CS 450 PA2:
Yuyang Luo A20372496

Part (1) trace system call read:

User Level:

- User program like cat or other call function **int read(int, void*, int)**; defined in file user.h:10 to read N(the 3rd parameter) chars to the target char array(the 2nd parameter) from target file direction(the 1st parameter), the **fd** always given by sys-call named open and check in user process, if **fd** is illegal (like the file doesn't exist or cannot open), program will print an error information and stop.
- After **int read(int, void*, int);** called by sys-time called process jump into file *usys.S:15*, run **SYSCALL(read)** and execute assembly code, put parameters in stack and move to kernel mode.

System level:

- System execute void **syscall(void)** defined in *syscall.c:136*. In **syscall(void)**, **myproc(void)** defined in *proc.c:58* first be called to give the proc information, then process save the syscall ID in int num and check it, if ID was illegal, an error will be print and return to user level, else use the function pointer array and syscall id to execute syscall.
- For read syscall, system jump to **sys_read(void)** defined in *sysfile.c:70*, and use **argfd(int,int*,file**)**, **argint(int,int*)**and **argptr(int,char**,int)** to get the saved parameters:
  - In **argint(int n,int* ip**) defined in *syscall.c:50* ,first convert the offset (1st parameter) to address, then call **int fetchint(uint addr, int *ip**) to put target int value in to the memory point by the ***ip**, if success, return 0, else return -1.
    - In **int fetchint(uint addr, int *ip);** first use **myproc()** to get proc info(PCB) then check if the target memory in the proc memory range, if in, put the target **int** value into the memory point by ***ip** and **return 0**, else **return -1** for error.
  - In static **int argfd(int n, int *pfd, struct file **pf);** defined in *sysfile.c:22*, first ues **argint(int,int*)** to get the fd value, then check fd is illegal and fd is defined, if check passed, save the file struct variable to ****pf**, and **return 0**, else **return -1** for error.
  - In **int argptr(int n, char **pp, int size)** defined in *syscall.c:60*,first ues **argint(int,int*)** to get the point as int value, then check the memory block size is illegal, if check pass, save the pointer to pp and **return 0**, else **return -1** for error.
  - If any check failed (include the fd given to **read()** is undefined), return -1 for error, else jump to **int fileread(struct file *f, char *addr, int n)** defined in *file.c:97*
    - In **int fileread(struct file *f, char *addr, int n)** first check the given file is readable, is cannot read return -1;
    - Then check the type of given file f, if f is pipe, then use **piperead()** to read source.
      - In **int piperead(struct pipe *p, char *addr, int n)** defined in *pipe.c:101*, first use **void acquire(struct spinlock *lk)** defined in *spinlock.c:25* to lock the source(not trace in)
      - Then check is pipe empty, if is empty, return -1 for error, else read chars from data in pipe, write it into memory that *addr points to and return how many chars have been read. In both cases, before return, void **release(struct spinlock *lk)** defined in *spinlock.c:47* will be called to release the lock.
    - If the file type is INODE, first call **void ilock(struct inode *ip)** defined in *fs.c:301* to lock this node, then use **readi()** to read source.
      - In i**nt readi(struct inode *ip, char *dst, uint off, uint n)** defined in *fs.c:454*, first check is read from device, if read from device, call the member function **int (*read)(struct inode*, char*, int)** of **struct devsw** from the target to read data and return the chars length that read if the device number is legal, or return -1 for use an illegal device number.

        - Else check is the size to read large to the file size, **return -1** if this check filed or use **bread()** and **bmap()** to map the target data in cache (I think) and move the data from cache to target memory space of *** dst** until the n chars has been read. Finally return the size of chars have been read.
    - **fileread()** return the read length from **readi()** or **piperead()**
- **s**ys_read**()** return the read length from **fileread()**; syscall completed and return to user mode.

