Homework #2 Xv6

Course

CSE 460 Operating Systems

Instructor

Dr. Yan Zhang

Meeting Time

Mon. & Wed., 4:00 p.m. - 5:15 p.m.

Due Date

March 11, 2020

Authors

Kevin T. Vo

Esdras Lopez

Joseph Gonzales

Trevor Shortlidge

Brian Ayala

Group Members’ Information

Name	**Coyote ID**	**Responsibility**
Kevin T. Vo	006316930	● Implemented date system call & command (Part a) into xv6 ● Implement part of uid, gid, ppid (Part c) ● Contributed to this report/documentation
Esdras Lopez	006198864	● Implemented part of uid, gid, ppid (Part c) ● Contributed to this report/documentation
Joseph Gonzales	006242648	● Implemented ps command and Ctrl+P (Part d) ● Contributed to this report/documentation
Trevor Shortlidge	006310209	● Implemented part of Ctrl+P (Part b) ● Reformatted output of ps command (Part d) ● Contributed to this report/documentation
Brian Ayala	006191688	● Implemented part of Ctrl+P (Part b) ● Contributed to this report/documentation

Table of Contents

I. Demonstration of the date() System Call ……………………………………………4

II. Demonstration of the Ctrl-P Special Control Sequence…………………………….......6

III. Demonstration of UIDs, GIDs, and PPIDs……………………………………….......7

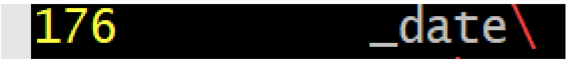
IV. Demonstration of the "ps" Command………………………………………………13

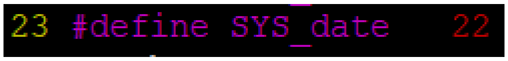
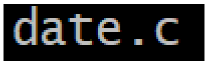
I. Demonstration of the date() System Call

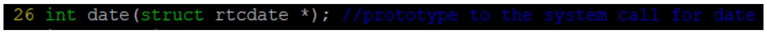
The system call “date” has been implemented to the Xv6 operating system where it would display the current day, month, year, hour, minute, and second in the format below:

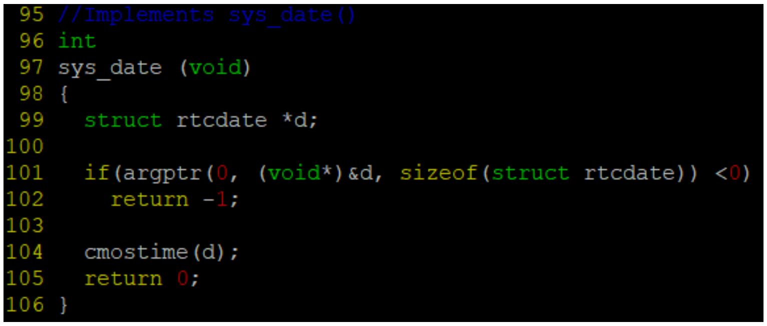
DAY/MONTH/YEAR HOUR:MINUTE:SECOND

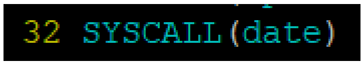
During the addition of this system call the following files have been created/modified:

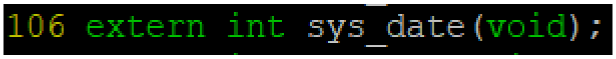
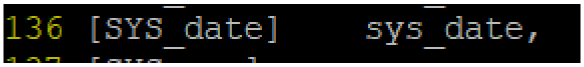
Filename	Created / Modified	Line Numbers	Code
Makefile			 `176          _date\`

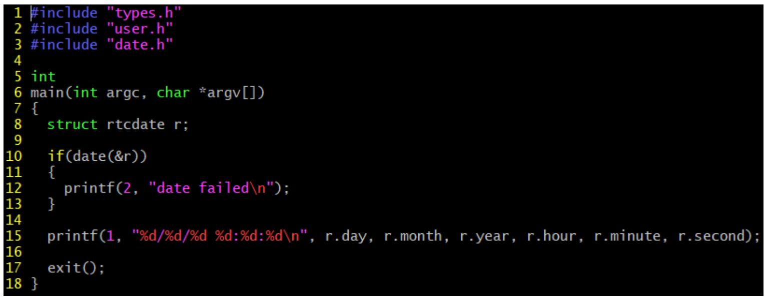
syscall.h	Modified	23
		&
		248
			
			`23 #define SYS_date   22`
			
			`date.c`

user.h	Modified	26	 `26 int date(struct rtcdate *); //prototype to the system call for date`

sysproc.c	Modified	95-106	
			` 95 //Implements sys_date()`
			` 96 int`
			` 97 sys_date (void)`
			` 98 {`
			` 99   struct rtcdate *d;`
			`100`
			`101   if(argptr(0, (void*)&d, sizeof(struct rtcdate)) <0)`
			`102     return -1;`
			`103`
			`104   cmostime(d);`
			`105   return 0;`
			`106 }`

