1. 개요

본 과제는 교육용 운영체제 xv6의 설치 및 컴파일 환경을 구축하고, 사용자와 커널 간의 인터페이스인 시스템 콜을 확장하는 실습을 목표로 한다. 구체적으로, 특정 정수를 기반으로 메시지를 출력하고 연산 결과를 반환하는 helloxv6()와 특정 프로세스의 pid, ppid, 상태, 메모리 크기, 이름을 조회하는 get_procinfo() 시스템 콜을 구현한다. 각각의 시스템 콜을 호출하여 동작을 검증할 응용프로그램인 helloxv6.c와 psinfo.c를 작성하고 셸에서 실행함으로써, 시스템 콜 경로, 인자 전달 등 OS의 핵심 원리를 학습한다.

2. 상세설계

2-1. xv6 설치 및 컴파일

2-1-1. xv6 다운로드

```
ubuntu@OS:~$ git clone https://github.com/mit-pdos/xv6-public
Cloning into 'xv6-public'...
remote: Enumerating objects: 13990, done.
remote: Total 13990 (delta 0), reused 0 (delta 0), pack-reused 13990 (from 1)
Receiving objects: 100% (13990/13990), 17.24 MiB | 2.60 MiB/s, done.
Resolving deltas: 100% (9466/9466), done.
ubuntu@OS:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  xv6-public
```

설치 완료 및 디렉토리 생성된 것을 확인한다.

2-1-2. qemu 다운로드

```
ubuntu@OS:~/xv6-public$ sudo apt-get install qemu-kvm
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  cpu-checker ibverbs-providers ipxe-qemu ipxe-qemu-256k-compat-efi-roms libcacard0 libfdt1 libibverbs1 libiscsi7 libpmem1
```

컴파일 및 실행을 위해 qemu를 다운한다.

2-1-3. xv6 컴파일 및 실행

```
ubuntu@OS:~/xv6-public$ make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-
 -fno-pie -no-pie -fno-pic -O -nostdinc -I. -c bootmain.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-
 -fno-pie -no-pie -fno-pic -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -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
boot block is 467 bytes (max 510)
```

make 실행 결과의 일부이다. 이를 통해 컴파일을 진행한다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk...
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

make qemu-nox 실행 결과이다. 새로운 창 없이 xv6를 실행한다.

