ADD System Call to XV6

First of we're gonna add a new header file called proc_info.h to add proc_info structure in order to use as parameters in functions.

So add proc_info.h file and add below structure

```
#include "types.h"

struct proc_info{
   uint memsize;
   int pid;
   char name[16];
   int state;
};
```

Now we can go to syscall.h file , where each number is assigned to a different system call in this XV6 system. As you can see there are currently 21 system calls (assuming you haven't add any previous system calls :)) already defined in this file. Now we're going to add our new number to reserve for our new system call.

```
#define SYS_proc_dump 22
```

Next, you need to add a pointer to your system call in syscall.c file. This file contains an array of function pointers which uses above-defined numbers (indexes) as pointers to system calls which are defined in different locations. In order to add your custom system call, add following line to this file.

```
[SYS_proc_dump] sys_proc_dump,
```

This means, when system call occurred with system call number 22, function pointed by function pointer `sys_proc_dump` will be called. So, you have to implement this function. However, this file is not the right place to do so and we are just going to add the function prototype here.

So, find suitable place inside this file and add following line. You can see that all other 21 system call functions are defined similarly.

The function prototype which needs to be added to syscall.c file is as follows.

```
extern int sys_proc_dump(void);
```

Next, you will implement system call function. In order to do this, open sysproc.c file where system call functions are defined.

First of all add proc_info.h header file

```
#include "proc_info.h"
```

Second of all define the sys_proc_dump_function

```
// Copying 4 elements of each processes ptable to send to user space (pid, memSize, state, name)
extern struct proc_info * getptable_proc(void);

int sys_proc_dump(void){
    // these lines are for buffer ****
    int size;
    char *buf;
    char *s;
    // ****
    struct proc_info *p = '\0';

    // these function (argint, argptr) come from syscall.c file ****
    if (argint(0, &size) <0){
        return -1;
    }
    if (argptr(1, &buf,size) <0){
        return -1;
    }
    // ****
    s = buf;
    p = getptable_proc(); // from line 96

    // defining buffer size
    while(buf + size > s){
```

```
        *(int *)s = p -> pid;
        s+=4;
        *(int *)s = p->memsize;
        s+=4;
        *(int *)s = p->state;
        s+=4;
        memmove(s,p->name,16);
        s+=16;
        p++;
    }
    // return zero as successful
    return 0;
}
```

Now you have just two little files to edit and these files will contain interface for your user program to access system call. Open file called usys.S and add line below at the end.

```
SYSCALL(proc_dump)
```

Next,open file called user.h and add following line. This is function that user program will be calling. As you know now, there's no such function implemented in system. Instead, call to below function from user program will be simply mapped to system call number 22 which is defined as SYS_proc_dump preprocessor directive. The system knows what exactly is this system call and how to handle it.

```
int proc_dump(int, void *);
```

Now to complete the system call you need to change proc.c file like below:

first of all add header proc_info.h

```
#include "proc_info.h"
```

second of all add below structure:

```
// simple list of proc_info to return to user programs that call
proc_dump
struct {
    struct proc_info procInfo[NPROC];
} procinfotable;
```

