컴퓨터학부, 20202925, 강준현

1.개요

기존 시스템콜의 구현과 호출과정 등 시스템콜의 전반적인 이해를 해보고, 새 시스템콜을 직접 구현하고 추가해 보며 시스템콜의 대한 이해도를 높혀본다. 또한 이과정에서 xv6 커널을 이해해 본다. memsize를 시스템 콜으라 통해 호출한 프로세스의 메모리 사용량을 출력해본다. 이를 통해 proc등 의 이해도를 높혀본다. trace라는 시스템콜을 구현해 보며 시스템콜의 전반적인 호출과 리턴에 대한 과정을 이해해본다.

2.상세설계

memsize: proc구조체의 sz를 이용한다.

trace

시스템콜 호출과정

시스템콜 호출 -> usys.S -> trapasm.S -> trap.c -> syscall.c

trace 에서는 proc 구조체에 mask에 받아온 mask저장

syscall.c 의 syscall 함수 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) 부분은 시스템콜 호출 성공 부분이다. 이부분에 분기문을 이용하여 num을 비트 쉬프트를 해 mask와 &로 추적의 대상을 찾는다. curproc->tf->eax = syscalls[num](); 부분은 system call의 리턴값을 저장하는 부분이므로 호출문에 리턴값을 출력하는 데 이용한다. pid 는 현재 proc구조체의 pid를 이용하여 출력한다. syscall 함수 내부에 cprintf를 이용해 trace 결과를 출력해준다. 그 이유는 system call 이 호출될 때마다 출력을 해줘야하는데 system call 호출 과정마다 무조건 지나가는 부분이기 때문이다.

3.구현방법

1. memsize: myproc()을 통해 현재 프로세스의 proc정보를 얻어와 proc구조체의 sz를 통해 현재 프로세스의 메모리 사용량을 return해준다. 이를 memsizetest.c라는 쉘프로그램에서 출력해 사용자에게 보여준다.

2. trace: ssu\_trace.c 쉘프로그램에서 마스크값, 명령어를 입력받는다. argv[1]을 atoi함수를 통해 mask값으로 변환하고, 이 mask값은 1<<시스템콜번호로 들어온다. argv[2] 부터 끝까지 모두 argv\_2라는 새로운 곳에 옮겨 명령어만 뽑아준다. 그리고 trace(mask)를 통해 우선 trace시스템콜을 호출한 다음. exec(argv[2], argv\_2)를 통해 명령어를 실행해준다. proc구조체에mask값을 넣고fork 함수에 부모의 마스크 값을 자식 프로세스에 복사하는 줄을 추가하여 생성되는 모든 자식 프로세스에서도 mask값을 통해 trace를 수행할 수 있게한다.

trace시스템콜에서는 if(argint(0, &mask) < 0) return -1;을 통해 인자로 넘어온 mask값을 myproc()->mask = mask로 할당해 mask값을 proc구조체에 저장한다.

syscall.c 의 syscall 함수에 시스템콜이 성공적으로 호출되는 부분에 mask\_check = (curproc->mask & (1<<num)); 비트 &연산을 통해 해당 mask값을 통해 어느 시스템콜 번호인지 알아내도록 한다. if 와else if를 반복해 mask\_check == (1<<1), mask\_check == (1<<2) ... 를 통해 어느 시스템콜을 trace하는 것인지 알아내고 그 시스템콜이면,         cprintf("syscall traced: pid = %d, syscall = fork, %d returned\n", curproc->pid, curproc->tf->eax); 를 통해 pid, systemcall 이름, system call 리턴값을 출력해준다.

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

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

    //trace() call, 비트연산자 & 로 mask값에 맞는 함수들만 trace()호출

    int mask\_check = (curproc->mask & (1<<num));

    if(mask\_check == (1<<1))

        cprintf("syscall traced: pid = %d, syscall = fork, %d returned\n", curproc->pid, curproc->tf->eax);

...

