Project 1B
COM S 352
Spring 2023

1. Introduction

For this project iteration, you will practice adding new system calls to the xv6 operating system.

Task	Points
1. Add the system call getppid. `int getppid(void);` The system call returns the ID of the calling process’ parent process.	10
2. Add the system call getcpids. `int getcpids(int cpids*);` The cpids is a pointer to an array of 64 integers. (Note: there can be at most 64 processes at a time in xv6.) The system call returns the number of child processes of the calling process. Let’s denote the return as n. If n>0, the first n elements of the array pointed to by cpids should contain the IDs of the n child processes of the calling process.	18
3. Add the system call getswapcount. `int getswapcount(void);` For each process in the xv6 system, keep track how many times it has been swapped off a CPU. The system call returns the number of times that the calling process has been swapped off a CPU since its creation.	18
4. Documentation (see submission instructions below)	4

2. Implementation Guide

2.1. Adding System Calls on the Kernel Side

To add new systems calls on the kernel side, modify the following files.

`kernel/syscall.h`

Each system call has a unique number that identifies it. This is used by the user side application to indicate the desired system call. Add the following system call numbers.

```
#define SYS_getppid 22
#define SYS_getcpids 23
#define SYS_getswapcount 24
```

The system call numbers are used as an index into the `syscalls[]` array of function pointers, which we will modify next.

`kernel/syscall.c`

Follow the example of the other systems calls to add the system call declarations and function pointers in `syscalls[]`.

```
...
extern uint64 sys_getppid(void);
extern uint64 sys_getcpids(void);
extern uint64 sys_getswapcount(void);

...
[SYS_getppid] sys_getppid,
[SYS_getcpids] sys_getcpids,
[SYS_getswapcount] sys_getswapcount,
```

`kernel/proc.h`

The Process Control Blocks (PCBs) in xv6 are stored in an array called `proc` (we will refer to this as the process table) which is declared in `kernel/proc.c`. The `struct proc` is defined in `kernel/proc.h`. We want each process to have a swapcount value to keep track of the number of times that the process is swapped off a CPU, so add an `int swapcount` to the `struct proc`.

`kernel/proc.c`

Most of the code will go into `kernel/proc.c, including`:

1. The default swapcount value of a process should be 0. Find the function `freeproc()`. Notice that this function zeros out many fields of proc structure so it can be reused. Follow the example and set the swapcount value of the process to 0.

2. Every time when a process is swapped off a CPU, the function sched() is called to trigger the context switch. Thus, the swapcount of the process that is swapped off should be incremented in this function. Note that, the process returned from myproc() in the following code of sched(void) is the process to be swapped off. Also note that, the increment of swapcount should be done before swtch(…) is called, because the execution of any code after the calling of swtch(…) will happen only when the process is swapped into the CPU again.

```
void
sched(void)
{
      int intena;
      struct proc *p = myproc();

      if(!holding(&p->lock))
            panic("sched p->lock");
      if(mycpu()->noff != 1)
            panic("sched locks");
      if(p->state == RUNNING)
            panic("sched running");
      if(intr_get())
            panic("sched interruptible");

      intena = mycpu()->intena;
      swtch(&p->context, &mycpu()->context);
      mycpu()->intena = intena;
}
```

3. Define the following functions to implement the required new system calls, based on the hints in the TODO comments.

```
uint64
sys_getppid(void)
{

      /*TODO:
      (1) call myproc() to get the caller's struct proc
      (2) follow the field "parent" in the struct proc to
      find the parent's struct proc
      (3) in the parent's struct proc, find the pid and return it
      */

}

uint64
sys_getcpids(void) {

      int child_pids[64];
      int number=0;

      //get the caller's struct proc
      struct proc *p = myproc();
```

```
        /*TODO:
        (1) find the caller's pid from its struct proc;
        (2) loop through the array proc (which is defined
        in proc.c as well) to find all the processes whose parent
        is the caller (i.e., whose parent's pid equals to
        the caller's pid), count the number of these processes
        at variable number, and record their pids
        to array child_pids.
        */


        //get the argument (i.e., address of an array)
        //passed by the caller
        uint64 user_array;
        argaddr(0, &user_array);

        //copy array child_pids from kernel memory
        //to user memory with address user_array
        if (copyout(p->pagetable, user_array, (char *)child_pids,
               number*sizeof(int)) < 0)
         return -1;

        // TODO: return the number of child processes found.
}


int
sys_getswapcount(void) {

        // TODO:
        // (1) call myproc() to get the caller's struct proc
        // (2) return the value of field "swapcount" in struct proc

}
```

2.2. Adding System Calls on the User Side

For a user application to call a system call the call must be declared as a function. A Perl script takes care of generating the assembly code. Update the following two user side files.

`user/usys.pl`

Add the new system call to the Perl script that automatically generates the required assembly code. Follow the example of `uptime` to add entries for `getppid`, `getcpids` and get`swapcount`.

`user/user.h`

Add the following system call declarations.

```
int getppid(void);
int getcpids(int *cpids);
int getswapcount(void);
```

2.3. Testing the System Calls (Not required, but recommended)

In the user directory, modify an existing program file or create your own. An example of testing system call getppid is the following.

```
…
int main(int argc, char *argv[])
{
     printf("The pid of parent is %d\n", getpid());
     if(fork()==0){
          printf("This is child. The pid of my parent is %d\n",
                getppid());
     exit(0);
     }
     //note: the two pids printed above should be the same!
     wait(0);
}
…
```

3. Documentation

Documentation is required for the project. Every location that you add/modify code in the kernel must have a comment explaining the purpose of the change.

Include a `README` file with your name(s), a brief description, and a list of all files added or modified in the project.

4. Submission

Make sure the code compiles, focus on working code over having every feature complete. We will look at the code for partial credit. Document anything that is incomplete in the `README` file.

Submit a zip file of the xv6-riscv directory. On the Linux command line, the zip file can be created using:

```
$ zip -r project-1b-xv6-riscv.zip xv6-riscv
```

Submit `project-1b-xv6-riscv.zip`.