2-1-4. 실행 테스트

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16320
echo           2 4 15172
forktest       2 5 9480
grep           2 6 18540
init           2 7 15760
kill           2 8 15204
ln             2 9 15060
ls             2 10 17688
mkdir          2 11 15300
rm             2 12 15280
sh             2 13 27924
stressfs       2 14 16192
usertests      2 15 67300
wc             2 16 17056
zombie         2 17 14880
helloxv6       2 18 15264
psinfo         2 19 15876
console        3 20 0
$
```

ls 실행결과 정상적으로 결과가 나온 것을 볼 수 있다.

2-2. xv6 시스템콜 개념 및 추가 절차

2-2-1. 시스템 콜 개념

OS는 보안성과 안정성을 위해 주소공간을 유저 영역(user-space)와 커널 영역(kernel space)로 분리한다. 사용자 프로그램은 직접 하드웨어나 핵심 자료구조에 접근할 수 없고, 대신 OS가 제공하는 인터페이스인 시스템 콜(system call)을 통해서만 커널 기능을 이용할 수 있다. 이때, 시스템 콜 호출은 일반 함수 호출과 달리 특권 모드 전환이 필요하기 때문에 CPU는 트랩(trap)이라는 메커니즘을 제공한다. 이는 소프트웨어 인터럽트의 일종으로, 사용자 모드에서 커널 모드로 안전하게 제어권을 넘기는 통로 역할을 한다.

2-2-2. 시스템 콜 추가 절차

xv6에서 새로운 시스템 콜을 추가하는 절차는 다음과 같이 요약된다.

i) user.h에 사용자 인터페이스(프로토타입)와 구조체 정의를 추가한다.

ii) usys.S에 SYSCALL(함수이름)을 등록하여, 사용자 함수 호출이 내부적으로 트랩을 발생시키도록 한다.

iii) syscall.h에 SYS_함수이름 매크로를 정의하여 시스템 콜 번호를 고유하게 매핑한다.

iv) syscall.c의 디스패치 테이블에서 번호와 커널 핸들러를 연결한다.

v) sysproc.c에서 실제 커널 핸들러를 구현하여 인자검증, 로직 실행, 결과 반환을 한다.

이 절차를 통해 사용자 프로그램에서의 함수 호출이 커널 내부의 로직 실행으로 안전하게 이어지도록 전체 경로를 연결한다.

2-2-3. 기존 소스코드 분석

xv6는 이미 여러 시스템 콜을 구현하고 있으며, 그 경로를 통해 사용자 함수 호출이 커널까지 이어지는 구조를 이해할 수 있다. 먼저 usys.S에는 SYSCALL(fork),SYSCALL(exit) 등 기존 시스템 콜의 어셈블리 스텁이 정의되어 있다. 이 매크로는 호출 시 시스템 콜 번호를 레지스터에 적재하고, 소프트웨어 인터럽트를 발생시켜 커널 모드로 진입하게 한다. syscall.h에는 각 시스템 콜의 번호가 매크로로 정의되어 있으며, 이 번호는 커널의 syscall.c에서 사용된다. syscall.c에는 syscalls[] 배열이 존재하여 시스템 콜 번호와 커널 핸들러 함수 포인터가 매핑된다. 마지막으로 trap.c는 CPU에서 발생한 소프트웨어 인터럽트(시스템 콜 요청)를 처리하여 syscall() 함수를 호출하고, 이 함수가 번호 기반으로 올바른 커널 함수를 실행한다. 이러한 구조 분석을 통해, xv6의 시스템 콜 확장은 단순히 새 함수를 추가하는 것이 아니라, 기존의 일관된 경로에 새 항목을 연결하는 작업임을 확인할 수 있다.

2-3. hello_number()

2-3-1. 개요

hello_number()는 본 과제에서 추가한 첫 번째 시스템 콜로, 사용자가 지정한 정수를 커널 내부에서 확인한 후, 메시지 출력한 뒤 해당 정수의 2배를 다시 사용자 프로그램으로 반환한다. 이 기능은 커널의 sysproc.c에 구현된 sys_hello_number() 함수에서 수행되며, 테스트용 사용자 프로그램 helloxv6.c를 통해 호출된다. 단순한 산술 연산과 출력이지만, 사용자 호출이 OS 커널을 거쳐 다시 사용자 공간으로 결과가 되돌아오는 전형적인 시스템 콜의 구조를 확인할 수 있다. 이 시스템 콜은 usys.S의 스텁에서 소프트웨어 인터럽트를 발생시키고, trap.c의 syscall()을 거쳐 syscall.c의 디스패처가 sys_hello_number()로 연결된 뒤 결과를 사용자 공간으로 반환한다.

2-3-2. 프로토타입

user.h에 아래와 같이 사용자 인터페이스가 선언되어 있다.

```
int hello_number(int n);
```

이 함수는 하나의 정수를 입력 받아, 커널 핸들러인 sys_hello_number()를 통해 처리된다. 커널에서는 argint()를 사용하여 사용자로부터 전달된 인자를 안전하게 가져온 뒤, cprintf()로 커널 콘솔에 메시지를 출력한다. 커널은 cprintf로 커널 콘솔에 메시지를 출력하고, 계산 결과를 반환한다. 반환값은 사용자 공간에서 printf로 출력된다.

2-3-3. 호출 흐름

i) 사용자 호출

```
#include "types.h"      // type definitions used by xv6
#include "stat.h"       // file status structures
#include "user.h"       // user-space system call interfaces

int main(int argc, char *argv[]){
        // call hello_number with test arguments
        int res = hello_number(5);
        int res2 = hello_number(-7);

        // print the return values of hello_number
        printf(1, "hello_number(5) returned %d\n", res);
        printf(1, "hello_number(-7) returned %d\n", res2);

        // terminate
        exit();
}
```

> helloxv6.c에서 사용자 호출이 시작되는 부분이자, 리턴값 출력되는 부분이다.

사용자 프로그램 helloxv6.c에서 hello_number(5)가 호출된다. 이는 일반적인 C 함수 호출처럼 보이지만, 실제로는 시스템 콜 인터페이스를 통해 커널로 제어권이 넘어간다.

ii) 어셈블리 스텁

```
SYSCALL(hello_number)
```

> usys.S에서 사용자 호출을 시스템 콜 번호 적재와 트랩 실행으로 변환하는 부분이다.