Part (2) implement of system call procState() on XV6

File modify:

```
--- defs.h  2017-09-27 11:52:06.357718900 -0500
+++ defs_.h 2017-08-23 13:40:36.000000000 -0500
@@ -120,7 +120,6 @@
 int             wait(void);
 void            wakeup(void*);
 void            yield(void);
-int             procState(void);

 // swtch.S
 void            swtch(struct context**, struct context*);
```

```
--- proc.c  2017-10-03 20:39:26.165950200 -0500
+++ proc_.c 2017-08-23 13:40:36.000000000 -0500
@@ -532,31 +532,3 @@
     cprintf("\n");
   }
 }
-char* get_state(enum procstate statestate)
-{
-  switch(statestate)
-  {
-    case UNUSED:    return "unused";
-    case EMBRYO:    return "sleep ";
-    case ZOMBIE:    return "zombie";
-    case RUNNING:   return "run   ";
-    case SLEEPING:  return "sleep ";
-    case RUNNABLE:  return "runble";
-    default:       return "unknow";
-  }
-}
-int procState(void)
-{
-  acquire(&ptable.lock);
-  struct proc * p;
-  cprintf("Name\tState\tPid\tMemory\n");
-
-  for(p=ptable.proc;p<&ptable.proc[NPROC];p++)
-  {
-    if(p->state==UNUSED)
-      continue;
```

```
-    cprintf("%s\t|%s\t|%d\t|%d
KB\n",p->name,get_state(p->state),p->pid,(int)(p->sz/1024));
-  }
-  release(&ptable.lock);
-  return 0;
-}
```

```
--- syscall.c   2017-10-03 20:42:03.930533500 -0500
+++ syscall_.c  2017-08-23 13:40:36.000000000 -0500
@@ -103,7 +103,6 @@
 extern int sys_wait(void);
 extern int sys_write(void);
 extern int sys_uptime(void);
-extern int sys_ps(void);

 static int (*syscalls[])(void) = {
 [SYS_fork]    sys_fork,
@@ -127,7 +126,6 @@
 [SYS_link]    sys_link,
 [SYS_mkdir]   sys_mkdir,
 [SYS_close]   sys_close,
-[SYS_ps]      sys_ps,
 };

 void
```

```
--- syscall.h   2017-09-27 11:34:32.781784700 -0500
+++ syscall_.h  2017-08-23 13:40:36.000000000 -0500
@@ -20,4 +20,3 @@
 #define SYS_link   19
 #define SYS_mkdir  20
 #define SYS_close  21
-#define SYS_ps     22
```

```
--- sysproc.c 2017-09-27 11:37:32.494974800 -0500
+++ sysproc_.c  2017-08-23 13:40:36.000000000 -0500
@@ -89,8 +89,3 @@
   release(&tickslock);
   return xticks;
 }
-
-int sys_ps(void)
-{
-  return procState();
-}
```

```
--- usys.S  2017-09-27 11:42:03.864580700 -0500
+++ usys_.S 2017-08-23 13:40:36.000000000 -0500
@@ -29,4 +29,3 @@
 SYSCALL(sbrk)
 SYSCALL(sleep)
 SYSCALL(uptime)
-SYSCALL(ps)
\ No newline at end of file
```

New file:

```
//ps.c
#include "types.h"
#include "stat.h"
#include "user.h"

int ps();

int main(int argc, char *argv[])
{
    ps();
    exit();
}
```

Description:

The function of **int procState(void)** is defined in file *proc.c,* where the variable **ptable** also defined in for access **ptable**, where save all process information of xv6.

When user use ps command to execute the ps file, finction **int ps()**; defined in ps.c then system will jump to usys.S, the line **SYSCALL(ps**) defined the function **ps()** is a system call, then system get into system level and do the same thing like **read()** does, use function **syscall(void)** to call **int sys_ps(void)**, which defined in *sysproc.c*, and in this function, **procState()** finally be called.

Exception handling

ps command doesn't need any parameter, so don't need function like **argint()** to get parameter, and when procState() execute, use **acquire(&ptable.lock);** and **release(&ptable.lock);** to make sure there will be no resource conflict when print processes' information.

Test Case:

Direct run ps

```
$ ps
Name    State   Pid     Memory
init    |sleep  |1      |12 KB
sh      |sleep  |2      |16 KB
ps      |run    |3      |12 KB
```

Run ps in background

```
$ ps&
$ Name  State   Pid     Memory
init    |sleep  |1      |12 KB
sh      |run    |2      |16 KB
ps      |run    |6      |12 KB
```

Run ps in a new sh

```
$ sh
$ ps
Name    State   Pid     Memory
init    |sleep  |1      |12 KB
sh      |sleep  |2      |16 KB
sh      |sleep  |12     |16 KB
ps      |run    |13     |12 KB
```

Run ps after a background task:

```
$ test 10 .&
$ ps
Name    State   Pid     Memory
init    |sleep  |1      |12 KB
sh      |sleep  |19     |16 KB
ps      |run    |31     |12 KB
test    |runble |30     |12 KB
```

In all test case, ps result just same as wish.