 돌며 절대 리턴 되지 않는다. 계속 반복적으로 무한루프를 돌며 process table에 있는 프로세스들을 반복하며 RUNNABLE한 상태의 process를 순차적으로 찾고 이를 찾으면 switchuvm(p)를 통해 선택한 프로세스로 바꿔주고, p->state = RUNNING으로 바꿔준다. 그 후 swtch(&cpu->scheduler, p->context); 를 통해 현재 cpu에 있는 레지스터 를 저장하고, p->context의 레지스터 정보를 불러온다. 이 함수는 프로세스가 exit되거나 스케줄러함수를 부르는 조건을 만족했을 때 리턴된다. 그리고 다시 switchkvm()이 호출되고 c-> proc=0 으로 해주며 다시 다음 프로세스를 찾기 위해 무한 루프를 돈다.

ssu\_schedule()를 통해 원하는 로직으로 스케줄링을 하기 위해서는 기존의 scheduler()에서 호출을 하게 해 수행하는 방법을 이용한다. 기존에 순차적으로 proces table을 돌던 for 부분을 없애고 ssu\_schedule()를 호출해 이 안에서 process table을 for로 돌며 최소의 priority를 갖는 프로세스를 찾아 주는 방법으로 설계한다. 그리고 찾은 프로세스를 기존 스케줄함수에서 받아 실행해준다. 선정된 프로세스의 priority를 new\_priority=old\_priority+(TIME\_SLICE/weight)규칙에 맞게 업데이트 해주고, 이 값을 기반으로 현재 관리되고 있는 최소의 priority도 ptable에 정보를 갱신한다.

process가 wake up이 되면 관리하고 있는 최소의 priority를 부여하는 것이므로 wakeup1()에서 해당 프로세스에 ptable의 최소 priority로 값을 부여한다.

또한 userinit ()에서 최초의 priority를 3으로 지정한다. allocproc()에서 프로세스가 생성될때 마다 weight값을 +1씩 증가시키고, 최소의 priority로 할당해준다.

sdebug.c 에서는 5개 자식을 생성한 후 weightset()으로 weight를 지정한다.

3. 구현방법

우선 proc구조체에 weight과 priority를 uint 자료형으로 넣어, 모든 프로세스가 weight과 priority값을 갖게 해준다.

기존의 scheduler() 에   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) 이 반복문을 없애며 기존 규칙을 벗어나고, ssu\_schedule()에서에   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) if(ret==NULL || (ret->priority > p->priority)) 을 통해 priority가 최소인 프로세스를 찾아준다. 선정된 프로세스는 void update\_priority(struct proc \*proc)함수에서 new\_priority=old\_priority+(TIME\_SLICE/weight)규칙에 맞도록 업데이트 해주고,     update\_min\_priority(); 를 통해 최소의 priority를 ptable에서 관리하게 해준다.     update\_min\_priority() 에서는 ssu\_schedule()과 같은 방법으로 최소의 priority를 찾아 ptable에 저장한다. 'debug=1'옵션이 들어올 경우에는 ssu\_schedule()에서 선정된 프로세스의 proc구조체의 정보를 갖고 출력을 해준다.

process가 wake up이 되면 관리하고 있는 최소의 priority를 부여하는 것이므로 wakeup1()에서 해당 프로세스에assign\_min\_priority()를 통해 현재 ptable의 최소 priority로 값을 부여한다.

또한 userinit ()에서 최초의 priority를 3으로 지정한다. allocproc()에서 프로세스가 생성될때 마다 weight값을 +1씩 증가시키고, assign\_min\_priority() 통해 현재 ptable의 최소의 priority로 할당해준다.

sdebug.c 에서는 PNUM만큼 fork를 통해 자식을 생성하고 생성된 순서에 맞게 weightset()호출을 해 weight를 지정해준다. 무한 루프를 돌며 counter 을 clock tick마다 +1 씩하며 TOTAL\_COUNTER 가 되면 exit()을 통해 자식 프로세스를 종료시킨다. 그리고PRINT\_CYCLE 가 되면 정보를 출력한다.

    for(n=0; n<PNUM; n++){

        uint counter = 0;

        pid = fork();

        uint stime = uptime();//프로세스 생성 시간

        if(pid < 0)

            break;

        if(pid == 0){ //자식 프로세스

            weightset(n+1); //weight 1부터 +1씩 하면서 지정

            int ppid = getpid();//자식 프로세스의 pid를 얻는 시스템콜

            while(1){

                counter++; //clock tick마다 1씩 증가

                if(counter == TOTAL\_COUNTER){ // 이경우 출력하고, 프로세스 종료

                    printf(1,"PID: %d terminated\n", ppid);

                    exit();

                //  break;

                }

                if(counter == PRINT\_CYCLE){ //천만이 되면 정보 출력

                    uint etime = uptime();

                    uint t = (etime - stime)\*10; //tick 차이 ms로 만들기

                    printf(1,"PID: %d, WEIGHT: %d, TIMES : %d ms\n", ppid, n+1, t);

                }

            }

        }

    }

weightset()함수에서는 0으로 들어온 경우 return으로 예외를 처리해주고, 그 외의 경우에는 커널함수 do\_weightset()에 weight를 넘기며 do\_weightset()은 myproc()->weight에 지정한 weight를 넣는다.

acquire(&ptable.lock);