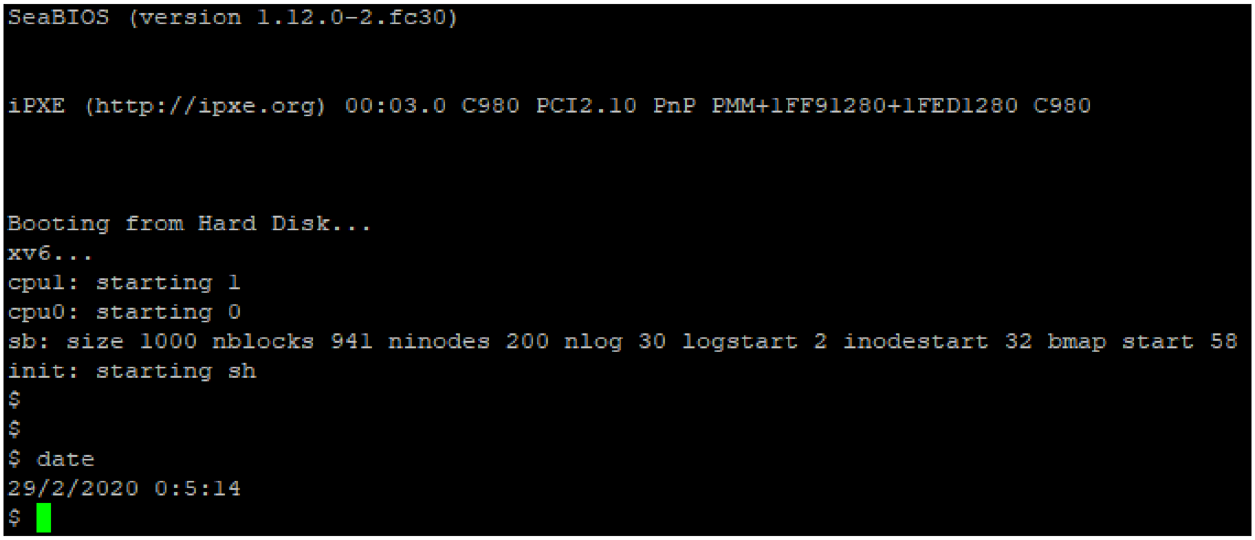
usys.S	Modified	32	 `32 SYSCALL(date)`

syscall.c	Modified	106
		&
		136
			
			`106 extern int sys_date(void);`
			
			`136 [SYS_date]    sys_date,`

date.c	Created	1-18	

```
#include "types.h"
#include "user.h"
#include "date.h"

int
main(int argc, char *argv[])
{
  struct rtcdate r;

  if(date(&r))
  {
    printf(2, "date failed\n");
  }

  printf(1, "%d/%d/%d %d:%d:%d\n", r.day, r.month, r.year, r.hour, r.minute, r.second);

  exit();
}
```

Result from Execution of date() System Call:

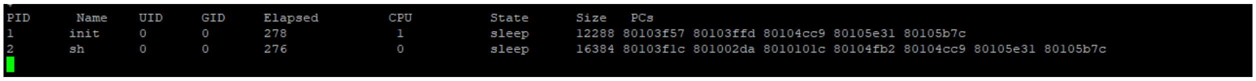


```
SeaBIOS (version 1.12.0-2.fc30)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF91280+1FED1280 C980



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
$
$
$ date
29/2/2020 0:5:14
$
```

Results from the syscall "date" comes from implementing the above into a full fledge syscall. Modified syscall.h that contains the mapping of the call number linked to SYS_date. Modified defs.h to hold the kernerl wide function call for date. Modified user.h that contains the functions prototype for date that is required to run user programs. Implemented the function for date inside of sysproc.c by getting the arguments off the stack. This implementation was provided to us. Next we modified the usys.S that exports the system call. Modified syscall.c to add the extern int for being in another file. Then the last step is to add the date.c file where we invoke the syscall implementation.

II. Demonstration of the Ctrl-P Special Control Sequence

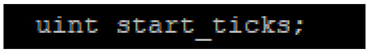
Result for Ctrl-P Special Control Sequence:



```
PID      Name    UID     GID     Elapsed         CPU           State      Size   PCs
1       init     0       0       278              1            sleep      12288 80103f57 80103ffd 80104cc9 80105e31 80105b7c
2       sh       0       0       276              0            sleep      16384 80103f1c 801002da 8010101c 80104fb2 80104cc9 80105e31 80105b7c
```

Note: UID,GID, CPU are from part D, Dr.Zhang said it was ok to leave elapsed time in milliseconds.

Proc.h line 56 -> added uint start_ticks to structure proc



```
uint start_ticks;
```

Modified procdump in proc.c to display our results in the ptable to output elapsed time, & size

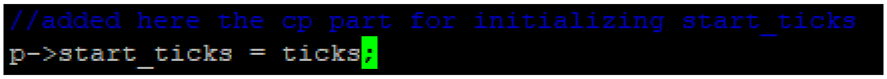
Elapsed time result is done by subtracting ticks minus start_ticks do give us the delta in milliseconds.

However, changing it to a float and dividing by 1000 to give us the result in seconds was not working with cprintf. Refer to to the comment I made before about Dr.Zhang letting our group use milliseconds instead.

Also p->sz is a built in variable in proc that deals with the size of each block of memory.