third, add the proc_dump system call definition

```
//Syscall proc_dump

struct proc_info *getptable_proc(void) {
    struct proc *p;

    cprintf("------------------------------ In system call
--------------------------------\n");


    int count = 0; // to count the number of processes in the ptable to
use in procinfotable
    int pcount = 0; // to count the number of not (UNUSED) processes
    // trying to use ptable defined by xv6 creaters to initialize
procinfotable structure and
    // then send it back to user programs
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        // if the process is not UNUSED
        if (p->sz > 0) {
            procinfotable.procInfo[count].pid = p->pid;
            procinfotable.procInfo[count].memsize = p->sz;
            procinfotable.procInfo[count].state = p->state;
            int i = 0;
            // the name of process is 16 bits (chars) so we can loop on
the name and initialize the names in procinfotable.procInfo list
            while (i != 16) {
                procinfotable.procInfo[count].name[i] = p->name[i];
                i++;
```

```
            }
            // printing each process's information
            cprintf("PID: %d   -   SIZE: %d   -   STATE: %d   -   NAME:
%s\n", p->pid, p->sz, p->state, p->name);
            pcount++;
        }
        count++;
    }

    // *****
    // the remaining code is actually not important and you can
simply uncomment below code
    //            return procinfotable.procInfo;
    // you can also don't uncomment the above code and continue
with below code
    // (I'm just sorting these processes below)
    // *****

    // for those who want to know what I'm doing now , i'm try to get
the not UNUSED processes and
    // the sort them and after that return all the processes to the user
program as an array of proc_info
    // structure

    // an array of proc_info with the number of UNUSED processes
    struct proc_info listProc[pcount];

    // initialising listProc
    int i = 0;
    struct proc_info *pi;
    for (pi = procinfotable.procInfo; pi <
&procinfotable.procInfo[NPROC]; pi++) {
        if (pi->memsize > 0) {
            listProc[i].memsize = pi->memsize;
            listProc[i].pid = pi->pid;
            listProc[i].state = pi->state;
            int j = 0;
            while (j != 16) {
                listProc[i].name[j] = pi->name[j];
```

```
            j++;
        }
        i++;
    }
}

    // Using simple sort algorithm (bubble sort) to sort the processes
in listProc by process size
    for (i = 0; i < pcount - 1; ++i) {
        for (int j = 0; j < pcount - 1; ++j) {
            if (listProc[j].memsize >= listProc[j + 1].memsize) {
                int size = listProc[j].memsize;
                int state = listProc[j].state;
                int pid = listProc[j].pid;
                char name[16];
                int k = 0;
                while (k != 16) {
                    name[k] = procinfotable.procInfo[j].name[k];
                    k++;
                }
                listProc[j].memsize = listProc[j + 1].memsize;
                listProc[j].pid = listProc[j + 1].pid;
                listProc[j].state = listProc[j + 1].state;
                k = 0;
                while (k != 16) {
                    listProc[j].name[k] = listProc[j + 1].name[k];
                    k++;
                }
                listProc[j + 1].memsize = size;
                listProc[j + 1].pid = pid;
                listProc[j + 1].state = state;
                k = 0;
                while (k != 16) {
                    listProc[j + 1].name[k] = name[k];
                    k++;
                }
            }
        }
```

```
    }

    // changing the top procinfotable.procInfo list with the sorted
listProc processes

    for (int j = 0; j < pcount; ++j) {
        procinfotable.procInfo[j].state = listProc[j].state;
        procinfotable.procInfo[j].pid = listProc[j].pid;
        procinfotable.procInfo[j].memsize = listProc[j].memsize;
        int k = 0;
        while (k != 16) {
            procinfotable.procInfo[j].name[k] = listProc[j].name[k];
            k++;
        }
    }

    cprintf("--------------- In user program (sorted in system call by size)
---------------\n");

    // returning the list of proc_info to user programs that call this
system call
    return procinfotable.procInfo;

}
```

If you have completed all above procedure, you have successfully added new system call to xv6. However, in order to test functionality of this, you would need to add user program which calls this system call.

The user program could be as follows:

add a new file called whatever you want. I named it ps.c .

in ps.c do this:

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

// from proc_info.h proc_info structure
// per-process state // proc_info.state contains states as integers. in
xv6 we have these states for processes

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

struct proc_info {
    int pid;            // Process ID
    int memsize;          // process size
    int state;           // process state
    char name[16];        // process name
};


int
main(int argc, char *argv[]) {
    struct proc_info ptable[NPROC];
    struct proc_info *p;
    int err;

    err = proc_dump(NPROC * sizeof(struct proc_info), &ptable);
    if (err != 0)
        printf(1, "Error getting ptable");

    p = &ptable[0];

    while (p != &ptable[NPROC - 1]) {
        // I'm printing no UNUSED processes
        if (p->memsize > 0) {
            printf(1, "PID: %d    -    SIZE: %d    -    ", p->pid, p-
>memsize);
            switch (p->state) {
```

```
            case UNUSED:
               printf(1, "STATE: %s ", "UNUSED");
               break;
            case EMBRYO:
               printf(1, "STATE: %s ", "EMBRYO");
               break;
            case SLEEPING:
               printf(1, "STATE: %s ", "SLEEPING");
               break;
            case RUNNABLE:
               printf(1, "STATE: %s ", "RUNNABLE");
               break;
            case RUNNING:
               printf(1, "STATE: %s ", "RUNNING");
               break;
            case ZOMBIE:
               printf(1, " %s ", "ZOMBIE");
               break;
         }
         printf(1, "   -    NAME: %s \n", p->name);
      }
      p++;
   }

   exit();
}
```

In order to add this user program to xv6, you need to :

Edit the Makefile:

under UPROGS=\ **add this line**

_ps

then under EXTRA=\ add this :

ps.c

Now, our Makefile and our user program is ready to be tested. Enter the following commands to compile the whole system

```
make clean
make qemu

// we can type
ps
```

you can also see ps command after typing ls