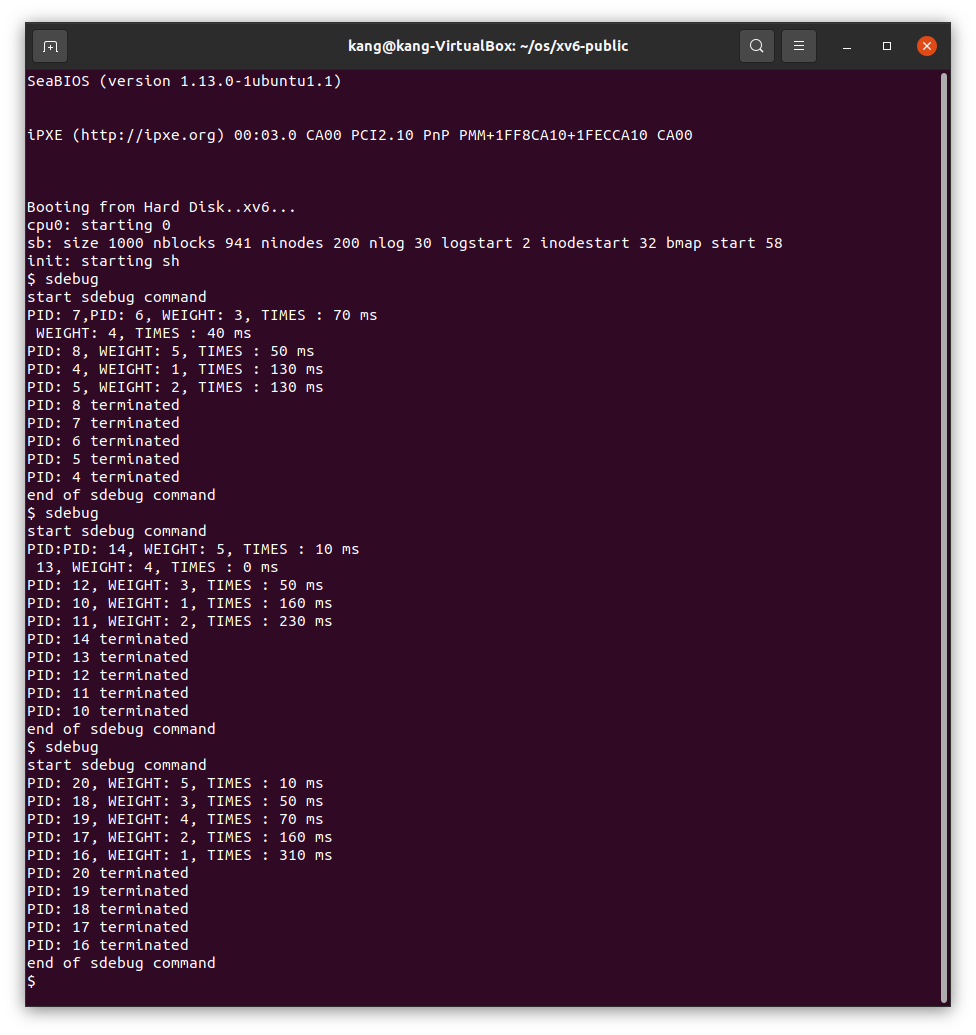
  myproc()->weight=weight;

  release(&ptable.lock);

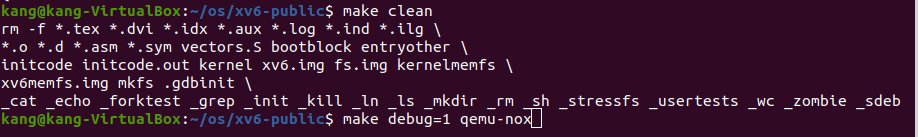
do\_weightset() 은 proc.c에 만들고, defs.h에 추가해준다.

