Description

For this assignment, I became familiar with: locks for kernel-level data structures; concurrency; implementing new system calls to support process ownership; implement a new system call to obtain process information; implement tracking for the amount of time a process uses a CPU; implementing a new user-level command to display process state; implementing a new user-level command to time process execution; modifying the xv6 console to display additional process information; and writing a project report to properly document project work.

Deliverables

The following features were added to xv6:

- UIDs, GIDs, & PPIDS:

  At this point xv6 has no concept of user or groups. User IDs (UID) and Group IDs (GID) will be added to xv6, where we will *track process ownership*. The PPID is the "parent process identifer" or parent PID. This will be determined on-the-fly, wiht process one being shown as its own parent; consistent with UNIX/Linux.

  The kernel code cannot assume that arguments passed into the kernel are valid, and so the kernel code must check the values for the correct range (`0 <= value <= 32767`).

  The following items were added to xv6 to implement the UID, GID, & PPID features:

  – User ID (`uid`) and GID (`gid`) fields to the process structure.
  – Parent Process ID (`ppid`) of current process enabled to be determined on-the-fly.
  – Ability to set UID and GID of current process.
  – System calls to access the User ID, Group ID, and Parent Process ID of the current process; system calls to set the User ID and Group ID:

  ```
  uint getuid(void) // UID of the current process
  uint getgid(void) // GID of the current process
  uint getppid(void) // PPID of parent process
  int setuid(uint) // set UID
  int setgid(uint) // set GID
  ```

  – Test code for the new UID, GID, and PPID system calls.
  – A new system call, `testids`, for running the new test program.

- Process Execution Time:

  Currently, our xv6 system tracks when a process enters the system and displays *elapsed* time in the console command "control-P". We will now track how much CPU time a process uses. There are two situations where a context switch occurs in xv6: one to put a process into a CPU and one to take a process out of its current CPU.

  The following items were added to xv6 to implement Process Execution Time:

  – Fields for Total Elapsed Ticks, and Ticks when sheduled within a CPU:

  ```
  uint cpu_ticks_total // total elapsed ticks in CPU
  uint cpu_ticks_in  // ticks when scheduled
  ```

- The ”ps” Command:

  Xv6 does not have the `ps` command like Linux, so we will have to add our own. This command is used to find out information regarding active processes in the system. A `ps` program will be written to implement the system call. An active process will be defined in the `RUNNABLE`, `SLEEPING`, or `RUNNING` state. Under no circumstances will processes in either the `UNUSED` or `EMBRYO` states be included.

- The Time Command: User command to determine the number of seconds that a program takes to run.

Implementation

UIDs, GIDs, and PPIDs

All code for the deliverables was conditionally compiled using the `CS333_P2` flag in the `Makefile` (line 3).
The following files were modified to add the UID, GID, and PPID fields and associated system calls:

- `param.h.` (Lines 15 & 16) Two `#define` statements added, one each for `UID` and `GID`, for initializing the first process and increasing readability of the code.
- `proc.h.` (Lines 72 - 77) Five fields were added to the process structure (`struct proc`) :

  ```
  uint getuid(void) // UID of the current process
  uint getgid(void) // GID of the current process
  uint getppid(void) // PPID of parent process
  int setuid(uint) // set UIDppss
  int setgid(uint) // set GID
  ```

- `proc.c.` UID, GID, and PPID values were added to the routines where new processes are created:
  - Lines 108 - 112 In the routine `userinit()`, where the first process is created piece-by-piece at boot time, the `proc` structure's `uid` and `gid` fields are initialized to the `UID` and `GID` constands defined in `param.h`.
  - Lines 174 - 177 In the routine `fork()`, where a new process structure is allocated and the original process is copied into the new one, the new `uid` and `gid` fields in `struct proc` need to be updated as well. They are given the value of the original process.
- `sh.c.` We should be able to set the UID and GID of the currently executing shell with appropriate built-in commands. The shell includes in the parser the ability to identify built-ins, as built-ins begin with an underscore (_). These have already been implemented. I removed the phrase `NOT_YET` from the code `#ifdef USE_BUILTINS_NOT_YET` (Lines 145 & 260). The lines now read `#ifdef USE_BUILTINS`.
- `testids.c.` A simple test program was created to test the five new system calls for setting and getting the UID, GID, and PPID. A line (`_testids`) was appended in the `Makefile` (Line 21) to add a system call for the new test program.
- `usys.S.` The user-side subs for the new system calls were added (Lines 34 - 38).
- `syscall.h.` The system call numbers for `getuid()`, `getgid()`, `getppid()`, `setuid(uint)`, & `setgid(uint)` were created by appending the existing list (Lines 28 - 32).
- `syscall.c`
  - Modified to include the kernel-side function prototypes (Lines 106 - 110).
  - Added entries to the function dispatch table `syscalls[]` (Lines 141 - 145).
  - Added entries to the `syscallnames[]` array to print the system call name when the `PRINT_SYSCALLS` flag is defined (Lines 180 - 184).
- `sysproc.c.` The following functions were added to this file to implement the kernel-side system calls:
  - `sys_getuid()` (Lines 115 - 118) Returns the UID value of the current process.
  - `sys_getgid()` (Lines 121 - 124) Returns the GID value of the current process.
  - `sys_getppid()` (Lines 127 - 130) Returns the PID value of the current process' parent.

