Project 2: Implement various system call focused on process in xv6 – Updates

Part-1 Getting process statistics and modifying system call in xv6

System Environment

Operating System: XV6

Compiler/Development Environment: GCC (GNU Compiler Collection).

Languages Used: C

Overview

In this assignment, we extended the xv6 operating system by adding three essential fields to the process control block (proc structure): **creation time**, **end time**, and **total time**.

We implemented a new system call to record and retrieve this information, allowing us to better understand the process life cycle.

Additionally, we created a **test.c** script that invokes existing functions like 'uniq' and 'head' and can be easily expanded to call newly created functions.

This enhancement provides valuable insights into process execution, making xv6 more powerful and informative for users and developers alike.

Code Structure

Process Control Block (PCB) Enhancements

Added new fields to the **struct proc** (Process Control Block):

- **ctime**: Represents the creation time of a process.
- **etime**: Signifies the end time of a process.
- **rtime**: Records the total time a process has executed.

User-Space Testing Script `test.c`

This C code tests the **`waitx`** system call in xv6. It forks a child process, executes a specified program in the child, and then waits for it to complete using **`waitx`**. Finally, it prints the runtime, creation time, and end time of the child process.

`waitx` system call

It acquires the process table lock to ensure exclusive access to process-related data structures.

It enters a loop to scan through the process table (**ptable**) and look for exited children processes (**ZOMBIE** state).

When it finds a child process in the **ZOMBIE** state, it extracts essential information such as process ID (**pid**), creation time (**ctime**), and end time (**etime**) from the process structure (**struct proc**).

It calculates the runtime (**rtime**) of the child process by subtracting the creation time from the end time and stores these values.

The process-related fields (**ctime**, **etime**, **rtime**) in the child process structure are then reset to zero to avoid any lingering values.

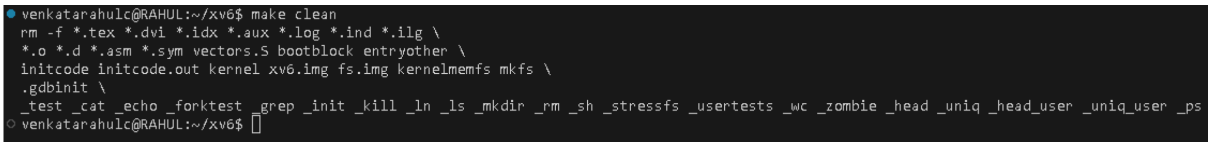
Finally, it releases the process table lock, cleans up the child process's resources (e.g., memory, stack), and returns the child process's PID.

Approach

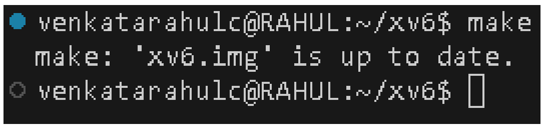
- Problem Understanding: Understood the requirement to add a new system call `waitx` in xv6 to retrieve runtime, creation time, and end time information of child processes.
- `waitx` System Call Design: Designed the `waitx` system call with three integer pointers as arguments (int *rtime, int *ctime, int *etime) to hold runtime, creation time, and end time.
- Process Structure Enhancement: Modified the struct proc in proc.h to include new fields `ctime`, `etime`, and `rtime`.
- Field Initialization: Implemented code to initialize these fields `ctime` during process creation, `etime` when a process exits, `rtime` calculated as the difference between `etime` and `ctime`.
- `waitx` Implementation: Acquiring the process table lock. Locating exited child processes in the ZOMBIE state. Extracting process information, including PID, `ctime`, and `etime`. Calculating `rtime`. Resetting `ctime`, `etime`, and `rtime` for consistency. Releasing the process table lock and returning the child's PID.
- Main Function: The main function serves as the entry point of the program. It handles command-line arguments and orchestrates the execution of the "head" command.

Steps to Run

- **Compile the program**
    - Use the Xv6 build system to compile the waitx utility.