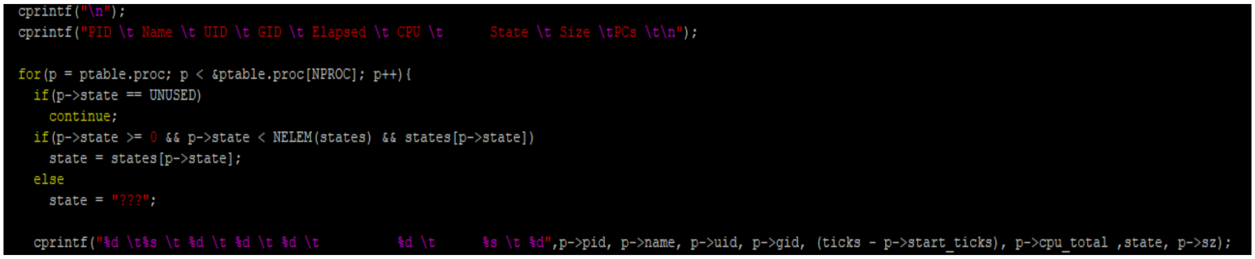
We just outputted the size : p->sz to display.

Proc.c line 131 -> initialized start_ticks to ticks that is a global counter in milliseconds.



```
//added here the cp part for initializing start_ticks
p->start_ticks = ticks;
```

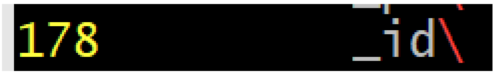
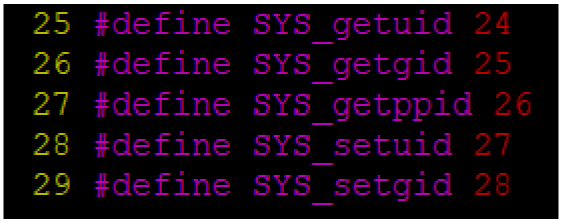
proc.c procdump() - line 540

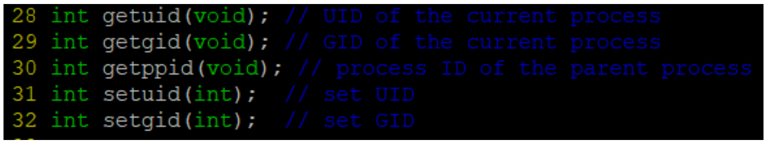


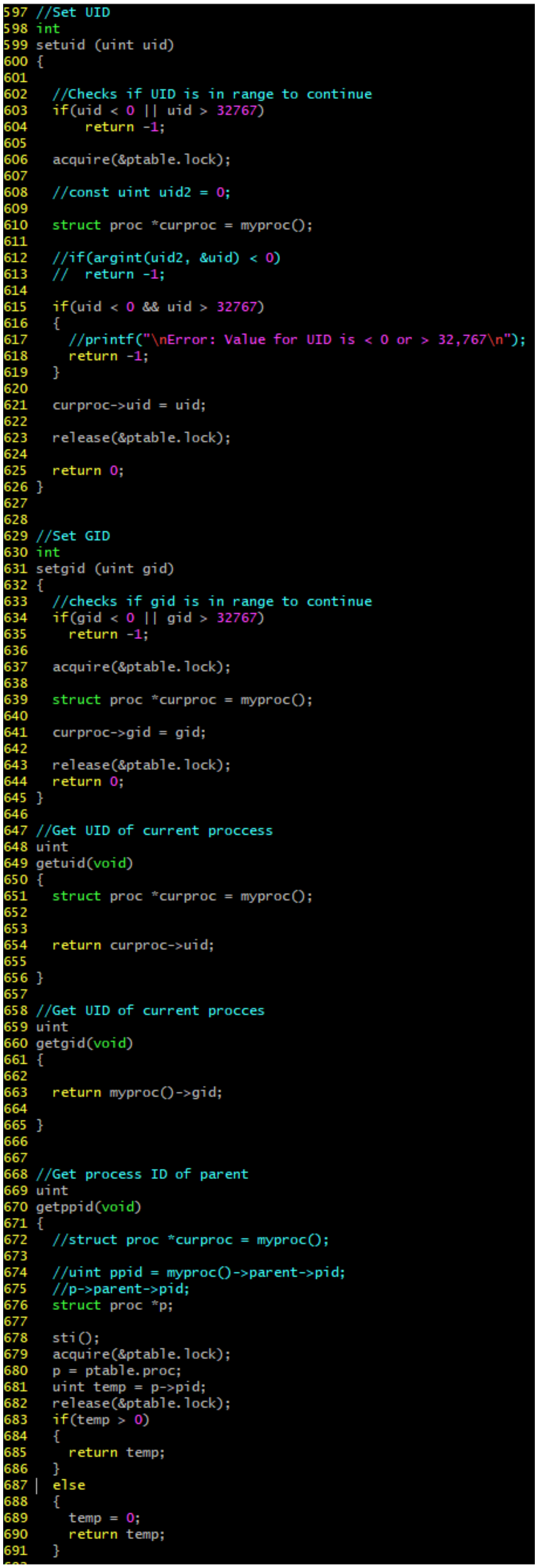
```
cprintf("\n");
cprintf("PID \t Name \t UID \t GID \t Elapsed \t CPU \t      State \t Size \tPCs \t\n");

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == UNUSED)
    continue;
  if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
    state = states[p->state];
  else
    state = "???";

  cprintf("%d \t%s \t %d \t %d \t %d \t          %d \t      %s \t %d",p->pid, p->name, p->uid, p->gid, (ticks - p->start_ticks), p->cpu_total ,state, p->sz);
```