```
#define SYS_hello_number 22     // new system call number for sys_hello
```

> syscall.h에서 시스템 콜 번호를 정의한 부분이다.

```
[SYS_hello_number] sys_hello_number,  // hello_number() -> sys_hello_number()
```

> syscall.c에서 번호와 커널 함수 연결을 구현한 부분이다.

사용자 호출은 먼저 usys.S의 SYSCALL() 스텁으로 들어간다. 이 스텁은 EAX 레지스터에 시스템 콜 번호를 적재하고, 트랩을 실행하여 커널로 진입한다. 여기서 사용되는 번호는 syscall.h에 정의되어 있으며, 커널 쪽에서는 syscall.c의 syscalls[]테이블에서 매핑되어 디스패치될 준비가 되어있다. 즉, usys.S가 트랩을 발생시키고, syscall.h와 syscall.c를 통해 번호가 핸들러로 연결된다.

iii) 시스템 콜 번호 확인

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

> 기구현된 trap.c()에서 트랩 번호를 검사하는 부분이다.

```
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()이 eax에서 번호를 읽는 부분이다.

커널로 진입하면 trap.c의 trap() 함수가 트랩 번호를 확인한 뒤 syscall() 함수를 호출한다. syscall()은 트랩 프레임의 EAX에 저장된 시스템 콜 번호를 읽고, syscalls[] 배열에서 해당 번호를 찾아 함수 포인터를 실행한다. 이때 SYS_hello_number는 sys_hello_number()와 연결되어 있다.

iv) 커널 핸들러 실행

```
// sys_hello_number
// return n*2 back to user space
int
sys_hello_number(void){
        int n;
        if(argint(0,&n) <0) return -1;  // fetch arg #0 into n
        cprintf("Hello, xv6! Your number is %d\n",n); // print funcion in kernel
        return n*2;
}
```

> sysproc.c 에서 sys_hello_number()를 구현한 부분이다.

커널의 sysproc.c에 정의된 sys_hello_number()가 호출되면 argint()로 사용자 프로그램이 전달한 첫번째 인자를 읽어오고, cprintf로 커널 콘솔에 메시지를 출력한다. 마지막으로 return으로 계산된 값을 반환하는데, 이 반환값은 트랩 프레임의 EAX에 기록된다.

v) 사용자 영역 복귀

커널이 반환한 값은 트랩 프레임을 통해 사용자 공간으로 복귀한다. 이후 helloxv6.c의 printf()가 이 값을 출력한다. 실행 화면에서는 커널 출력(cprintf 메시지)과 사용자 출력(printf 결과)이 구분되어 나타난다.

2-4. get_procinfo()

2-4-1. 개요

이 함수는 지정된 프로세스의 정보를 사용자 프로그램이 조회할 수 있도록 설계되었다. 이 시스템 콜은 프로세스의 PID, 부모 PID, 현재 상태, 주소 공간 크기, 그리고 이름을 반환하며, 사용자 프로그램이 전달한 버퍼에 구조체 형태로 기록된다. 커널 내부에서는 sysproc.c의 sys_get_procinfo() 함수로 구현되었으며, 프로세스 테이블에 접근할 때는 ptable.lock을 이용하여 동기화 문제를 방지한다.

2-4-2. 프로토타입

사용자 인터페이스가 user.h에 아래와 같이 정의되어 있다.

```
struct procinfo{
    int pid;
    int ppid;
    int state;
    uint sz;
    char name[16];
};
int get_procinfo(int pid, struct procinfo *uinfo);
```

i) 입력 인자 : pid(조회 대상 pid, 0이하면 자신), uinfo(사용자 영역 구조체 포인터)

ii) 출력 값 : 성공 시 구조체가 채워지고 0 리턴, 실패시 -1 리턴.