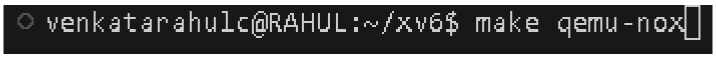


```
venkatarahulc@RAHUL:~/xv6$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs mkfs \
.gdbinit \
_test _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _head _uniq _head_user _uniq_user _ps
venkatarahulc@RAHUL:~/xv6$ 
```

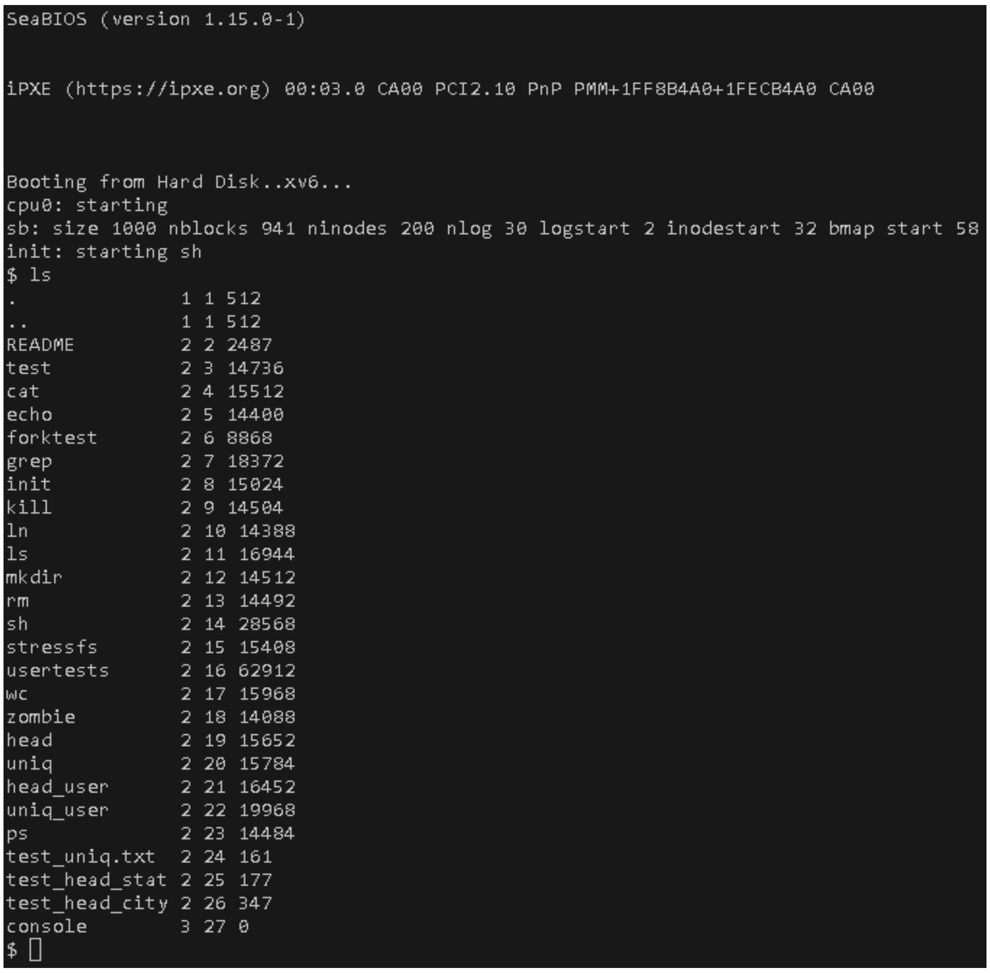


```
venkatarahulc@RAHUL:~/xv6$ make
make: 'xv6.img' is up to date.
venkatarahulc@RAHUL:~/xv6$ 
```

- **Access the Xv6 Environment**
  - Boot or launch the Xv6 operating system on the system or in an emulator, such as QEMU

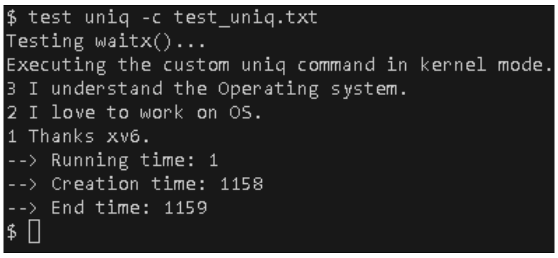


```
○ venkatarahulc@RAHUL:~/xv6$ make qemu-nox
```