4.결과

malloc 전, 후 차이가 2048byte가 아닌 이유

malloc 함수에

nbytes는 요청한 size이고 이를 nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1; 를 통해 nbytes를 위해 필요한 units들과 header를 위한 1을 더해 numits에 할당한다. 이후 morecore라는 함수를 호출해 sbrk를 통해 메모리를 할당하게 되는데 이때 morecore함수에서 nu값이 4096 보다 작으면 4096 을 할당하게 되어있다. 이는 page를 할당히기 때문에 1page=4096이므로 2048을 입력해도 4096만큼 할당한다.

static Header\*

morecore(uint nu)

{

char \*p;

Header \*hp;

if(nu < 4096)

nu = 4096;

p = sbrk(nu \* sizeof(Header));

if(p == (char\*)-1)

return 0;

hp = (Header\*)p;

hp->s.size = nu;

free((void\*)(hp + 1));

return freep;

}

void\*

malloc(uint nbytes)

{

Header \*p, \*prevp;

uint nunits;

nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;

if((prevp = freep) == 0){

base.s.ptr = freep = prevp = &base;

base.s.size = 0;

}

for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){

if(p->s.size >= nunits){

if(p->s.size == nunits)

prevp->s.ptr = p->s.ptr;

else {

p->s.size -= nunits;

p += p->s.size;

p->s.size = nunits;

}

freep = prevp;

return (void\*)(p + 1);

}

if(p == freep)

if((p = morecore(nunits)) == 0)

return 0;