4. 결과

sdebug



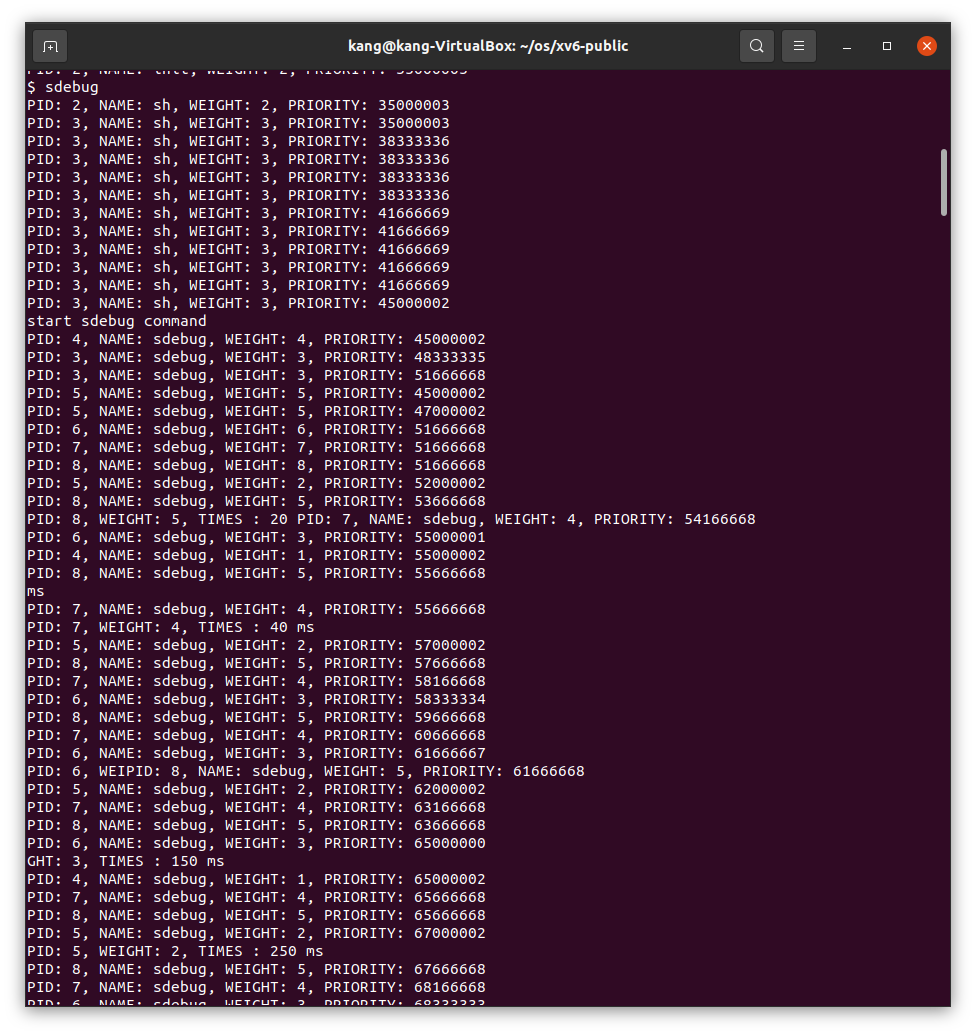
make debug=1 qemu-nox

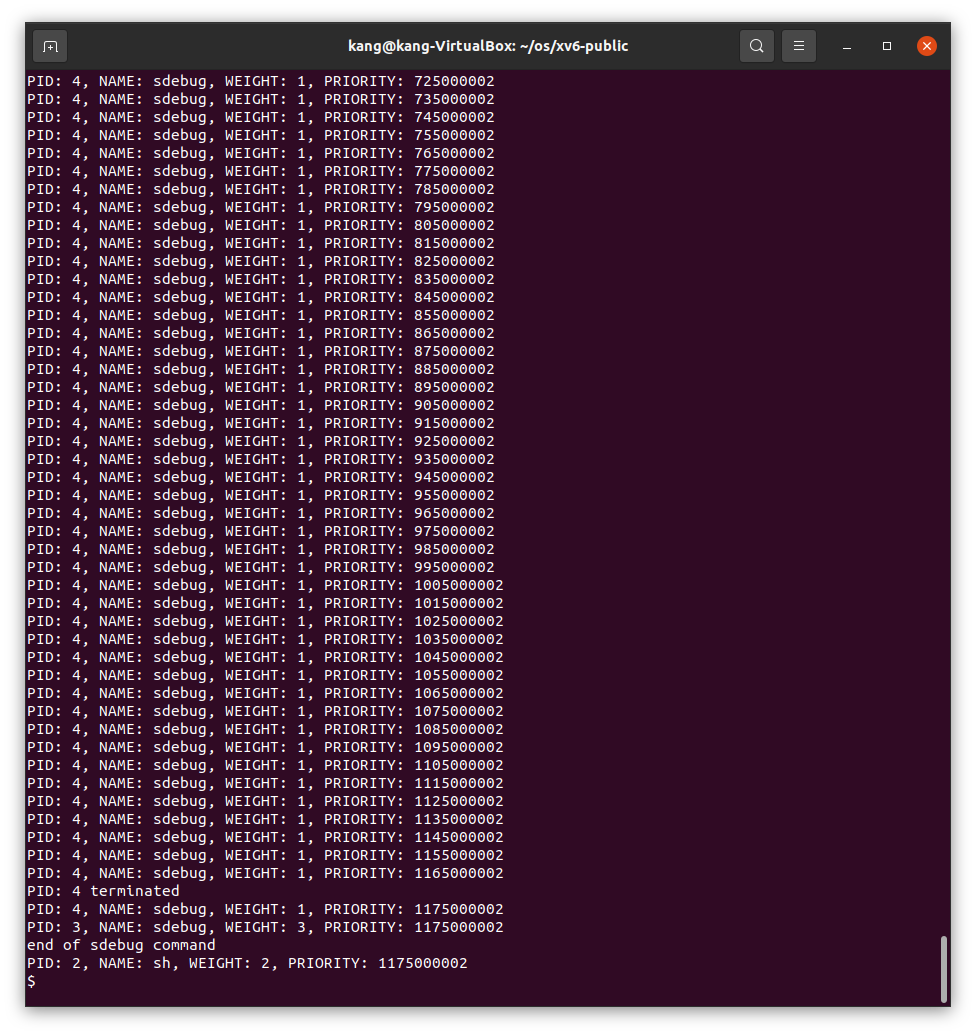


텍스트이(가) 표시된 사진

자동 생성된 설명

sdebug





5. 소스코드

**Makefile**

OBJS = \

    bio.o\

    console.o\

    exec.o\

    file.o\

    fs.o\

    ide.o\

    ioapic.o\

    kalloc.o\

    kbd.o\

    lapic.o\

    log.o\

    main.o\

    mp.o\

    picirq.o\

    pipe.o\

    proc.o\

    sleeplock.o\

    spinlock.o\

    string.o\

    swtch.o\

    syscall.o\

    sysfile.o\

    sysproc.o\

    trapasm.o\

    trap.o\

    uart.o\

    vectors.o\

    vm.o\

# Cross-compiling (e.g., on Mac OS X)

# TOOLPREFIX = i386-jos-elf

# Using native tools (e.g., on X86 Linux)

#TOOLPREFIX =

# Try to infer the correct TOOLPREFIX if not set

ifndef TOOLPREFIX

TOOLPREFIX := $(shell if i386-jos-elf-objdump -i 2>&1 | grep '^elf32-i386$$' >/dev/null 2>&1; \

    then echo 'i386-jos-elf-'; \

    elif objdump -i 2>&1 | grep 'elf32-i386' >/dev/null 2>&1; \

    then echo ''; \

    else echo "\*\*\*" 1>&2; \

    echo "\*\*\* Error: Couldn't find an i386-\*-elf version of GCC/binutils." 1>&2; \

    echo "\*\*\* Is the directory with i386-jos-elf-gcc in your PATH?" 1>&2; \

    echo "\*\*\* If your i386-\*-elf toolchain is installed with a command" 1>&2; \

    echo "\*\*\* prefix other than 'i386-jos-elf-', set your TOOLPREFIX" 1>&2; \

    echo "\*\*\* environment variable to that prefix and run 'make' again." 1>&2; \

    echo "\*\*\* To turn off this error, run 'gmake TOOLPREFIX= ...'." 1>&2; \

    echo "\*\*\*" 1>&2; exit 1; fi)

endif

# If the makefile can't find QEMU, specify its path here

# QEMU = qemu-system-i386

# 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; \

    elif which qemu-system-x86\_64 > /dev/null; \

    then echo qemu-system-x86\_64; exit; \

    else \

    qemu=/Applications/Q.app/Contents/MacOS/i386-softmmu.app/Contents/MacOS/i386-softmmu; \

    if test -x $$qemu; then echo $$qemu; exit; fi; fi; \

    echo "\*\*\*" 1>&2; \

    echo "\*\*\* Error: Couldn't find a working QEMU executable." 1>&2; \

    echo "\*\*\* Is the directory containing the qemu binary in your PATH" 1>&2; \

    echo "\*\*\* or have you tried setting the QEMU variable in Makefile?" 1>&2; \

    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-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer

CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)

ASFLAGS = -m32 -gdwarf-2 -Wa,-divide

# FreeBSD ld wants ``elf\_i386\_fbsd''

LDFLAGS += -m $(shell $(LD) -V | grep elf\_i386 2>/dev/null | head -n 1)

# Disable PIE when possible (for Ubuntu 16.10 toolchain)

ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]no-pie'),)

CFLAGS += -fno-pie -no-pie

endif

ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]nopie'),)

CFLAGS += -fno-pie -nopie

endif

ifeq ($(debug), 1)