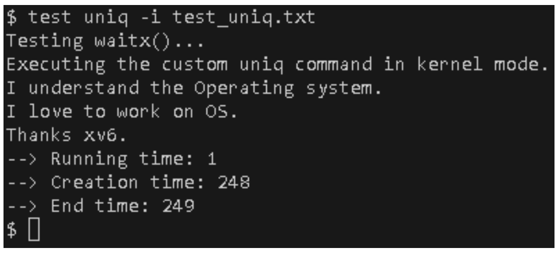
```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2487
test           2 3 14736
cat            2 4 15512
echo           2 5 14400
forktest       2 6 8868
grep           2 7 18372
init           2 8 15024
kill           2 9 14504
ln             2 10 14388
ls             2 11 16944
mkdir          2 12 14512
rm             2 13 14492
sh             2 14 28568
stressfs       2 15 15408
usertests      2 16 62912
wc             2 17 15968
zombie         2 18 14088
head           2 19 15652
uniq           2 20 15784
head_user      2 21 16452
uniq_user      2 22 19968
ps             2 23 14484
test_uniq.txt  2 24 161
test_head_stat 2 25 177
test_head_city 2 26 347
console        3 27 0
$ 
```

- **Run the Program**
  - test uniq -c “filename”



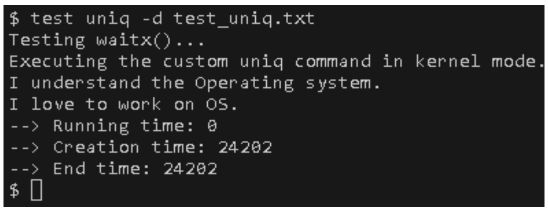
```
$ test uniq -c test_uniq.txt
Testing waitx()...
Executing the custom uniq command in kernel mode.
3 I understand the Operating system.
2 I love to work on OS.
1 Thanks xv6.
--> Running time: 1
--> Creation time: 1158
--> End time: 1159
$ 
```

  - test uniq -i “filename”



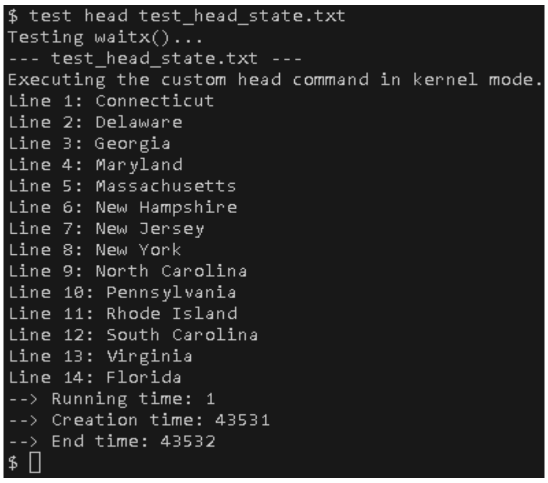
```
$ test uniq -i test_uniq.txt
Testing waitx()...
Executing the custom uniq command in kernel mode.
I understand the Operating system.
I love to work on OS.
Thanks xv6.
--> Running time: 1
--> Creation time: 248
--> End time: 249
$ 
```

- test uniq -d “filename”



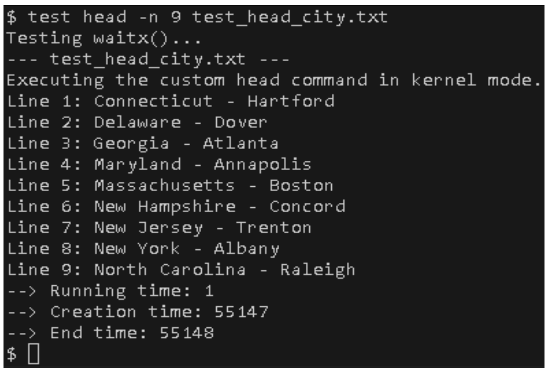
```
$ test uniq -d test_uniq.txt
Testing waitx()...
Executing the custom uniq command in kernel mode.
I understand the Operating system.
I love to work on OS.
--> Running time: 0
--> Creation time: 24202
--> End time: 24202
$ 
```