III. Demonstration of UIDs, GIDs, and PPIDs

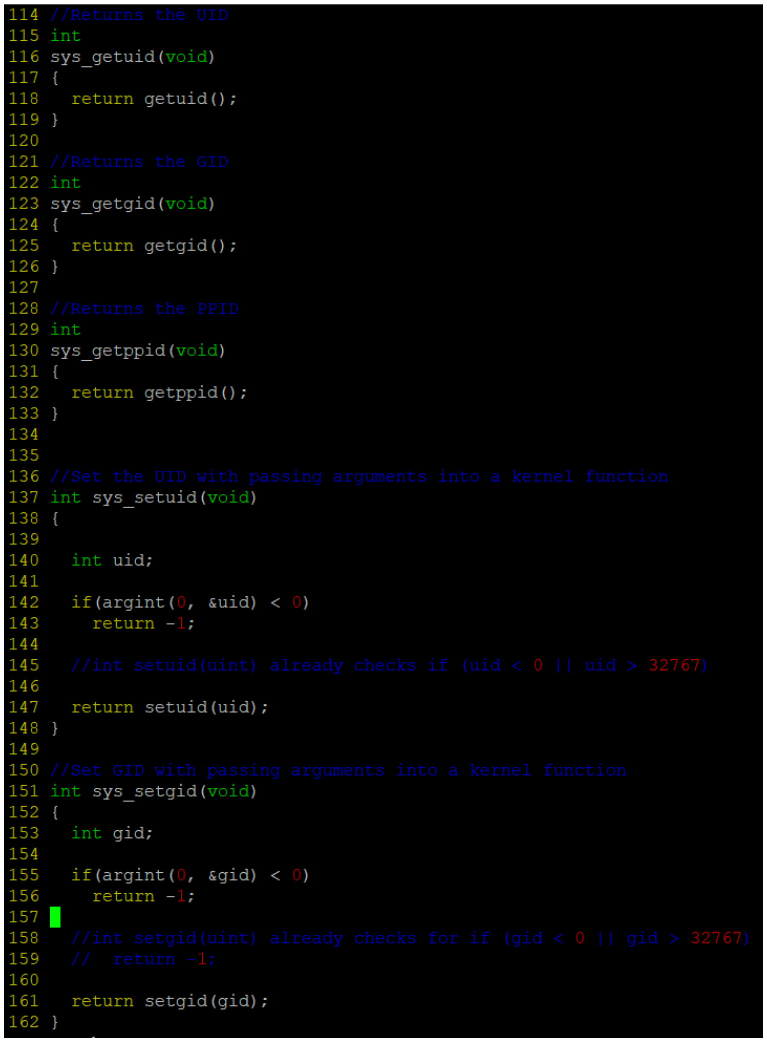
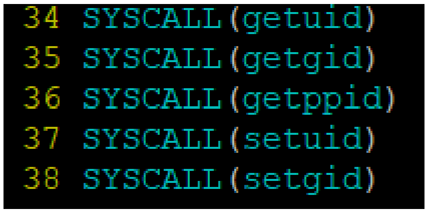
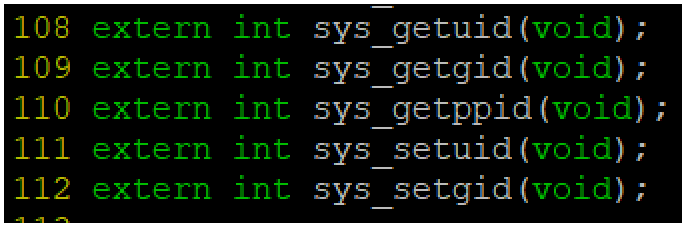
The implements the feature of users and groups into xv6 through uid and gid where it be used to store ID unsigned integers for tracking the ownership of a process for a user or group. By typing “id” into the terminal when executing xv6, the system call will invoke and demonstrate this feature by displaying the UID, GID and PPID that has been established by the test function within “id.c”.

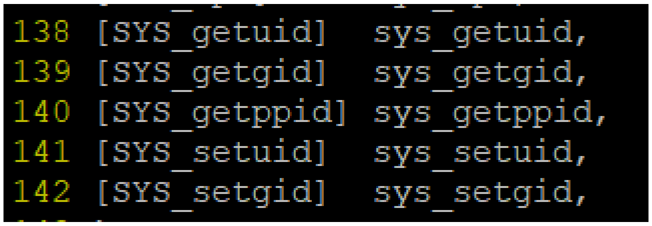
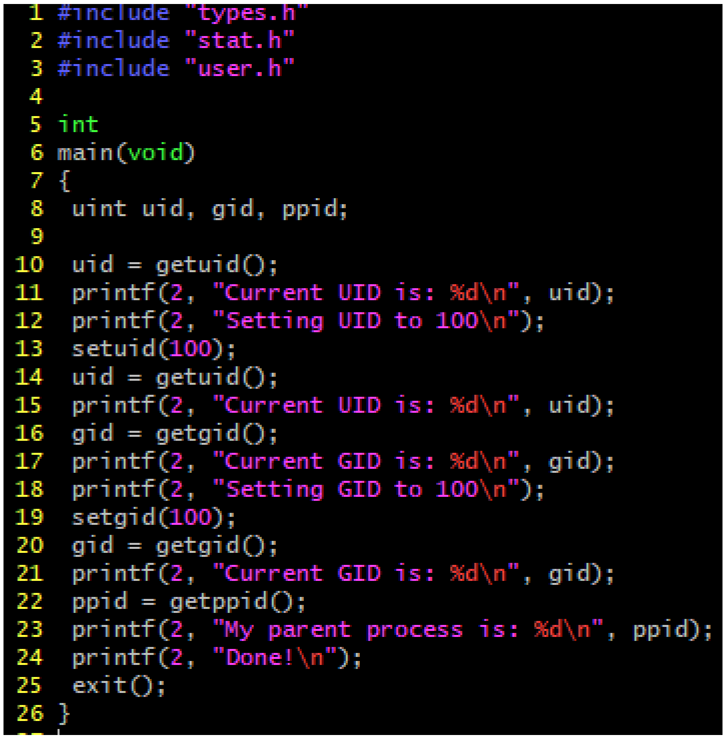
Filename	Created / Modified	Line Numbers	Image
Makefile		178 & 248	 178 _id\\ id.c
syscall.h	Modified	25-29	 25 #define SYS_getuid 24 26 #define SYS_getgid 25 27 #define SYS_getppid 26 28 #define SYS_setuid 27 29 #define SYS_setgid 28