2-4-3. 호출 흐름

i) 사용자 호출

```
#include "types.h"      // type definitions used by xv6
#include "stat.h"       // file status structures
#include "user.h"       // user-space system call interfaces

// convert numeric process state to string
static char* s2str(int s){
        switch(s){
                case 0 : return "UNUSED";
                case 1 : return "EMBRYO";
                case 2 : return "SLEEPING";
                case 3 : return "RUNNABLE";
                case 4 : return "RUNNING";
                case 5 : return "ZOMBIE";
        }
        return "UNKNOWN";
}

int main(int argc, char *argv[]){
        struct procinfo info;
        int pid = (argc >= 2) ? atoi(argv[1]) : 0;      // if no argument, pid = 0 is "self"

        // call get_procinfo() to fetch process info
        if(get_procinfo(pid, &info) < 0){
                printf(2, "psinfo : get_procinfo() failed\n");
                exit();
        }
        // print process info
        printf(1, "PID=%d PPID=%d STATE=%s SZ=%d NAME=%s\n",info.pid,info.ppid,s2str(info.state),info.sz,info.name);

        // terminate the program
        exit();
}
```

> psinfo.c 에서 get_procinfo()를 호출하는 부분.

사용자 프로그램 psinfo.c의 main()에서 get_procinfo()가 호출된다. 인자가 없으면 자기자신이며 특정 숫자를 주면 해당 프로세스를 조회한다.

ii) 어셈블리 스텁

```
SYSCALL(get_procinfo)
```

> usys.S에서 사용자 호출을 시스템 콜 번호 적재와 트랩 실행으로 변환하는 부분이다.

```
#define SYS_get_procinfo 23             // new system call number for sys_procinfo
```

> syscall.h에서 시스템 콜 번호를 정의한 부분이다.

```
[SYS_get_procinfo] sys_get_procinfo,  // get_procinfo() -> sys_get_procinfo()
```

> syscall.c에서 번호와 커널 함수 연결을 구현한 부분이다.

usys.S에 정의된 SYSCALL() 스텁이 실행된다. 이 스텁은 EAX 레지스터에 SYS_get_procinfo 번호를 적재하고, 트랩 명령으로 SW 인터럽트를 발생시켜 커널로 진입된다.

iii) 시스템 콜 번호 확인

helloxv6에서 설명했던 방식과 동일하게 커널 진입 후 trap함수가 syscall()을 호출하고, syscalls[]배열에서 해당번호를 찾아 실행된다.

iv) 커널 핸들러 실행

```
// sys_get_procinfo
// copy selected fields of the target process into user buffer
// reference ptable in proc.c
extern struct{
        struct spinlock lock;
        struct proc proc[NPROC];
}ptable;
struct k_procinfo{
        int pid,ppid,state;
        uint sz;
        char name[16];
};

int
sys_get_procinfo(void){
       int pid;
       char *uaddr;             // destination buffer
       struct proc *p, *t;
       struct k_procinfo kinfo; // ?

  // fetch args : pid, user buffer pointer
       if(argint(0, &pid) < 0) return -1;
       if(argptr(1, &uaddr, sizeof(struct k_procinfo)) < 0) return -1;

  acquire(&ptable.lock);
       // resolve target : pid <= 0 is "self"
  if(pid <= 0) t = myproc();
       else {
               t = 0;
               for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
                       if(p->pid == pid){ t = p; break; }
       }
  // validate existence and state
       if(t == 0 || t->state == UNUSED){
               release(&ptable.lock);
               return -1;
       }

       // fill kernel-side struct
       kinfo.pid = t->pid;
       kinfo.ppid = t->parent ? t->parent->pid : 0;
       kinfo.state = t->state;
       kinfo.sz = t->sz;
       safestrcpy(kinfo.name,t->name,sizeof(kinfo.name));
       release(&ptable.lock);

       // copy to user-side space
       if(copyout(myproc()->pgdir,(uint)uaddr, (void*)&kinfo, sizeof(kinfo)) < 0) return -1;
       return 0;
}
```

> sysproc.c에서 구현한 sys_get_procinfo() 함수이다.