- test head “filename”



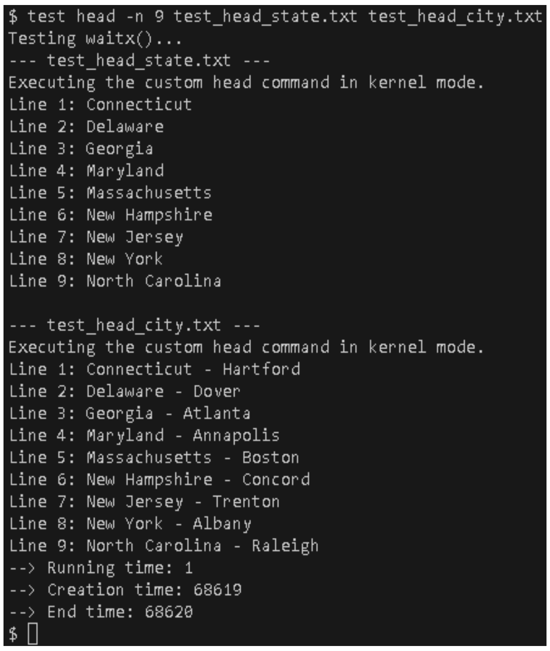
```
$ test head test_head_state.txt
Testing waitx()...
--- test_head_state.txt ---
Executing the custom head command in kernel mode.
Line 1: Connecticut
Line 2: Delaware
Line 3: Georgia
Line 4: Maryland
Line 5: Massachusetts
Line 6: New Hampshire
Line 7: New Jersey
Line 8: New York
Line 9: North Carolina
Line 10: Pennsylvania
Line 11: Rhode Island
Line 12: South Carolina
Line 13: Virginia
Line 14: Florida
--> Running time: 1
--> Creation time: 43531
--> End time: 43532
$ 
```

- test head -n “number” “filename”



```
$ test head -n 9 test_head_city.txt
Testing waitx()...
--- test_head_city.txt ---
Executing the custom head command in kernel mode.
Line 1: Connecticut - Hartford
Line 2: Delaware - Dover
Line 3: Georgia - Atlanta
Line 4: Maryland - Annapolis
Line 5: Massachusetts - Boston
Line 6: New Hampshire - Concord
Line 7: New Jersey - Trenton
Line 8: New York - Albany
Line 9: North Carolina - Raleigh
--> Running time: 1
--> Creation time: 55147
--> End time: 55148
$ 
```

- test head -n “number” “filename1” “filename2”



```
$ test head -n 9 test_head_state.txt test_head_city.txt
Testing waitx()...
--- test_head_state.txt ---
Executing the custom head command in kernel mode.
Line 1: Connecticut
Line 2: Delaware
Line 3: Georgia
Line 4: Maryland
Line 5: Massachusetts
Line 6: New Hampshire
Line 7: New Jersey
Line 8: New York
Line 9: North Carolina

--- test_head_city.txt ---
Executing the custom head command in kernel mode.
Line 1: Connecticut - Hartford
Line 2: Delaware - Dover
Line 3: Georgia - Atlanta
Line 4: Maryland - Annapolis
Line 5: Massachusetts - Boston
Line 6: New Hampshire - Concord
Line 7: New Jersey - Trenton
Line 8: New York - Albany
Line 9: North Carolina - Raleigh
--> Running time: 1
--> Creation time: 68619
--> End time: 68620
$ 
```

Part-2 Implementing ps on xv6

System Environment

Operating System: XV6

Compiler/Development Environment: GCC (GNU Compiler Collection).

Languages Used: C

Overview

This enhancement adds a custom **ps** command to the xv6 operating system, enabling users to obtain essential process information for better system analysis and monitoring. The **ps** command displays process details, including the Process ID (PID), status (e.g., running, zombie, waiting), start time, total execution time, and process name. This valuable feature enhances the usability of xv6 by providing users with a comprehensive view of process statuses and execution times.

This implementation includes a new system call **ps**, and a user-level program **ps.c**, to access and display process data.