CFLAGS += -DDEBUG

endif

#make debug=1 qemu 라고 옵션이 들어오면 #define DEBUG 로 선언되는 것처럼 사용가능

xv6.img: bootblock kernel

    dd if=/dev/zero of=xv6.img count=10000

    dd if=bootblock of=xv6.img conv=notrunc

    dd if=kernel of=xv6.img seek=1 conv=notrunc

xv6memfs.img: bootblock kernelmemfs

    dd if=/dev/zero of=xv6memfs.img count=10000

    dd if=bootblock of=xv6memfs.img conv=notrunc

    dd if=kernelmemfs of=xv6memfs.img seek=1 conv=notrunc

bootblock: bootasm.S bootmain.c

    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c

    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o

    $(OBJDUMP) -S bootblock.o > bootblock.asm

    $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock

    ./sign.pl bootblock

entryother: entryother.S

    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o

    $(OBJCOPY) -S -O binary -j .text bootblockother.o entryother

    $(OBJDUMP) -S bootblockother.o > entryother.asm

initcode: initcode.S

    $(CC) $(CFLAGS) -nostdinc -I. -c initcode.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o

    $(OBJCOPY) -S -O binary initcode.out initcode

    $(OBJDUMP) -S initcode.o > initcode.asm

kernel: $(OBJS) entry.o entryother initcode kernel.ld

    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother

    $(OBJDUMP) -S kernel > kernel.asm

    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > kernel.sym

# kernelmemfs is a copy of kernel that maintains the

# disk image in memory instead of writing to a disk.

# This is not so useful for testing persistent storage or

# exploring disk buffering implementations, but it is

# great for testing the kernel on real hardware without

# needing a scratch disk.

MEMFSOBJS = $(filter-out ide.o,$(OBJS)) memide.o

kernelmemfs: $(MEMFSOBJS) entry.o entryother initcode kernel.ld fs.img

    $(LD) $(LDFLAGS) -T kernel.ld -o kernelmemfs entry.o  $(MEMFSOBJS) -b binary initcode entryother fs.img

    $(OBJDUMP) -S kernelmemfs > kernelmemfs.asm

    $(OBJDUMP) -t kernelmemfs | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > kernelmemfs.sym

tags: $(OBJS) entryother.S \_init

    etags \*.S \*.c

vectors.S: vectors.pl

    ./vectors.pl > vectors.S

ULIB = ulib.o usys.o printf.o umalloc.o

\_%: %.o $(ULIB)

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^

    $(OBJDUMP) -S $@ > $\*.asm

    $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $\*.sym

\_forktest: forktest.o $(ULIB)

    # forktest has less library code linked in - needs to be small

    # in order to be able to max out the proc table.

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o \_forktest forktest.o ulib.o usys.o

    $(OBJDUMP) -S \_forktest > forktest.asm

mkfs: mkfs.c fs.h

    gcc -Werror -Wall -o mkfs mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so

# that disk image changes after first build are persistent until clean.  More

# details:

# http://www.gnu.org/software/make/manual/html\_node/Chained-Rules.html

.PRECIOUS: %.o

UPROGS=\

    \_cat\

    \_echo\

    \_forktest\

    \_grep\

    \_init\

    \_kill\

    \_ln\

    \_ls\

    \_mkdir\

    \_rm\

    \_sh\

    \_stressfs\

    \_usertests\

    \_wc\

    \_zombie\

    \_sdebug\

fs.img: mkfs README $(UPROGS)

    ./mkfs fs.img README $(UPROGS)

-include \*.d

clean:

    rm -f \*.tex \*.dvi \*.idx \*.aux \*.log \*.ind \*.ilg \

    \*.o \*.d \*.asm \*.sym vectors.S bootblock entryother \

    initcode initcode.out kernel xv6.img fs.img kernelmemfs \

    xv6memfs.img 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 file=xv6.img,index=0,media=disk,format=raw -smp $(CPUS) -m 512 $(QEMUEXTRA)

qemu: fs.img xv6.img

    $(QEMU) -serial mon:stdio $(QEMUOPTS)

qemu-memfs: xv6memfs.img

    $(QEMU) -drive file=xv6memfs.img,index=0,media=disk,format=raw -smp $(CPUS) -m 256

qemu-nox: fs.img xv6.img

    $(QEMU) -nographic $(QEMUOPTS)

.gdbinit: .gdbinit.tmpl

    sed "s/localhost:1234/localhost:$(GDBPORT)/" < $^ > $@

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

    @echo "\*\*\* Now run 'gdb'." 1>&2

    $(QEMU) -serial mon:stdio $(QEMUOPTS) -S $(QEMUGDB)

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

    @echo "\*\*\* Now run 'gdb'." 1>&2

    $(QEMU) -nographic $(QEMUOPTS) -S $(QEMUGDB)

# 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.c ulib.c user.h cat.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 sdebug.c\

    README dot-bochsrc \*.pl toc.\* runoff runoff1 runoff.list\

    .gdbinit.tmpl gdbutil\

dist:

    rm -rf dist

    mkdir dist

    for i in $(FILES); \

    do \

        grep -v PAGEBREAK $$i >dist/$$i; \

    done

    sed '/CUT HERE/,$$d' Makefile >dist/Makefile

    echo >dist/runoff.spec

    cp $(EXTRA) dist

dist-test:

    rm -rf dist

    make dist

    rm -rf dist-test

    mkdir dist-test

    cp dist/\* dist-test

    cd dist-test; $(MAKE) print

    cd dist-test; $(MAKE) bochs || true

    cd dist-test; $(MAKE) qemu

# update this rule (change rev#) when it is time to

# make a new revision.

tar:

    rm -rf /tmp/xv6

    mkdir -p /tmp/xv6

    cp dist/\* dist/.gdbinit.tmpl /tmp/xv6

    (cd /tmp; tar cf - xv6) | gzip >xv6-rev10.tar.gz  # the next one will be 10 (9/17)

.PHONY: dist-test dist

**proc.h**

// 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?

  struct proc \*proc;           // The process running on this cpu or null

};

extern struct cpu cpus[NCPU];

extern int ncpu;

//PAGEBREAK: 17

// Saved registers for kernel context switches.

// Don't need to save all the segment registers (%cs, etc),

// because they are constant across kernel contexts.

// Don't need to save %eax, %ecx, %edx, because the

// x86 convention is that the caller has saved them.

// Contexts are stored at the bottom of the stack they

// describe; the stack pointer is the address of the context.