커널의 sysproc.c에 구현된 sys_get_procinfo()가 호출되면, argint()와 argptr()로 사용자 인자를 검증하고 버퍼포인터를 얻는다. ptable.lock을 획득한 후, 프로세스 테이블을 탐색하여 pid의 프로세스를 찾고, 유효성을 검사한다. 만약 유효하다면 procinfo 구조체를 채워서 copyout()으로 사용자 버퍼에 복사한다.

v) 사용자 영역 복귀

성공 시 사용자 버퍼 info에 프로세스 정보가 저장되고, 리턴값으로 0이 사용자 프로그램으로 전달된다. psinfo.c는 이 반환값으로 성공하면 프로세스 정보 출력, 실패하면 오류 메시지를 출력한다.

2-5. Makefile

2-5-1. 개요

xv6에서 사용자 프로그램을 QEMU로 부팅되는 파일시스템 이미지에 포함시키는 작업은 Makefile의 사용자 프로그램 목록을 통해 이루어진다. 커널 소스 수정만으로는 실행파일이 이미지에 들어가지 않으며, 빌드시 사용자 프로그램을 컴파일,링크하고 이를 이미지파일로 패킹하는 단계가 필요하다. 이번 과제에서 추가한 두 사용자 함수를 검증하기 위한 사용자 프로그램을 이미지에 포함되도록 빌드 시스템에 통합하기 위한 과정이 Makefile 수정이다.

2-5-2. 수정사항

```
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _helloxv6\
        _psinfo\
```

UPROGS 변수에 _helloxv6,_psinfo를 등록하였다. 따라서 사용자실행 파일로 빌드되어 파일시스템 이미지에 포함된다.

3. 결과

```
init: starting sh
$ helloxv6
Hello, xv6! Your number is 5
Hello, xv6! Your number is -7
hello_number(5) returned 10
hello_number(-7) returned -14
$
$ psinfo
PID=5 PPID=2 STATE=RUNNING SZ=12288 NAME=psinfo
$
$ psinfo 1
PID=1 PPID=0 STATE=SLEEPING SZ=12288 NAME=init
$
$ psinfo 2
PID=2 PPID=1 STATE=SLEEPING SZ=16384 NAME=sh
$
$ psinfo 3
psinfo : get_procinfo() failed
$ psinfo 999
psinfo : get_procinfo() failed
$
```

3-1. helloxv6 실행결과

helloxv6 실행결과, 커널 Hello,... 메시지가 각각 출력이 되었다. 이는 커널의 cprintf() 호출에 의해 출력된 것이다. 동시에 사용자 프로그램의 printf()는 반환값을 출력하였다. 입력값 5에 대해 두 배인 10, 입력값 -7에 대해 두 배인 -14가 반환되어, 인자 전달과 반환값 처리가 올바르게 수행되었음을 확인할 수 있다.

3-2. psinfo 실행결과

psinfo 프로그램을 통해 get_procinfo()를 호출한 결과는 다음과 같다.

i) 인자 없이 실행 (pid=0, 자기 자신 조회)

자신의 프로세스(psinfo)의 pid, ppid, state, 메모리 크기, 이름이 정상적으로 출력되었다.

ii) 특정 pid 조회 (pid=1, pid=2)

init 프로세스와 sh 프로세스의 정보가 정확히 출력되었다.

iii) 잘못된 pid 조회 (pid=3, pid=999)

존재하지 않는 pid를 입력했을 때 -1을 반환하고, 사용자 프로그램이 실패 메시지를 출력하였다.

4. 소스코드

4-1. user.h

```
...
// new user-space syscall interface
// struct for get_procinfo()
struct procinfo{
      int pid;            // target pid
      int ppid;           // parent pid
      int state;          // procstate value
      uint sz;            // address-space size
      char name[16];   // process name
};
// prototype for new system calls
int hello_number(int n);
int get_procinfo(int pid, struct procinfo *uinfo);
```

4-2. usys.S

```
...
# user-space stubs for new syscalls
SYSCALL(hello_number)
SYSCALL(get_procinfo)
```

4-3. syscall.h

```
...
#define SYS_hello_number 22        // new system call number for sys_hello
#define SYS_get_procinfo 23        // new system call number for sys_procinfo
```

4-4. syscall.c

```
...
// new kernel handlers
extern int sys_hello_number(void);
extern int sys_get_procinfo(void);
...
static int (*syscalls[])(void) = {
(다른 시스템콜 함수들)
[SYS_hello_number] sys_hello_number,   // hello_number() -> sys_hello_number()
[SYS_get_procinfo] sys_get_procinfo,   // get_procinfo() -> sys_get_procinfo()
};
...
```