}

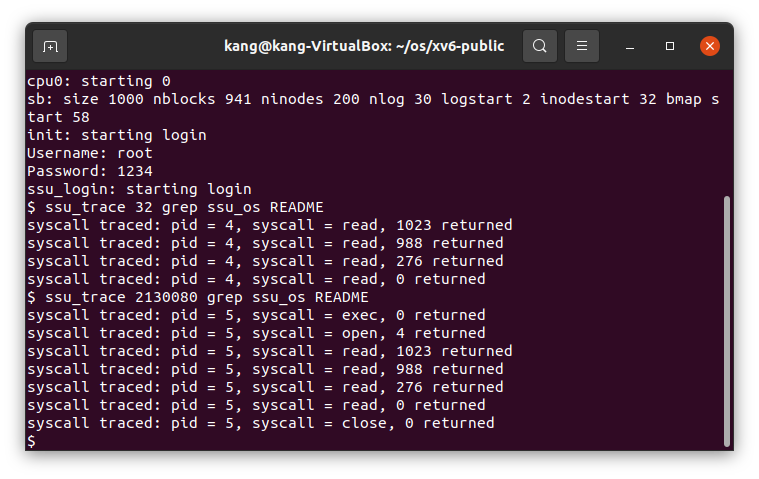
}

memsizetest.c --> memsize텍스트이(가) 표시된 사진

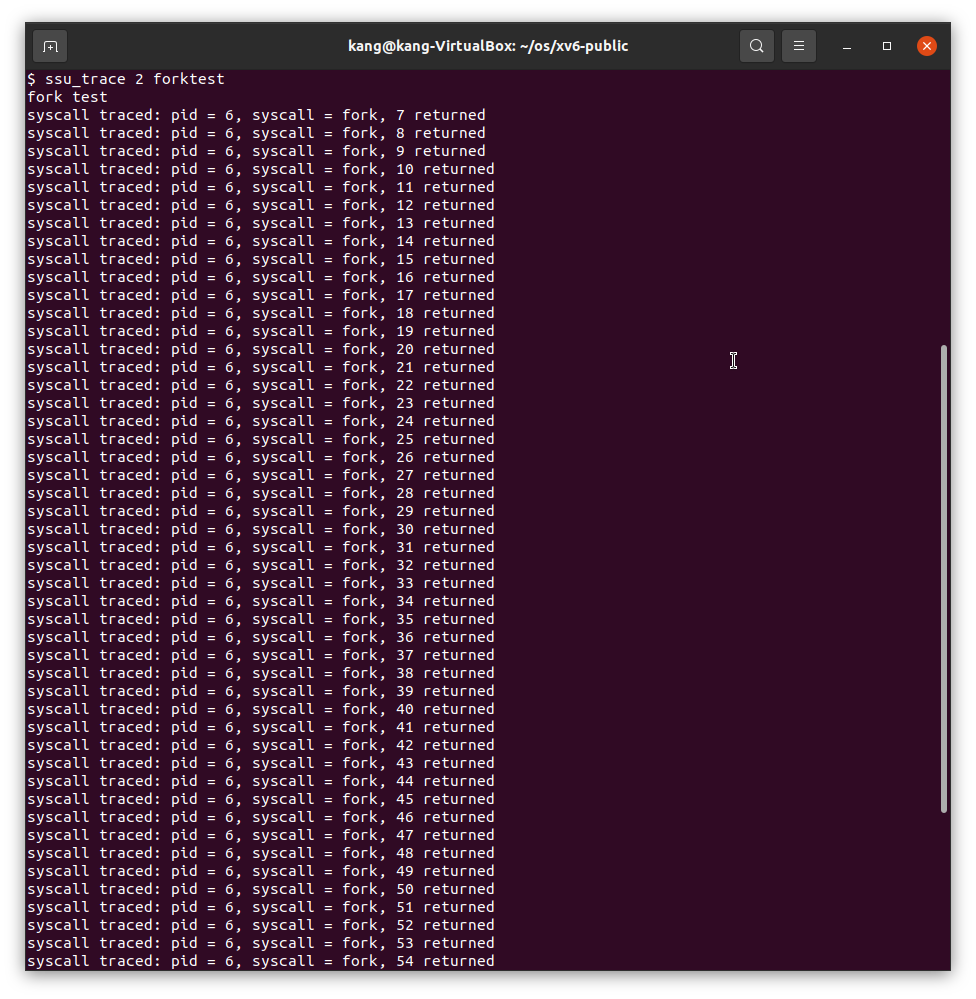
자동 생성된 설명

ssu\_trace 32 grep ssu\_os README (open)

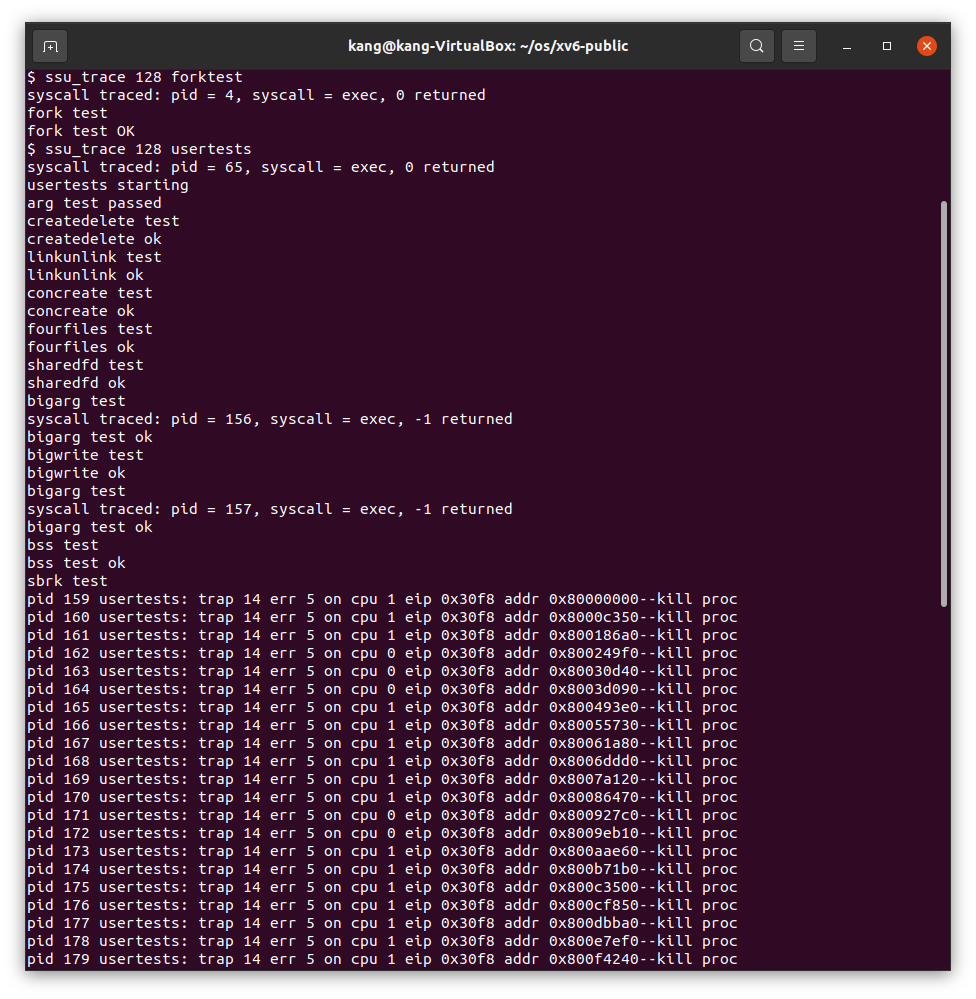
ssu\_trace 2130080 grep ssu\_os README (exec, open, read, close)



ssu\_trace 2 forktest (fork)



ssu\_trace 128 usertests (exec)   
모든 자식 프로세스에 대한 trace mask 활성화



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

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\

    \_memsizetest\

    \_ssu\_trace\

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 := 2

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 memsizetest.c ssu\_trace.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.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"

struct {

  struct spinlock lock;

  struct proc proc[NPROC];

} ptable;

static struct proc \*initproc;

int nextpid = 1;

extern void forkret(void);

extern void trapret(void);

static void wakeup1(void \*chan);

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:

  p->state = EMBRYO;

  p->pid = nextpid++;

  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[];

  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;

  np->mask = curproc->mask; // mask for tracing in child.