// The layout of the context matches the layout of the stack in swtch.S

// at the "Switch stacks" comment. Switch doesn't save eip explicitly,

// but it is on the stack and allocproc() manipulates it.

struct context {

  uint edi;

  uint esi;

  uint ebx;

  uint ebp;

  uint eip;

};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state

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)

  uint priority;           // Process priority

  uint weight;                  // Process weight

};

// Process memory is laid out contiguously, low addresses first:

//   text

//   original data and bss

//   fixed-size stack

//   expandable heap

**proc.c**

#include "types.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "x86.h"

#include "proc.h"

#include "spinlock.h"

#define TIME\_SLICE 10000000

#define NULL ((void \*)0)

uint weight =1;

struct {

  struct spinlock lock;

  struct proc proc[NPROC];

  uint min\_priority;

} ptable;

static struct proc \*initproc;

int nextpid = 1;

extern void forkret(void);

extern void trapret(void);

static void wakeup1(void \*chan);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*20202925\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

struct proc \*ssu\_schedule()

{

  struct proc \*p;

  struct proc \*ret =NULL;

    // Loop over process table looking for process to run.

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    //RUNNABLE 인 것들만 고른다.

    if(p->state != RUNNABLE)

      continue;

    //min\_priority 갖는 것 찾고 업데이트 해야함.

    if(ret==NULL || (ret->priority > p->priority))

      ret=p; // 첫 경우는 ret=NULL이므로 들어오고, 그 다음 부터는 priority가 더 작은 것만 들어오며 최소 priority찾는다.

  }

  //debug=1 옵션이 들어왔을 경우 출력되는 문장

#ifdef DEBUG

  if(ret!=NULL)

    cprintf("PID: %d, NAME: %s, WEIGHT: %d, PRIORITY: %d \n", ret->pid, ret->name, ret->weight, ret-> priority);

#endif

  return ret;

}

void update\_priority(struct proc \*proc)

{

  proc->priority = proc->priority + (TIME\_SLICE / proc->weight) ;//priority 업데이트 규칙에 맞게 proc의 priority 값을 업데이트한다.

}

void update\_min\_priority(void)

{

  struct proc \*min = NULL;

  struct proc \*p;

  if(min != NULL)

    ptable.min\_priority = min->priority;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->state != RUNNABLE)

      continue;

    if(min==NULL || (min->priority > p->priority)){

      min=p; // 첫 경우는 min=NULL이므로 들어오고, 그 다음 부터는 priority가 더 작은 것만 들어오며 최소 priority찾는다.

    }

  }

  if(min != NULL) //ptable 의 min\_priority 업데이트

    ptable.min\_priority = min->priority;

  return;

}

void assign\_min\_priority(struct proc \*proc)

{

  //현재 관리되고 있는 priority중 가장 작은 값을 부여

  proc->priority = ptable.min\_priority;

}

void

pinit(void)

{

  initlock(&ptable.lock, "ptable");

}

// Must be called with interrupts disabled

int

cpuid() {

  return mycpu()-cpus;

}

// Must be called with interrupts disabled to avoid the caller being

// rescheduled between reading lapicid and running through the loop.

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");

}

// Disable interrupts so that we are not rescheduled

// while reading proc from the cpu structure

struct proc\*

myproc(void) {

  struct cpu \*c;

  struct proc \*p;

  pushcli();

  c = mycpu();

  p = c->proc;

  popcli();

  return p;

}

//PAGEBREAK: 32

// Look in the process table for an UNUSED proc.

// If found, change state to EMBRYO and initialize

// state required to run in the kernel.

// Otherwise return 0.

static struct proc\*

allocproc(void)

{

  struct proc \*p;

  char \*sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

    if(p->state == UNUSED)

      goto found;

  release(&ptable.lock);

  return 0;

found:

  //weight생성될때마다 1씩 추가

  p->weight=weight++;

  p->state = EMBRYO;

  p->pid = nextpid++;

  //과제 명세에따라 독점 실행을 막기 위한 프로세스 생성 시 관리하고 있는 process의 priority중 가장 작은 값 부여

  assign\_min\_priority(p);

  release(&ptable.lock);

  // Allocate kernel stack.

  if((p->kstack = kalloc()) == 0){

    p->state = UNUSED;

    return 0;

  }

  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.

  sp -= sizeof \*p->tf;

  p->tf = (struct trapframe\*)sp;

  // Set up new context to start executing at forkret,

  // which returns to trapret.

  sp -= 4;

  \*(uint\*)sp = (uint)trapret;

  sp -= sizeof \*p->context;

  p->context = (struct context\*)sp;

  memset(p->context, 0, sizeof \*p->context);

  p->context->eip = (uint)forkret;

  return p;

}

//PAGEBREAK: 32

// Set up first user process.

void

userinit(void)

{

  struct proc \*p;

  extern char \_binary\_initcode\_start[], \_binary\_initcode\_size[];

  ptable.min\_priority = 3;//시스템 시작 시 3개의 유저 프로세스가 돌아가 3으로 지정하고 시작

  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;

  release(&ptable.lock);

}

// Grow current process's memory by n bytes.

// Return 0 on success, -1 on failure.

int

growproc(int n)

{

  uint sz;

  struct proc \*curproc = myproc();

  sz = curproc->sz;

  if(n > 0){

    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)

      return -1;

  } else if(n < 0){

    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)

      return -1;

  }

  curproc->sz = sz;

  switchuvm(curproc);

  return 0;

}

// Create a new process copying p as the parent.

// 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;

  struct proc \*curproc = myproc();

  // Allocate process.

  if((np = allocproc()) == 0){

    return -1;

  }

  // Copy process state from proc.

  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){

    kfree(np->kstack);

    np->kstack = 0;

    np->state = UNUSED;

    return -1;

  }

  np->sz = curproc->sz;

  np->parent = curproc;

  \*np->tf = \*curproc->tf;

  // Clear %eax so that fork returns 0 in the child.

  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)

    if(curproc->ofile[i])

      np->ofile[i] = filedup(curproc->ofile[i]);

  np->cwd = idup(curproc->cwd);

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

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;

}

// Exit the current process.  Does not return.

// An exited process remains in the zombie state

// until its parent calls wait() to find out it exited.

void

exit(void)

{

  struct proc \*curproc = myproc();

  struct proc \*p;

  int fd;

  if(curproc == initproc)

    panic("init exiting");

  // Close all open files.

  for(fd = 0; fd < NOFILE; fd++){

    if(curproc->ofile[fd]){

      fileclose(curproc->ofile[fd]);

      curproc->ofile[fd] = 0;

    }

  }

  begin\_op();

  iput(curproc->cwd);

  end\_op();

  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().

  wakeup1(curproc->parent);

  // Pass abandoned children to init.