user.h	Modified	28-32	
			28 int getuid(void); // UID of the current process
			29 int getgid(void); // GID of the current process
			30 int getppid(void); // process ID of the parent process
			31 int setuid(int); // set UID
			32 int setgid(int); // set GID

proc.c	Modified	597-691	

```
597 //Set UID
598 int
599 setuid (uint uid)
600 {
601
602   //Checks if UID is in range to continue
603   if(uid < 0 || uid > 32767)
604       return -1;
605
606   acquire(&ptable.lock);
607
608   //const uint uid2 = 0;
609
610   struct proc *curproc = myproc();
611
612   //if(argint(uid2, &uid) < 0)
613   //  return -1;
614
615   if(uid < 0 && uid > 32767)
616   {
617     //printf("\nError: Value for UID is < 0 or > 32,767\n");
618     return -1;
619   }
620
621   curproc->uid = uid;
622
623   release(&ptable.lock);
624
625   return 0;
626 }
627
628
629 //Set GID
630 int
631 setgid (uint gid)
632 {
633   //checks if gid is in range to continue
634   if(gid < 0 || gid > 32767)
635     return -1;
636
637   acquire(&ptable.lock);
638
639   struct proc *curproc = myproc();
640
641   curproc->gid = gid;
642
643   release(&ptable.lock);
644   return 0;
645 }
646
647 //Get UID of current proccess
648 uint
649 getuid(void)
650 {
651   struct proc *curproc = myproc();
652
653
654   return curproc->uid;
655
656 }
657
658 //Get UID of current procces
659 uint
660 getgid(void)
661 {
662
663   return myproc()->gid;
664
665 }
666
667
668 //Get process ID of parent
669 uint
670 getppid(void)
671 {
672   //struct proc *curproc = myproc();
673
674   //uint ppid = myproc()->parent->pid;
675   //p->parent->pid;
676   struct proc *p;
677
678   sti();
679   acquire(&ptable.lock);
680   p = ptable.proc;
681   uint temp = p->pid;
682   release(&ptable.lock);
683   if(temp > 0)
684   {
685     return temp;
686   }
687 | else
688   {
689     temp = 0;
690     return temp;
691   }
```

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td>sysproc.c</td><td>Modified</td><td>114-162</td><td>
114 //Returns the UID
115 int
116 sys_getuid(void)
117 {
118   return getuid();
119 }
120
121 //Returns the GID
122 int
123 sys_getgid(void)
124 {
125   return getgid();
126 }
127
128 //Returns the PPID
129 int
130 sys_getppid(void)
131 {
132   return getppid();
133 }
134
135
136 //Set the UID with passing arguments into a kernel function
137 int sys_setuid(void)
138 {
139
140   int uid;
141
142   if(argint(0, &uid) < 0)
143   return -1;
144
145   //int setuid(uint) already checks if (uid < 0 || uid > 32767)
146
147   return setuid(uid);
148 }
149
150 //Set GID with passing arguments into a kernel function
151 int sys_setgid(void)
152 {
153   int gid;
154
155   if(argint(0, &gid) < 0)
156   return -1;
157
158   //int setgid(uint) already checks for if (gid < 0 || gid > 32767)
159   // return -1;
160
161   return setgid(gid);
162 }</td></tr>
<tr><td>usys.S</td><td>Modified</td><td>34-38</td><td>
34 SYSCALL(getuid)
35 SYSCALL(getgid)
36 SYSCALL(getppid)
37 SYSCALL(setuid)
38 SYSCALL(setgid)</td></tr>
<tr><td>syscall.c</td><td>Modified</td><td>108-112
&
138-142</td><td>
108 extern int sys_getuid(void);
109 extern int sys_getgid(void);
110 extern int sys_getppid(void);
111 extern int sys_setuid(void);
112 extern int sys_setgid(void);</td></tr>
</table>

			
id.c	Created	1-26	

```
138 [SYS_getuid]   sys_getuid,
139 [SYS_getgid]   sys_getgid,
140 [SYS_getppid]  sys_getppid,
141 [SYS_setuid]   sys_setuid,
142 [SYS_setgid]   sys_setgid,
```

```
 1 #include "types.h"
 2 #include "stat.h"
 3 #include "user.h"
 4
 5 int
 6 main(void)
 7 {
 8  uint uid, gid, ppid;
 9
10  uid = getuid();
11  printf(2, "Current UID is: %d\n", uid);
12  printf(2, "Setting UID to 100\n");
13  setuid(100);
14  uid = getuid();
15  printf(2, "Current UID is: %d\n", uid);
16  gid = getgid();
17  printf(2, "Current GID is: %d\n", gid);
18  printf(2, "Setting GID to 100\n");
19  setgid(100);
20  gid = getgid();
21  printf(2, "Current GID is: %d\n", gid);
22  ppid = getppid();
23  printf(2, "My parent process is: %d\n", ppid);
24  printf(2, "Done!\n");
25  exit();
26 }
```

Description of changes made within the file "syscall.h":

- The system calls responsible for setting and returning the UID, GID, and returning PPID are added so that it system call name can be mapped to a system call number so that it can be invoked within xv6.