  // 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();

    // 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);

  }

}

// 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;

}

// 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");

  }

}

**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)

  int mask;            // Mask for trace

};

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

//   text

//   original data and bss

//   fixed-size stack

//   expandable heap

**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\_memsize(void);

extern int sys\_trace(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\_memsize]   sys\_memsize,

[SYS\_trace]   sys\_trace,

};

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]();

    //trace() call, 비트연산자 & 로 mask값에 맞는 함수들만 trace()호출

    int mask\_check = (curproc->mask & (1<<num));

    if(mask\_check == (1<<1))

        cprintf("syscall traced: pid = %d, syscall = fork, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<2))

        cprintf("syscall traced: pid = %d, syscall = exit, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<3))

        cprintf("syscall traced: pid = %d, syscall = wait, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<4))

        cprintf("syscall traced: pid = %d, syscall = pipe, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<5))

        cprintf("syscall traced: pid = %d, syscall = read, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<6))

        cprintf("syscall traced: pid = %d, syscall = kill, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<7))

        cprintf("syscall traced: pid = %d, syscall = exec, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<8))

        cprintf("syscall traced: pid = %d, syscall = fstat, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<9))

        cprintf("syscall traced: pid = %d, syscall = chdir, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<10))

        cprintf("syscall traced: pid = %d, syscall = dup, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<11))

        cprintf("syscall traced: pid = %d, syscall = getpid, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<12))

        cprintf("syscall traced: pid = %d, syscall = sbrk, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<13))

        cprintf("syscall traced: pid = %d, syscall = sleep, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<14))

        cprintf("syscall traced: pid = %d, syscall = uptime, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<15))

        cprintf("syscall traced: pid = %d, syscall = open, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<16))

        cprintf("syscall traced: pid = %d, syscall = write, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<17))

        cprintf("syscall traced: pid = %d, syscall = mknod, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<18))

        cprintf("syscall traced: pid = %d, syscall = unlink, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<19))

        cprintf("syscall traced: pid = %d, syscall = link, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<20))

        cprintf("syscall traced: pid = %d, syscall = mkdir, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<21))

        cprintf("syscall traced: pid = %d, syscall = close, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<22))

        cprintf("syscall traced: pid = %d, syscall = memsize, %d returned\n", curproc->pid, curproc->tf->eax);

    else if(mask\_check == (1<<23))

        cprintf("syscall traced: pid = %d, syscall = trace, %d returned\n", curproc->pid, curproc->tf->eax);

  } 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\_memsize  22

#define SYS\_trace  23

**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;

}

// return process memory usage

int

sys\_memsize(void)

{

  uint size;

  size = myproc()->sz;

  return size;

}

//trace

int

sys\_trace(void)

{

  //pid, return, name

//  int pid, r;

//  char \*name;

  int mask;

if(argint(0, &mask) < 0)

    return -1;

  myproc()->mask=mask;

//  printf(1,"syscall traced: pid = %d, syscall = %s, %d returned\n", pid, name, r);

  return 1;

}

**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 memsize(void);

int trace(int);

// 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(memsize)

SYSCALL(trace)

**memsizetest.c**

#include "types.h"

#include "stat.h"

#include "user.h"

#define SIZE 2048

int main(void)

{

    int msize = memsize();

    printf(1, "The process is using %dB\n", msize);

    char \*tmp = (char \*)malloc(SIZE \* sizeof(char));

    printf(1, "Allocating more memory\n");

    msize = memsize();

    printf(1,"The process is using %dB\n", msize);

    free(tmp);

    printf(1, "Freeing memory\n");

    msize=memsize();

    printf(1, "The process is using %dB\n", msize);

    exit();

}

**ssu\_trace.c**

#include "types.h"

#include "stat.h"

#include "user.h"

int main(int argc, char \*\*argv)

{

    int i, mask;

    char \*argv\_2[100];

    if(argc<3){

        printf(1,"usage error ssu\_trace\n");

        exit();

    }

    mask=atoi(argv[1]);

    for(i=0 ; i<argc; i++){

        argv\_2[i]=malloc(100);

        argv\_2[i]=argv[i+2];

    }

    trace(mask);

    exec(argv[2], argv\_2);

    exit();

}