Project 1B
COM S 352
Spring 2024

1. Introduction

For this project iteration, you will practice adding new system calls to the xv6 operating system, as well as read and modify some code in the kernel of the operating system.

Task	Points
1. Add system call **getppid**. `int getppid(void);` The system call returns the ID of the calling process' parent process.	10

2. Add system call **ps**.
 	
`int ps(char *psinfo);`
 	
Here, psifo is a pointer to an array of struct ps_info defined below.
 	
`struct ps_struct{`
`  int pid; //pid of a process`
`  int ppid; //parent id of the process`
`  char state[10]; //state of the process`
`  char name[16]; //name of the process`
`};`
 	
The system call returns the number of processes currently in the system, and in the array pointed to by psinfo returns the information of each of these processes in the structs of the above ps_struct.
	18

3. Add the system call **getschedhistory**. `int getschedhistory(char *history);` Here, history is a pointer to an array of struct sched_history defined below.	18

struct sched_history{
int runCount; //number of times that the process has been scheduled to run on CPU
int systemcallCount; //no. Of times that the process has made system calls
int interruptCount; //no. Of interrupts that have been made when this process runs
int preemptCount; //no. Of times that the process is pre-empted
int trapCount; //no. Of times that the process has trapped from the user mode to the kernel mode
int sleepCount; //no. Of times that the process voluntarily given up CPU
};
 	
The system call returns the pid of the current process, and in the struct_history pointed to by history the counts of the process.	

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_ps 23
#define SYS_getschedhistory 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_ps(void);
extern uint64 sys_getschedhistory(void);

...
[SYS_getppid] sys_getppid,
[SYS_ps] sys_ps,
[SYS_getschedhistory] sys_getschedhistory,
```

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 defined in `kernel/proc.h`. The struct contains the information of a process that you need to implement system calls getppid and ps.

To implement system call getschedhistory, you need more information and so **we suggest that you add the following fields to the struct of proc:**

```
int runCount; //number of times that the process has been
scheduled to run on CPU
```

```
int systemcallCount; //no. Of times that the process has
made system calls

int interruptCount; //no. Of interrupts that have been made
when this process runs

int preemptCount; //no. Of times that the process is pre-
empted

int trapCount; //no. Of times that the process has trapped
from the user mode to the kernel mode

int sleepCount; //no. Of times that the process voluntarily
given up CPU
```

Each proc struct has a field named state, of which the values are declared in enum procstate.

kernel/proc.c

Several code snippets in file proc.c deserve careful reading as you need to understand and sometimes modify them to implement the new system calls.

1. Definition of array proc[NPROC]

   In the beginning part of the file, a global array proc[NPROC] is declared. This array contains NPROC (which is 64 by default) number of proc structs. The array keeps the information of the processes in the system.

2. Function myproc(void)

   This is the function that returns a pointer pointing to the proc struct of the current process. It is convenient to use it to retrieve the current process.

3. Function freeproc(struct proc *p)

   The proc structs in array proc[NPROC] should be reusable after the processes that use them have terminated. This function is to make the space usable by cleaning the values of all the fields in a proc structure. **When you add new fields to proc struct, e.g., runConut, do remember to set the values of these fields to 0 in this function**.

4. Function scheduler(void)

   This function implements the xv6 scheduler. The default scheduling algorithm is Round-Robin (RR). That is, it checks the proc structs in array proc[NPROC] and picks one process that is in state RUNNABLE to run next.

Note that, you need to **increment runCount after such a process is picked and before the context switch (I.e., invocation of swtch(…)) happens** in this function.

5. Function sleep(void *chan, struct spinlock *lk)

   This function is called when the currently-running process cannot proceed as waiting for certain event to complete. In this case, the current process voluntarily gives up the CPU. So, you need to **increment the current process' sleepCount before sched() is invoked**, in this function.

6. Function procdump(void)

   This function prints the information of the current process. Reading its code is helpful to your implementation of system call ps.

kernel/trap.c

1. Function usertrap(void)

   This function is executed when the currently running process traps from user mode to kernel mode. **It is the place where you add the code to increment the trapCount for the process.**

   In addition, the function contains branches to deal with system calls, interrupts, and particularly the handling of timer interrupts (i.e., if which_dev == 2). **These branches are the places where you can add code to increment systemcallCount, interruptCount, and preemptCount, respectively.** Note that, when a timer interrupts the currently-running process, the process is preempted.

2. Function kerneltrap()

   This function is executed when the kernel space running on behalf of the current process is interrupted. In particular, the execution could be interrupted by timer and preempted. In the following code snippet, you need to **add also the code to increment the preemptCount**.