- `sys_setuid(uint)` (Lines 133 - 145) Populates an `int` value pulled from an argument on the stack, then assigns that value to the UID of the current process.
- `sys_setuid(uint)` (Lines 148 - 160) Populates an `int` value pulled from an argument on the stack, then assigns that value to the GID of the current process.

Process Execution Time

The following files were modified to track how much CPU time a process uses:

- `proc.h`. (Lines 76, 77) added fields to process structure:

  ```
  uint cpu_ticks_total; // total elapsed ticks in CPU
  uint cpu_ticks_in;  // ticks when scheduled
  ```

- `proc.c`.
  - Modified `allocproc()` to initialize the new `cpu_ticks_total & cpu_ticks_in` fields to `0` (Lines 81, 82).
  - Modified `scheduler()` (Line 333) to set the process to being scheduled's `cpu_ticks_in` field to the value of the global `ticks`, at the time of scheduling.
  - Modified `sched()` (Line 379) to set accumulate the time the process has been in the scheduler, at the time of its removal from the schduler, This is done by subtracting `cpu_ticks_in` from `ticks` and adding that to the total `cpu_ticks_total` field, this keeping track of total seconds the process has been scheduled (spent time in the CPU).

The "ps" Command

The following files were modified to add the `ps` system call.

- `ps.c`. This file was created to run the `ps` command, using a system call. A `max` table size is determined, and an array of `uproc` structures is allocated with `malloc()` (Line 23). The max table size and newly allocated `utable` are passed into the system call, `getprocs`, in order to be populated with copied information about active processes. The system call returns an `int` value for the number of active processes copied (Line 25). This active process count is checked to determine an error code return, and then checks for 0 active processes. Both instances free the dynamic memory and exit the program (Lines 27 - 37). If it is determined that the number of copied processes number at least 1, and no more than `max`, the program will loop through the newly populated `utable` and display all of the information of each `uproc` (Lines 39 - 49). After all of the processes have been displayed, the dynamic memory is freed and the program exits.

- `sysproc.c`. A new kernel-side system call, `int sys_getproc(void)`, was added to execute the `ps` command (Lines 162 - 174). It declares local a `uint` variable and a `struct uproc` pointer, then pulls the arguments off of the stack to populate the local variables (Lines 165 - 172). The variables are then passed to another function of the same name (`int getprocs(uint max, struct uproc *table)`), implemented in `proc.c` (Line 173).

- `proc.c`. This function traverses the process table (`ptable`), determines active processes (`RUNNING`, `RUNNABLE`, or `SLEEPING`) and then copies the relevant information into the `uproc` array.
  A count is kept of the number of processes copied, and that number is returned to `sys_getprocs()`, which then returns that value back to the calling routine in `ps.c`. A local `int` variable is declared to keep count of the copied processes and then be the return value (Line 620). A local `struct proc` pointer is declared, to be pointed at the process table (Line 621) after the lock is aquired (Line 622). The `proc` pointer is directed at the first element of `ptable`, and proceeds to loop through each index of the process table, checking for active states (Line 623 - 625) as defined above. Upon reaching an active process, the fields of `struct proc` are copied into the current index of `struct uproc`, at which point the index is incremented (Lines 626 - 639). The lock is released, and the `uproc table` index is returned to the calling routine (Lines 642, 643).