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->parent == curproc){

      p->parent = initproc;

      if(p->state == ZOMBIE)

        wakeup1(initproc);

    }

  }

  // Jump into the scheduler, never to return.

  curproc->state = ZOMBIE;

  sched();

  panic("zombie exit");

}

// 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;

  struct proc \*curproc = myproc();

  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 != curproc)

        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;

        release(&ptable.lock);

        return pid;

      }

    }

    // No point waiting if we don't have any children.

    if(!havekids || curproc->killed){

      release(&ptable.lock);

      return -1;

    }

    // Wait for children to exit.  (See wakeup1 call in proc\_exit.)

    sleep(curproc, &ptable.lock);  //DOC: wait-sleep

  }

}

//PAGEBREAK: 42

// Per-CPU process scheduler.

// Each CPU calls scheduler() after setting itself up.

// Scheduler never returns.  It loops, doing:

//  - choose a process to run

//  - swtch to start running that process

//  - eventually that process transfers control

//      via swtch back to the scheduler.

void

scheduler(void)

{

  struct proc \*p;

  struct cpu \*c = mycpu();

  c->proc = 0;

  for(;;){

    // Enable interrupts on this processor.

    sti();

    acquire(&ptable.lock);

    p = ssu\_schedule(); //로직에 맞게 만들 스케듈러 호출 //// ssu에서 p->state !=  runnable

    if(p ==  NULL){

    release(&ptable.lock);

      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();

    //scheduler가 호출 될 때 마다 priority값을 업데이트 시켜줌

    update\_priority(p);

    //업데이트 한 값을 기반으로 스케줄 함수 호출이 있을 때 마다 min priority업데이트 시켜줌

    update\_min\_priority();

      // Process is done running for now.

      // It should have changed its p->state before coming back.

    c->proc = 0;

    release(&ptable.lock);

  }

}

// Enter scheduler.  Must hold only ptable.lock

// and have changed proc->state. Saves and restores

// intena because intena is a property of this

// kernel thread, not this CPU. It should

// be proc->intena and proc->ncli, but that would

// break in the few places where a lock is held but

// there's no process.

void

sched(void)

{

  int intena;

  struct proc \*p = myproc();

  if(!holding(&ptable.lock))

    panic("sched ptable.lock");

  if(mycpu()->ncli != 1)

    panic("sched locks");

  if(p->state == RUNNING)

    panic("sched running");

  if(readeflags()&FL\_IF)

    panic("sched interruptible");

  intena = mycpu()->intena;

  swtch(&p->context, mycpu()->scheduler);

  mycpu()->intena = intena;

}

// Give up the CPU for one scheduling round.

void

yield(void)

{

  acquire(&ptable.lock);  //DOC: yieldlock

  myproc()->state = RUNNABLE;

  sched();

  release(&ptable.lock);

}

// A fork child's very first scheduling by scheduler()

// will swtch here.  "Return" to user space.

void

forkret(void)

{

  static int first = 1;

  // Still holding ptable.lock from scheduler.

  release(&ptable.lock);

  if (first) {

    // Some initialization functions must be run in the context

    // of a regular process (e.g., they call sleep), and thus cannot

    // be run from main().

    first = 0;

    iinit(ROOTDEV);

    initlog(ROOTDEV);

  }

  // Return to "caller", actually trapret (see allocproc).

}

// Atomically release lock and sleep on chan.

// Reacquires lock when awakened.

void

sleep(void \*chan, struct spinlock \*lk)

{

  struct proc \*p = myproc();

  if(p == 0)

    panic("sleep");

  if(lk == 0)

    panic("sleep without lk");

  // Must acquire ptable.lock in order to

  // change p->state and then call sched.

  // Once we hold ptable.lock, we can be

  // guaranteed that we won't miss any wakeup

  // (wakeup runs with ptable.lock locked),

  // so it's okay to release lk.

  if(lk != &ptable.lock){  //DOC: sleeplock0

    acquire(&ptable.lock);  //DOC: sleeplock1

    release(lk);

  }

  // Go to sleep.

  p->chan = chan;

  p->state = SLEEPING;

  sched();

  // Tidy up.

  p->chan = 0;

  // Reacquire original lock.

  if(lk != &ptable.lock){  //DOC: sleeplock2

    release(&ptable.lock);

    acquire(lk);

  }

}

//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;

    assign\_min\_priority(p);//wake up 될 때 현재 관리되고 있는 최소의 priority로 할당

  }

}

// Wake up all processes sleeping on chan.

void

wakeup(void \*chan)

{

  acquire(&ptable.lock);

  wakeup1(chan);

  release(&ptable.lock);

}

// Kill the process with the given pid.

// Process won't exit until it returns

// to user space (see trap in trap.c).

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;

      release(&ptable.lock);

      return 0;

    }

  }

  release(&ptable.lock);

  return -1;

}

//PAGEBREAK: 36

// Print a process listing to console.  For debugging.

// Runs when user types ^P on console.

// No lock to avoid wedging a stuck machine further.

void

procdump(void)

{

  static char \*states[] = {

  [UNUSED]    "unused",

  [EMBRYO]    "embryo",

  [SLEEPING]  "sleep ",

  [RUNNABLE]  "runble",

  [RUNNING]   "run   ",

  [ZOMBIE]    "zombie"

  };

  int i;

  struct proc \*p;

  char \*state;

  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->state == UNUSED)

      continue;

    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])

      state = states[p->state];

    else

      state = "???";

    cprintf("%d %s %s", p->pid, state, p->name);

    if(p->state == SLEEPING){

      getcallerpcs((uint\*)p->context->ebp+2, pc);

      for(i=0; i<10 && pc[i] != 0; i++)

        cprintf(" %p", pc[i]);

    }

    cprintf("\n");

  }

}

//weigthset system call 에서 콜할 함수

void do\_weightset(uint weight)

{

  acquire(&ptable.lock);

  myproc()->weight=weight;

  release(&ptable.lock);

  return;

}

**sdebug.c**

#include "types.h"

#include "stat.h"

#include "user.h"

#define PNUM 5

#define PRINT\_CYCLE 10000000

#define TOTAL\_COUNTER 500000000

void sdebug\_func(void)