4-5. sysproc.c

```
...
#include "proc.h"            // for struct proc, myproc()...
#include "spinlock.h"     // for struct spinlock in extern ptable
...
// sys_hello_number
// return n*2 back to user space
int
sys_hello_number(void){
   int n;
   if(argint(0,&n) <0) return -1;   // fetch arg #0 into n
   cprintf("Hello, xv6! Your number is %d₩n",n); // print funcion in kernel
   return n*2;
```

```
}

// sys_get_procinfo
// copy selected fields of the target process into user buffer
// reference ptable in proc.c
extern struct{
  struct spinlock lock;
  struct proc proc[NPROC];
}ptable;
struct k_procinfo{
  int pid,ppid,state;
  uint sz;
  char name[16];
};

int
sys_get_procinfo(void){
  int pid;
  char *uaddr;                    // destination buffer
  struct proc *p, *t;
  struct k_procinfo kinfo; // ?

  // fetch args : pid, user buffer pointer
  if(argint(0, &pid) < 0) return -1;
  if(argptr(1, &uaddr, sizeof(struct k_procinfo)) < 0) return -1;

  acquire(&ptable.lock);
  // resolve target : pid <= 0 is "self"
  if(pid <= 0) t = myproc();
  else {
    t = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      if(p->pid == pid){ t = p; break; }
  }
  // validate existence and state
  if(t == 0 || t->state == UNUSED){
    release(&ptable.lock);
    return -1;
  }

  // fill kernel-side struct
  kinfo.pid = t->pid;
  kinfo.ppid = t->parent ? t->parent->pid : 0;
  kinfo.state = t->state;
  kinfo.sz = t->sz;
  safestrcpy(kinfo.name,t->name,sizeof(kinfo.name));
  release(&ptable.lock);

  // copy to user-side space
```

```
    if(copyout(myproc()->pgdir,(uint)uaddr, (void*)&kinfo, sizeof(kinfo)) < 0) return -1;
    return 0;
}
```

4-6. helloxv6.c

```
#include "types.h"   // type definitions used by xv6
#include "stat.h"    // file status structures
#include "user.h"    // user-space system call interfaces

int main(int argc, char *argv[]){
    // call hello_number with test arguments
    int res = hello_number(5);
    int res2 = hello_number(-7);

    // print the return values of hello_number
    printf(1, "hello_number(5) returned %d₩n", res);
    printf(1, "hello_number(-7) returned %d₩n", res2);

    // terminate
    exit();
}
```

4-7. psinfo.c

```
#include "types.h"   // type definitions used by xv6
#include "stat.h"    // file status structures
#include "user.h"    // user-space system call interfaces

// convert numeric process state to string
static char* s2str(int s){
    switch(s){
        case 0 : return "UNUSED";
        case 1 : return "EMBRYO";
        case 2 : return "SLEEPING";
        case 3 : return "RUNNABLE";
        case 4 : return "RUNNING";
        case 5 : return "ZOMBIE";
    }
    return "UNKNOWN";
}

int main(int argc, char *argv[]){
    struct procinfo info;
    int pid = (argc >= 2) ? atoi(argv[1]) : 0;   // if no argument, pid = 0 is "self"

    // call get_procinfo() to fetch process info
    if(get_procinfo(pid, &info) < 0){
        printf(2, "psinfo : get_procinfo() failed₩n");
        exit();
    }
    // print process info
    printf(1, "PID=%d PPID=%d STATE=%s SZ=%d NAME=%s₩n",info.pid,info.ppid,s2str(info.state),info.sz,info.name);
```

```
        // terminate the program
        exit();
    }
```

4-8. Makefile

```
    ...
    UPROGS=₩
        _cat₩
        _echo₩
        _forktest₩
        _grep₩
        _init₩
        _kill₩
        _ln₩
        _ls₩
        _mkdir₩
        _rm₩
        _sh₩
        _stressfs₩
        _usertests₩
        _wc₩
        _zombie₩
        _helloxv6₩
        _psinfo₩
    ...
```