- `uproc.h`. The `uproc` struct was added because all of the fields of the original `struct proc` that we are copying from are not needed for the implementation of `ps.c`.
  The fields of `uproc` are:

```
uint pid; // Process ID
uint uid; // User ID
uint gid; // Group ID
uint ppid; // Parent Process ID
uint elapsed_ticks; // time since process was scheduled, in miliseconds
uint CPU_total_ticks; // total time process has spent in CPU, in miliseconds
char state[STRMAX]; // RUNNING, RUNNABLE, or SLEEPING
uint size; // size of process in bytes
char name[STRMAX]; // name of process
```

- `usys.S`. The user-side stub for the new system call was added (Line 39).

- `user.h`. The user-side function prototype for the `ps` system call was added (Line 39). The system call takes an unsigned integer and a pointer to a user-defined `struct uproc`. The prototype is:
  `int getprocs(uint, struct uproc*);`
  The file `uproc.h` contains the `uproc` definition.

- `syscall.h`. The `getprocs` system call number was created by appending the existing list (Line 33).
- `syscall.c`. Modified to include the kernel-side function prototype (line 111); an entry in the function dispatch table `syscalls[]` (line 146); and an entry into the `syscallnames[]` array to print the system call name when the `PRINT_SYSCALLS` flag is defined (line 185). All prototypes here are defined as taking a *void* parameter as the function call arguments are passed into the kernel on the stack. Each implementation (`sys_getprocs()`) retrieves the arguments from the stack according to the syntax of the system call.
- `ps-test` Test program and command supplied by Instructor Morrissey.

Time User Command

The `time` user command will determine the number of seconds that a program takes to run. This command expects at least one argument: a string containing the name of the program or command that we wish to time. The `time` command will `fork` itself and `exec` the child process, timing how long it takes. There can be multiple `time` commands called within the same command line, in which case they will recursively time how long it takes the subsequent invocations to time their subsequent invocations, and so on. In the case of no program name being provided, an output of `0.00` seconds is displayed. The `time` command is based on the global variable `ticks`.
The file `time.c` was edited to add the `time` command:

- The `time` user command begins by checking arguments passed in from the command line. If there are zero or less arguments, the program simply exits immediately (lines 8 - 10). This is simply for checking for an error, there will always be at least one argument when this program is run: itself.
- Local (`uint`) variables are declared to keep track of: ticks before the program argument is time tested, ticks afterwards, the time it took to run the program, and then integers to calcualte various decimal places (line 11).
- When there is only one argument, which will always be `time`, the program will output `0.00` seconds, and exit (lines 12 - 15).
- When there are 2 or more arguments, this means at least one program has been provided to time test.
- The `start_time` is recorded by invoking the system call `upticks()` (line 16), and `fork()` is launched (line 17).
- Since `fork()` returns twice, we have one condition for if the `pid` returned is greater than 0, which means parent process. Parent processes are directed to `wait` for the child process to finish (lines 18 - 20).
- Another conditional checks if `pid` is 0, which means child process, which then calls `exec(argv[1], (argv + 1));` (lines 21 - 24).
- The first index in `argv[]` will always be `time`, and so this uses the second argument name, and uses pointer arithmetic with the second argument, directing the `exec` command to the 2nd index in the `argv` array.
- `end_time` is recorded by once again calling `upticks()` (line 25), and `elapsed_time` is calculated (line 26). These are in miliseconds, and the print functionality does not allow for directly printing floating point numbers, so minor calculations are needed.
- The ouput needs to be in format `x.xx`, and so modulo arithmetic is used to determine the `sec` whole number, and the tenth and hundredth decimal places (lines 27 – 29).
- The results are printed to the screen, and the program exits (lines 31, 32).

Further control–p Modifications

The control–p console command prints debugging information to the console. The following modifications were made to capture and display elapsed time as part of the existing control–p debugging information. The routine `procdump()` in `proc.c` was further modified from Project 1 to display more information:

- Print a header (lines 571, 572) to the console, constisting of:

  `PID, Name, UID, GID, PPID, Elapsed, CPU, State, Size, PCs`

- Calculate the *elapsed time* since process creation (lines 537–549). This section calculates elapsed time as seconds and hundredths of seconds as the granularity of the `ticks` variable is at hundredths of a second.

- Display fields in appropriate order for this redesign (lines 598–602).