{

    int n, pid;

    printf(1,"start sdebug command\n");

    for(n=0; n<PNUM; n++){

        uint counter = 0;

        pid = fork();

        uint stime = uptime();//프로세스 생성 시간

        if(pid < 0)

            break;

        if(pid == 0){ //자식 프로세스

            weightset(n+1); //weight 1부터 +1씩 하면서 지정

            int ppid = getpid();//자식 프로세스의 pid를 얻는 시스템콜

            while(1){

                counter++; //clock tick마다 1씩 증가

                if(counter == TOTAL\_COUNTER){ // 이경우 출력하고, 프로세스 종료

                    printf(1,"PID: %d terminated\n", ppid);

                    exit();

                //  break;

                }

                if(counter == PRINT\_CYCLE){ //천만이 되면 정보 출력

                    uint etime = uptime();

                    uint t = (etime - stime)\*10; //tick 차이 ms로 만들기

                    printf(1,"PID: %d, WEIGHT: %d, TIMES : %d ms\n", ppid, n+1, t);

                }

            }

        }

    }

//생성된 자식프로세스들이 모두 끝날 때까지 기다림

    for(; n > 0; n--){

        if(wait() < 0){

        printf(1, "wait stopped early\n");

        exit();

        }

    }

    if(wait() != -1){

        printf(1, "wait got too many\n");

        exit();

    }

    printf(1,"end of sdebug command\n");

    return;

}

int main(void)

{

    sdebug\_func();

    exit();

}

**defs.h**

struct buf;

struct context;

struct file;

struct inode;

struct pipe;

struct proc;

struct rtcdate;

struct spinlock;

struct sleeplock;

struct stat;

struct superblock;

// bio.c

void            binit(void);

struct buf\*     bread(uint, uint);

void            brelse(struct buf\*);

void            bwrite(struct buf\*);

// console.c

void            consoleinit(void);

void            cprintf(char\*, ...);

void            consoleintr(int(\*)(void));

void            panic(char\*) \_\_attribute\_\_((noreturn));

// exec.c

int             exec(char\*, char\*\*);

// file.c

struct file\*    filealloc(void);

void            fileclose(struct file\*);

struct file\*    filedup(struct file\*);

void            fileinit(void);

int             fileread(struct file\*, char\*, int n);

int             filestat(struct file\*, struct stat\*);

int             filewrite(struct file\*, char\*, int n);

// fs.c

void            readsb(int dev, struct superblock \*sb);

int             dirlink(struct inode\*, char\*, uint);

struct inode\*   dirlookup(struct inode\*, char\*, uint\*);

struct inode\*   ialloc(uint, short);

struct inode\*   idup(struct inode\*);

void            iinit(int dev);

void            ilock(struct inode\*);

void            iput(struct inode\*);

void            iunlock(struct inode\*);

void            iunlockput(struct inode\*);

void            iupdate(struct inode\*);

int             namecmp(const char\*, const char\*);

struct inode\*   namei(char\*);

struct inode\*   nameiparent(char\*, char\*);

int             readi(struct inode\*, char\*, uint, uint);

void            stati(struct inode\*, struct stat\*);

int             writei(struct inode\*, char\*, uint, uint);

// ide.c

void            ideinit(void);

void            ideintr(void);

void            iderw(struct buf\*);

// ioapic.c

void            ioapicenable(int irq, int cpu);

extern uchar    ioapicid;

void            ioapicinit(void);

// kalloc.c

char\*           kalloc(void);

void            kfree(char\*);

void            kinit1(void\*, void\*);

void            kinit2(void\*, void\*);

// kbd.c

void            kbdintr(void);

// lapic.c

void            cmostime(struct rtcdate \*r);

int             lapicid(void);

extern volatile uint\*    lapic;

void            lapiceoi(void);

void            lapicinit(void);

void            lapicstartap(uchar, uint);

void            microdelay(int);

// log.c

void            initlog(int dev);

void            log\_write(struct buf\*);

void            begin\_op();

void            end\_op();

// mp.c

extern int      ismp;

void            mpinit(void);

// picirq.c

void            picenable(int);

void            picinit(void);

// pipe.c

int             pipealloc(struct file\*\*, struct file\*\*);

void            pipeclose(struct pipe\*, int);

int             piperead(struct pipe\*, char\*, int);

int             pipewrite(struct pipe\*, char\*, int);

//PAGEBREAK: 16

// proc.c

int             cpuid(void);

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            setproc(struct proc\*);

void            sleep(void\*, struct spinlock\*);

void            userinit(void);

int             wait(void);

void            wakeup(void\*);

void            yield(void);

void        do\_weightset(uint);

// swtch.S

void            swtch(struct context\*\*, struct context\*);

// spinlock.c

void            acquire(struct spinlock\*);

void            getcallerpcs(void\*, uint\*);

int             holding(struct spinlock\*);

void            initlock(struct spinlock\*, char\*);

void            release(struct spinlock\*);

void            pushcli(void);

void            popcli(void);

// sleeplock.c

void            acquiresleep(struct sleeplock\*);

void            releasesleep(struct sleeplock\*);

int             holdingsleep(struct sleeplock\*);

void            initsleeplock(struct sleeplock\*, char\*);

// string.c

int             memcmp(const void\*, const void\*, uint);

void\*           memmove(void\*, const void\*, uint);

void\*           memset(void\*, int, uint);

char\*           safestrcpy(char\*, const char\*, int);

int             strlen(const char\*);

int             strncmp(const char\*, const char\*, uint);

char\*           strncpy(char\*, const char\*, int);

// syscall.c

int             argint(int, int\*);

int             argptr(int, char\*\*, int);

int             argstr(int, char\*\*);

int             fetchint(uint, int\*);

int             fetchstr(uint, char\*\*);

void            syscall(void);

// timer.c

void            timerinit(void);

// trap.c

void            idtinit(void);

extern uint     ticks;

void            tvinit(void);

extern struct spinlock tickslock;

// uart.c

void            uartinit(void);

void            uartintr(void);

void            uartputc(int);

// vm.c

void            seginit(void);

void            kvmalloc(void);

pde\_t\*          setupkvm(void);

char\*           uva2ka(pde\_t\*, char\*);

int             allocuvm(pde\_t\*, uint, uint);

int             deallocuvm(pde\_t\*, uint, uint);

void            freevm(pde\_t\*);

void            inituvm(pde\_t\*, char\*, uint);

int             loaduvm(pde\_t\*, char\*, struct inode\*, uint, uint);

pde\_t\*          copyuvm(pde\_t\*, uint);

void            switchuvm(struct proc\*);

void            switchkvm(void);

int             copyout(pde\_t\*, uint, void\*, uint);

void            clearpteu(pde\_t \*pgdir, char \*uva);

// number of elements in fixed-size array

#define NELEM(x) (sizeof(x)/sizeof((x)[0]))

**syscall.c**

#include "types.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

#include "x86.h"

#include "syscall.h"

// User code makes a system call with INT T\_SYSCALL.