Description of changes within the file "user.h":

- The prototype to the getter and setter functions for setting and returning the unsigned integers of UID and GID including one getter function for the PPID.
- The prototypes included are to the functions that holds the code to perform the task that a user uses a system call to request for.

Description of changes within the file "proc.c":

- proc.c contains the functions:
  uint getuid (void) // UID of the current process
  uint getgid (void) // GID of the current process
  uint getppid (void) // process ID of the parent process
  int setuid (uint) // set UID
  int setgid (uint) // set GID

  to actually sets the UID and GID using the setuid(uint) and setgid(uint) functions.

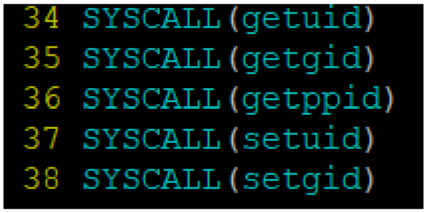
Note that user/xv6 will use these two functions to set the UID and GID. The getter functions uint getuid(void), uint getgid(void), and uint getppid(void) that returns the UID, GID, and PPID respectively. It is within setter functions where it checks if the value UID and GID are within range of [0, 32727].

Description of changes within the file "sysproc.c":

- Included the sys functions that returns the actual functions listed above to perform the process associated its system call.

Description of changes within the file "usys.S":

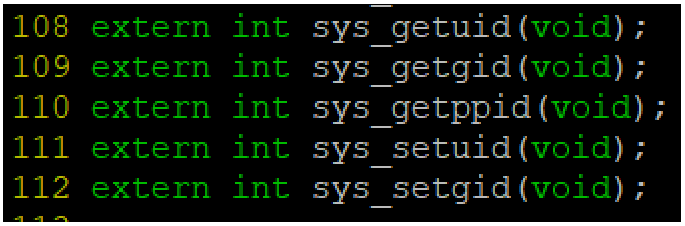
- "usys.S" contains the system calls to be made available by the kernel:



```
SYSCALL(getuid)
SYSCALL(getgid)
SYSCALL(getppid)
SYSCALL(setuid)
SYSCALL(setgid)
```

Description of changes within the file "syscall.c":

- First contains the system call entry points:



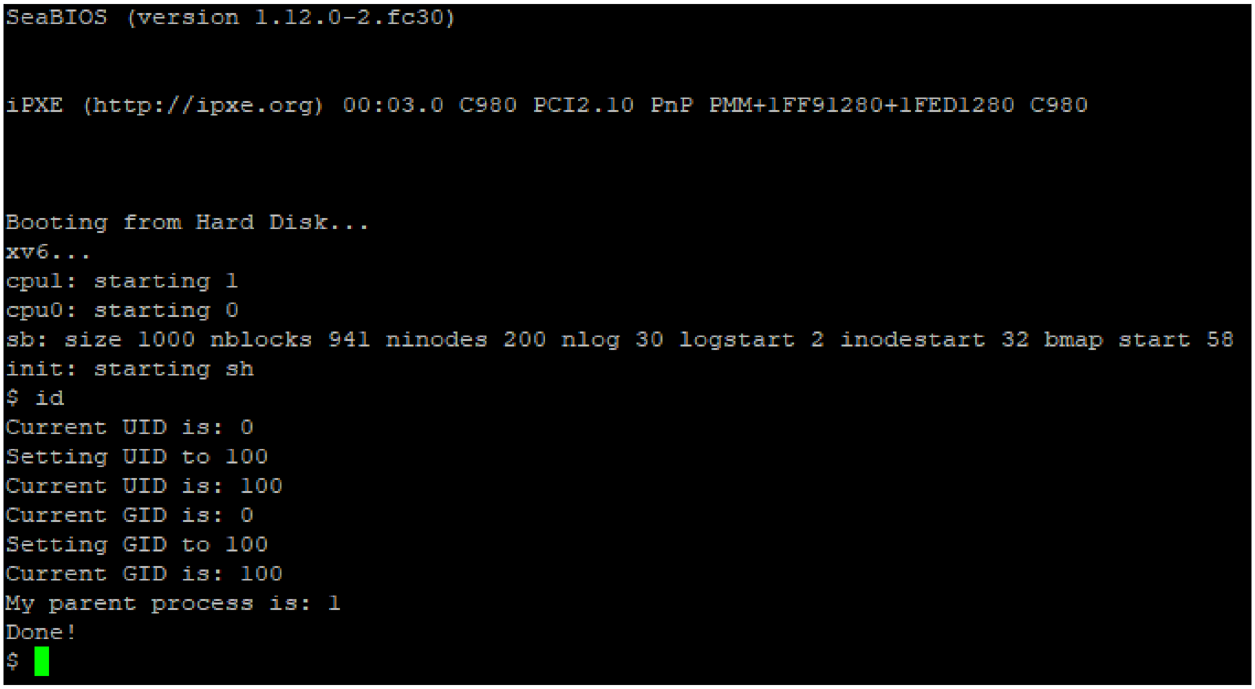
```
extern int sys_getuid(void);
extern int sys_getgid(void);
extern int sys_getppid(void);
extern int sys_setuid(void);
extern int sys_setgid(void);
```

- Secondly, "syscall.c" contains a dispatch table that defines from the symbol name made earlier within files "usys.S" and "syscall.h" to its function name. All of this is done within the "int (*syscalls[])(void)" within the "syscall.c".