```
// give up the CPU if this is a timer interrupt.
if(which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING){
        //to do: add code to increment preemptCount
        yield();
}
```

kernel/sysproc.c

The functions that implement the system calls are normally placed in this file. So, at the end of the file, append the following functions.

1. The function that implements system call getppid

```
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
        */

}
```

2. The function that implements system call ps

```
extern struct proc proc[NPROC];  //declare array proc which is defined in proc.c already
uint64
sys_ps(void){

    //define the ps_struct for each process and ps[NPROC] for all processes
    struct ps_struct{
        int pid;
        int ppid;
        char state[10];
        char name[16];
    } ps[NPROC];

    int numProc = 0; //variable keeping track of the number of processes in the system

    /*To do: From array proc, find the processes that are still in the system (i.e.,
    their states are not NUNUSED. For each of the process, retrieve the
information
    and put into a ps_struct defined above*/


    //save the address of the user space argment to arg_addr
    uint64 arg_addr;
    argaddr(0, &arg_addr);

    //copy array ps to the saved address
    if (copyout(myproc()->pagetable,
        arg_addr,
```

```
                (char *)ps,
                numProc*sizeof(struct ps_struct)) < 0)
                return -1;


        //return numProc as well
        return numProc;
}
```

3. The function that implements system call getschedhistory

```
uint64
sys_getschedhistory(void){

 //define the struct for reporting scheduling history
 struct sched_history{
   int runCount;
   int systemcallCount;
   int interruptCount;
   int preemptCount;
   int trapCount;
   int sleepCount;
 } my_history;

 //To do: retrieve the current process' information to fill my_history

 //save the address of the user space argment to arg_addr
 uint64 arg_addr;
 argaddr(0, &arg_addr);

 //To do: copy the content in my_history to the saved address
 if (copyout(p->pagetable,
      arg_addr,
      (char *)&my_history,
      sizeof(struct sched_history)) < 0)
   return -1;

 //To do: return the pid as well

}
```

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`, `ps` and getschedhistory.

user/user.h

Add the following system call declarations that specify the function interface for user programs.

```
int getppid(void);

struct ps_struct{
    int pid;
    int ppid;
    char state[10];
    char name[16];
};
int ps(char *psinfo);


struct sched_history{
    int runCount;
    int systemcallCount;
    int interruptCount;
    int preemptCount;
    int trapCount;
    int sleepCount;
};
int getschedhistory(char *history);
```

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

The attached update file broadcast.c contains testing sample. Compared to the original broadcast.c in Project 1.A, this version adds faked computational operations to all processes, and the parent process makes the new system calls as follows:

```
printf("\nCall system calls for Project 1.B\n\n");

printf("Result from calling getppid:\n");
int ppid = getppid();
printf("My ppid = %d\n", ppid);
```

```
printf("\nResult from calling ps:\n");
int ret;
struct ps_struct myPS[64];
ret = ps((char *)&myPS);
printf("Total number of processes: %d\n", ret);
for(int i=0; i<ret; i++){
   printf("pid: %d, ppid: %d, state: %s, name: %s\n",
   myPS[i].pid, myPS[i].ppid, myPS[i].state, myPS[i].name);
}

printf("\nResult from calling getschedhistory:\n");
struct sched_history myHistory;
ret = getschedhistory((char *)&myHistory);
printf("My scheduling history\n pid: %d\n runs: %d, traps: %d, interrupts: %d, preemptions:
      %d, sleeps: %d, system calls: %d\n",
   ret, myHistory.runCount, myHistory.trapCount, myHistory.interruptCount,
   myHistory.preemptCount, myHistory.sleepCount, myHistory.systemcallCount);
```

A sample input/output is as follows:

```
$ broadcast 5 hello
Parent: creates child process with id: 0
Child 0: start!
Parent: creates child process with id: 1
Child 1: start!
Parent: creates child process with id: 2
Child 2: start!
Parent: creates child process with id: 3
Child 3: start!
Parent: creates child process with id: 4
Child 4: start!
Parent broadcasts: hello
Child 0: get msg (hello)
Child 1: get msg (hello)
Child 2: get msg (hello)
Child 3: get msg (hello)
Child 4: get msg (hello)
Parent receives: completed!
```

Call system calls for Project 1.B

Result from calling getppid:
My ppid = 2

Result from calling ps:
Total number of processes: 8
pid: 1, ppid: 0, state: SLEEPING, name: init
pid: 2, ppid: 1, state: SLEEPING, name: sh
pid: 15, ppid: 2, state: RUNNING, name: broadcast
pid: 16, ppid: 15, state: ZOMBIE, name: broadcast
pid: 17, ppid: 15, state: ZOMBIE, name: broadcast
pid: 18, ppid: 15, state: ZOMBIE, name: broadcast
pid: 19, ppid: 15, state: ZOMBIE, name: broadcast
pid: 20, ppid: 15, state: ZOMBIE, name: broadcast

Result from calling getschedhistory:
My scheduling history
pid: 15
runs: 27, traps: 853, interrupts: 19, preemptions: 20, sleeps: 6, system calls: 834

Note: your outputs may be different. However, the following should hold:
runs = preemptions + sleeps + 1
traps = interrupts + system calls

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`.