// System call number in %eax.

// Arguments on the stack, from the user call to the C

// library system call function. The saved user %esp points

// to a saved program counter, and then the first argument.

// Fetch the int at addr from the current process.

int

fetchint(uint addr, int \*ip)

{

  struct proc \*curproc = myproc();

  if(addr >= curproc->sz || addr+4 > curproc->sz)

    return -1;

  \*ip = \*(int\*)(addr);

  return 0;

}

// Fetch the nul-terminated string at addr from the current process.

// Doesn't actually copy the string - just sets \*pp to point at it.

// Returns length of string, not including nul.

int

fetchstr(uint addr, char \*\*pp)

{

  char \*s, \*ep;

  struct proc \*curproc = myproc();

  if(addr >= curproc->sz)

    return -1;

  \*pp = (char\*)addr;

  ep = (char\*)curproc->sz;

  for(s = \*pp; s < ep; s++){

    if(\*s == 0)

      return s - \*pp;

  }

  return -1;

}

// Fetch the nth 32-bit system call argument.

int

argint(int n, int \*ip)

{

  return fetchint((myproc()->tf->esp) + 4 + 4\*n, ip);

}

// Fetch the nth word-sized system call argument as a pointer

// to a block of memory of size bytes.  Check that the pointer

// lies within the process address space.

int

argptr(int n, char \*\*pp, int size)

{

  int i;

  struct proc \*curproc = myproc();

  if(argint(n, &i) < 0)

    return -1;

  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)

    return -1;

  \*pp = (char\*)i;

  return 0;

}

// Fetch the nth word-sized system call argument as a string pointer.

// Check that the pointer is valid and the string is nul-terminated.

// (There is no shared writable memory, so the string can't change

// between this check and being used by the kernel.)

int

argstr(int n, char \*\*pp)

{

  int addr;

  if(argint(n, &addr) < 0)

    return -1;

  return fetchstr(addr, pp);

}

extern int sys\_chdir(void);

extern int sys\_close(void);

extern int sys\_dup(void);

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\_weightset(void);

static int (\*syscalls[])(void) = {

[SYS\_fork]    sys\_fork,

[SYS\_exit]    sys\_exit,

[SYS\_wait]    sys\_wait,

[SYS\_pipe]    sys\_pipe,

[SYS\_read]    sys\_read,

[SYS\_kill]    sys\_kill,

[SYS\_exec]    sys\_exec,

[SYS\_fstat]   sys\_fstat,

[SYS\_chdir]   sys\_chdir,

[SYS\_dup]     sys\_dup,

[SYS\_getpid]  sys\_getpid,

[SYS\_sbrk]    sys\_sbrk,

[SYS\_sleep]   sys\_sleep,

[SYS\_uptime]  sys\_uptime,

[SYS\_open]    sys\_open,

[SYS\_write]   sys\_write,

[SYS\_mknod]   sys\_mknod,

[SYS\_unlink]  sys\_unlink,

[SYS\_link]    sys\_link,

[SYS\_mkdir]   sys\_mkdir,

[SYS\_close]   sys\_close,

[SYS\_weightset] sys\_weightset,

};

void

syscall(void)

{

  int num;

  struct proc \*curproc = myproc();

  num = curproc->tf->eax;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    curproc->tf->eax = syscalls[num]();

  } else {

    cprintf("%d %s: unknown sys call %d\n",

            curproc->pid, curproc->name, num);

    curproc->tf->eax = -1;

  }

}

**syscall.h**

// 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\_weightset  22

**sysproc.c**

#include "types.h"

#include "x86.h"

#include "defs.h"

#include "date.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

int

sys\_fork(void)

{

  return fork();

}

int

sys\_exit(void)

{

  exit();

  return 0;  // not reached

}

int

sys\_wait(void)

{

  return wait();

}

int

sys\_kill(void)

{

  int pid;

  if(argint(0, &pid) < 0)

    return -1;

  return kill(pid);

}

int

sys\_getpid(void)

{

  return myproc()->pid;

}

int

sys\_sbrk(void)

{

  int addr;

  int n;

  if(argint(0, &n) < 0)

    return -1;

  addr = myproc()->sz;

  if(growproc(n) < 0)

    return -1;

  return addr;

}

int

sys\_sleep(void)

{

  int n;

  uint ticks0;

  if(argint(0, &n) < 0)

    return -1;

  acquire(&tickslock);

  ticks0 = ticks;

  while(ticks - ticks0 < n){

    if(myproc()->killed){

      release(&tickslock);

      return -1;

    }

    sleep(&ticks, &tickslock);

  }

  release(&tickslock);

  return 0;

}

// return how many clock tick interrupts have occurred

// since start.

int

sys\_uptime(void)

{

  uint xticks;

  acquire(&tickslock);

  xticks = ticks;

  release(&tickslock);

  return xticks;

}

//\*\*\*\*\*\*\*\*\*\*20202925\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int

sys\_weightset(void)

{

  int weight;

  if(argint(0, &weight) < 0)

    return -1;

  if(weight==0)

    return -1;

  do\_weightset(weight);

  return 0;

}

**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, const void\*, int);

int read(int, void\*, int);

int close(int);

int kill(int);

int exec(char\*, char\*\*);

int open(const char\*, int);

int mknod(const char\*, short, short);

int unlink(const char\*);

int fstat(int fd, struct stat\*);

int link(const char\*, const char\*);

int mkdir(const char\*);

int chdir(const char\*);

int dup(int);

int getpid(void);

char\* sbrk(int);

int sleep(int);

int uptime(void);

int weightset(uint);

// ulib.c

int stat(const char\*, struct stat\*);

char\* strcpy(char\*, const char\*);

void \*memmove(void\*, const void\*, int);

char\* strchr(const char\*, char c);

int strcmp(const char\*, const char\*);

void printf(int, const char\*, ...);

char\* gets(char\*, int max);

uint strlen(const char\*);

void\* memset(void\*, int, uint);

void\* malloc(uint);

void free(void\*);

int atoi(const char\*);

**usys.S**

#include "syscall.h"

#include "traps.h"

#define SYSCALL(name) \

.globl name; \

name: \

movl $SYS\_ ## name, %eax; \

int $T\_SYSCALL; \

ret

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(weightset)