Description of changes made within file “id.c”:

- Contains the test function to run the system calls implemented in this part.

Result from UIDs, GIDs, and PPIDs:



```
SeaBIOS (version 1.12.0-2.fc30)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF91280+1FED1280 C980



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
$ id
Current UID is: 0
Setting UID to 100
Current UID is: 100
Current GID is: 0
Setting GID to 100
Current GID is: 100
My parent process is: 1
Done!
$
```

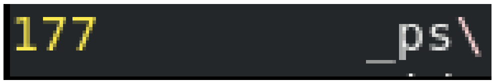
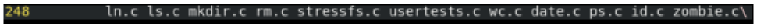
The following results derived from the test function within “id.c” for testing the uint getuid(void), uint getgid(void), uint getppid(void), int setuid(uint), and int setgid(uint) functions along with the system call to them. The setuid(uint) and setgid(uint) takes in an unsigned integer so assigns them the UID or GID for a process. The getters uint getuid(void) and uint getgid(void) returns value of the UID and GID that has been set. Lastly, uint getppid(void) returns back the parent process by first calling the sti() to interrupt the processor. Then, it acquires the p table which contains all the process. However, this function will only focus on the parent process with uint temp = p->pid; and then returns it. If the p->pid happens to be a negative value, then temp = 0.

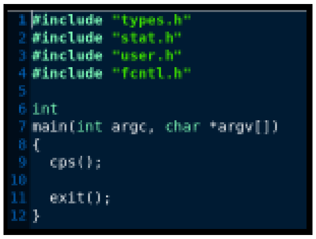
IV. Demonstration of the “ps” Command

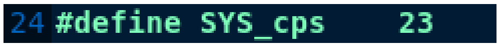
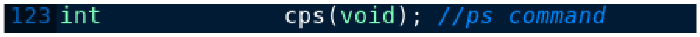
The system call “cps” has been implemented to the Xv6 operating system in order to display the ptable for the current processes, through the use of the “ps” command. It outputs the following process information: pid, name, UID, GID, PPID, ELAPSED time (in ms), CPU time (in ms), size, and state.

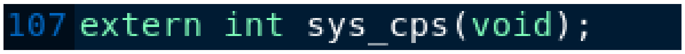
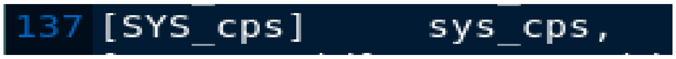
During the addition of this system call the following files have been created/modified:

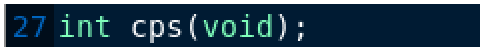
Filename	Created / Modified	Line Numbers	Image

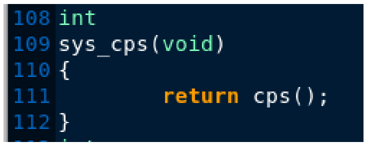
Makefile	Modified	177
		&
		248
			
			`177 _ps\`
			
			`248 ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c date.c ps.c id.c zombie.c\`

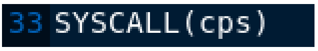
ps.c	Created	1-12	
			`1 #include "types.h"`
			`2 #include "stat.h"`
			`3 #include "user.h"`
			`4 #include "fcntl.h"`
			`5`
			`6 int`
			`7 main(int argc, char *argv[])`
			`8 {`
			`9   cps();`
			`10`
			`11   exit();`
			`12 }`

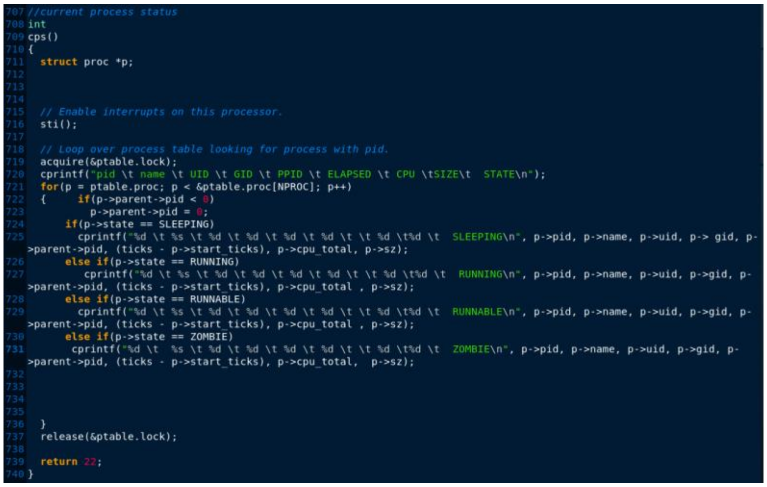
syscall.h	Modified	24	 `24 #define SYS_cps    23`
defs.h	Modified	123	 `123 int             cps(void); //ps command`

syscall.c	Modified	107
		&
		137
			
			`107 extern int sys_cps(void);`
			
			`137 [SYS_cps]    sys_cps,`

user.h	Modified	27	 `27 int cps(void);`

sysproc.c	Modified	108-112	
			`108 int`
			`109 sys_cps(void)`
			`110 {`
			`111         return cps();`
			`112 }`