- Certain blocks of code are wrapped in a preprocessor statement `#ifndef CS333_P2` (lines 562–567; 584–586; 590–594). This is to ensure to output the changes to `ctrl-p` properly while the `P2` flag is on in the `Makefile`, while also allowing for the flags for Program 1 to be reenabled and still function how it used to.222

Testing

UIDs, GIDs, & PPIDs

- Test: Correct setting of the UID/GID with a valid parameter for the system call. (see Figure 1)
  This test `PASSES`.

- Test: Correct behavior of the UID/GID system calls with invalid parameters (including returning an error code). (see Figure 1)
  This test `PASSES`.

- Test: Correct PPID returned for the `getppid()` system call. (see Figures 1, 2)
  This test `PASSES`.

  These three tests can be demonstrated in the `testids` user command. The PPID being correctly set can also be seen with the `ctrl-p` and `ps` commands. Both will be demonstrated.
  See Figures 1 and 2.

  Figure 1: Using `testids` user command.

  This demonstrates the correct setting of UID/GID fields with valid parameters. This also demonstrates correct behavior of the UID/GID system calls, showing how error codes are returned and handled. Getting parent process is also shown.

  Figure 2: Using `ps` command to show proper setting and getting of PPID.

  The `ps` command allows us to follow (Figure 2) how the processes are given PIDs and PPIDs, and we can see that the processes are correctly labeled. We can also see that `fork()` is properly setting UID/GID, as they are all supposed to be 0.

  These tests all `PASS`.

- Test: Using the shell built-in commands to show proper functioning of the UID/GID system calls.
  For this test I used the built-in shell commands, `_set uid` ***`int`***, `_set gid` ***`int`***, `_get uid`, and `_get gid`, typing them in to the command prompt. (see Figure 3)

  Figure 3: Using shell commands for UID and GID

  The output from the xv6 shell shows that the 4 built-in commands for setting/getting UIDs and GIDs works as expected.

  This test `PASSES`.

- Test: Correct setting of the UID/GID by the `fork()` system call. The `fork()` system call copies existing processes, and currently the UID and GID are both initilaized in `allocproc()` to 0.
  Figure 2 properly demonstrates `fork()` correctly copying the initial values.
  Figure 4 demonstrates that `fork()` properly copies UIDs and GIDs set manually via shell prompt and system call.
  This test `PASSES`.

  Figure 4: `fork` copies manually set UIDs and GIDs.

The "ps" Command

- Test: `getprocs()` tests for the parameter `max` set to 1, 16, 64, and 72 to show that the data structure is fileld properly for the specific values of `max`. (See Figures 8-13) This test `PASSES`.
- Test: `ps` command. Show that the correct processes are output and that the information matches `ctrl-p`.
  For this test, I alternated entering `ctrl-p` and `ps` at the shell prompt, in order to demonstrate the identical outputs of the two functions. The proper elapsed time is also shown.
  (See Figure 5).
  This test `PASSES`.

Figure 5: `ctrl-p` and `ps` command

- Test: Correct elapsed time shown in `ctrl-p`. I tested this by launching xv6 and pressing `ctrl-p` roughly every second.
  (See Figure 6).

  This test `PASSES`.

Figure 6: Correctly shows elapsed time with `ctrl-p`.

Time User Command

- Test: Time command:
  - No argument case.
    This is tested by entering "time" into the shell prompt. 0.00 seconds is returned.
    (See Figure 7).
    This test `PASSES`.
  - Bad command.
    This is tested by entering a nonsense string of characters into the shell prompt. Time is taken to run the command and determine it is nothing. Normal return to shell.
    (See Figure 7).
    This test `PASSES`.
  - Command with no arguments.
    This was tested by entering `time ps` at the shell prompt (`time` was called on itself as well to demonstrate further.) `ps` takes no arguments. Time returned the execution time of `ps` and the second `time` command.
    (See Figure 7).
    This test `PASSES`
  - Command with one or more arguments.
    This was tested by entering `time echo "string"` at the shell prompt. The command `echo` takes an argument and prints it back out. `time` successfully returned the execution time of this command.
    (See Figure 7).
    This test `PASSES`.

Figure 7: `time` command and the 4 tests

All tests have been determined to be `PASSING`.

Figure 8: max set to 1

Figure 9: max set to 16

Figure 10: max set to 32

Figure 11: max set to 64

Figure 12: max set to 72

Figure 13: max set to 72