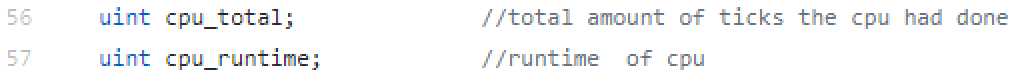
usys.S	Modified	33	 `33 SYSCALL(cps)`

proc.c	Modified	707-740	

```
707 //current process status
708 int
709 cps()
710 {
711   struct proc *p;
712
713
714
715   // Enable interrupts on this processor.
716   sti();
717
718   // Loop over process table looking for process with pid.
719   acquire(&ptable.lock);
720   cprintf("pid \t name \t UID \t GID \t PPID \t ELAPSED \t CPU \tSIZE\t  STATE\n");
721   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
722   {     if(p->parent->pid < 0)
723           p->parent->pid = 0;
724       if(p->state == SLEEPING)
725         cprintf("%d \t %s \t %d \t %d \t %d \t %d \t \t %d \t%d \t  SLEEPING\n", p->pid, p->name, p->uid, p-> gid, p->parent->pid, (ticks - p->start_ticks), p->cpu_total, p->sz);
726       else if(p->state == RUNNING)
727          cprintf("%d \t %s \t %d \t %d \t %d \t %d \t \t %d \t%d \t  RUNNING\n", p->pid, p->name, p->uid, p->gid, p->parent->pid, (ticks - p->start_ticks), p->cpu_total , p->sz);
728       else if(p->state == RUNNABLE)
729         cprintf("%d \t %s \t %d \t %d \t %d \t %d \t \t %d \t%d \t  RUNNABLE\n", p->pid, p->name, p->uid, p->gid, p->parent->pid, (ticks - p->start_ticks), p->cpu_total , p->sz);
730       else if(p->state == ZOMBIE)
731        cprintf("%d \t  %s \t %d \t %d \t %d \t %d \t \t %d \t%d \t  ZOMBIE\n", p->pid, p->name, p->uid, p->gid, p->parent->pid, (ticks - p->start_ticks), p->cpu_total,  p->sz);
732
733
734
735
736   }
737   release(&ptable.lock);
738
739   return 22;
740 }
```

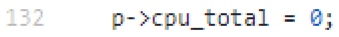
The total CPU time was completed as follows:

proc.h lines 56 & 57 -> added uint cpu_total and uint cpu_runtime to the proc structure.



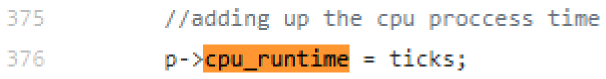
```
56      uint cpu_total;              //total amount of ticks the cpu had done
57      uint cpu_runtime;            //runtime  of cpu
```

proc.c line 132 -> cpu_total initialized to 0.



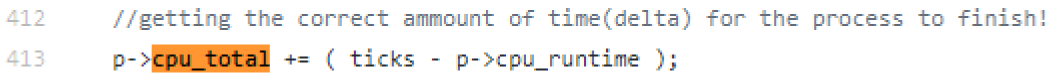
```
132      p->cpu_total = 0;
```

proc.c line 376 -> cpu_runtime set to ticks.



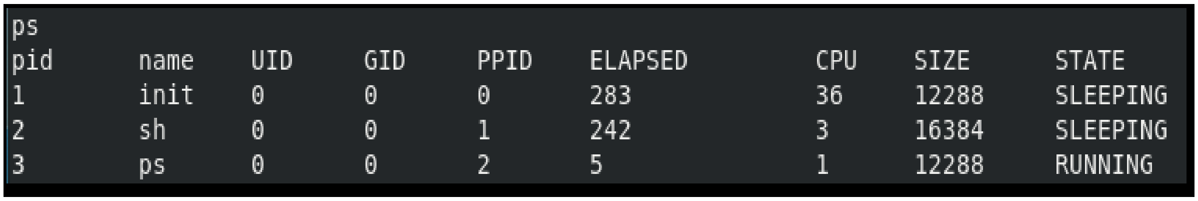
```
375          //adding up the cpu proccess time
376          p->cpu_runtime = ticks;
```

proc.c line 413 -> cpu_total calculated by adding (ticks minus the cpu_runtime) to give us the total CPU time in milliseconds.



```
412     //getting the correct ammount of time(delta) for the process to finish!
413     p->cpu_total += ( ticks - p->cpu_runtime );
```

Result from Execution of cps() System Call:



```
ps
pid      name     UID      GID      PPID     ELAPSED           CPU     SIZE       STATE
1        init     0        0        0        283               36      12288      SLEEPING
2        sh       0        0        1        242               3       16384      SLEEPING
3        ps       0        0        2        5                 1       12288      RUNNING
```

Results from the system call “cps” comes from implementing it, as well as the ps command, into Xv6. The “ps” command was created by creating the ps.c file and modifying the Makefile, in order to add the command to Xv6. Modification of syscall.h was done, mapping the call number linked to SYS_cps. Additionally, defs.h was modified to hold the function call for cps. Next, syscall.c was modified in order to include the extern int of sys_cps. A function prototype for cps was also added to user.h, used by user programs. Next, sysproc.c was modified to implement the function for cps. Exporting the system call was done by modifying usys.S to include the cps system call. Finally, the implementation for cps was added to proc.c, allowing the various process information to be output in a ptable.