Code Structure

ps.c (User-Level Program):

- **ps.c** is the user-level program that provides a command-line interface to the custom **ps** command.
- It includes necessary header files: **types.h**, **stat.h**, **user.h**, and **fcntl.h**.
- In **main()**, it first prints a header row with column names (process name, PID, state, ctime, rtime).
- It checks the number of command-line arguments (**argc**).
- If there are no arguments (only the program name), it calls **ps("")** to display information for all processes.
- If there are additional arguments, it iterates through them and calls **ps(argv[i])** to display information for processes with a specific name.

sysproc.c (System Call Interface):

- **sys_ps()** is defined in **sysproc.c** and serves as the interface between the user-level **ps** program and the kernel.
- It takes the process name as an argument using **argstr()**.
- If the argument extraction is successful, it calls the kernel function **ps(pname)** and returns the result.

proc.c (Kernel Implementation):

- The **ps()** function in **proc.c** implements the core logic for retrieving and displaying process information based on the provided process name.
- It begins by enabling interrupts using **sti()** to avoid potential race conditions.

- The process table (**ptable**) is locked using **acquire(&ptable.lock)** to ensure exclusive access.
- If the provided process name is empty (i.e., all processes should be displayed), it iterates through all processes in the process table (**for** loop).
- Within the loop, it checks the state of each process and prints its information (name, PID, state, ctime, rtime) accordingly.
- If a specific process name is provided, it only displays information for processes with a matching name.
- The **release(&ptable.lock)** statement releases the lock, allowing other processes to access the process table.
- The function returns a placeholder value (**25**) to signify the completion of the **ps** command.

Approach

Understanding the Task: To begin, I needed to grasp the task at hand, which was to enhance the xv6 operating system with a custom ps command capable of displaying process information.

User-Level Program (ps.c): The first step was to create a user-level program called ps.c. This program would serve as the entry point for users to request process details.

Handling User Inputs: To make the ps command versatile, I added code in ps.c to handle command-line arguments. This allowed users to specify which processes they wanted to see. Importantly, I ensured that if users didn't provide any arguments, the command would display a general list of processes.

Introducing a New System Call (sysproc.c): In the kernel space, specifically in sysproc.c, I introduced a new system call, which I named sys_ps(). This system call would be responsible for facilitating the transfer of process name arguments from user space to the kernel.

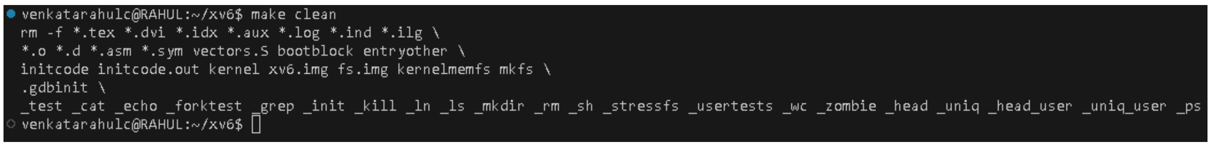
Kernel Implementation (proc.c): In the core of the kernel code, within proc.c, I implemented the ps(char *pname) function. To maintain proper synchronization and avoid concurrency issues, I used process table locks. Within this function, I systematically scanned through the list of processes, checking their states and names. Depending on the state and name of each process, I printed relevant information such as the Process ID (PID), status (e.g., running, sleeping), start time, and total execution time. After processing the required information, I responsibly released the process table lock.

Handling Return Values: Finally, I ensured that the ps(char *pname) function returned a designated value (e.g., 25) to signify successful execution.

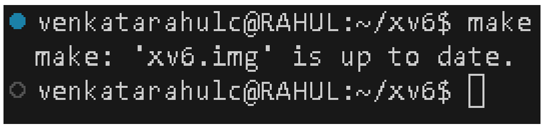
Steps to Run

- **Compile the program**

- Use the Xv6 build system to compile the ps utility

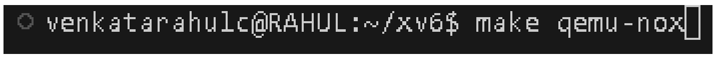


```
venkatarahulc@RAHUL:~/xv6$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs mkfs \
.gdbinit \
_test _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _head _uniq _head_user _uniq_user _ps
venkatarahulc@RAHUL:~/xv6$ 
```

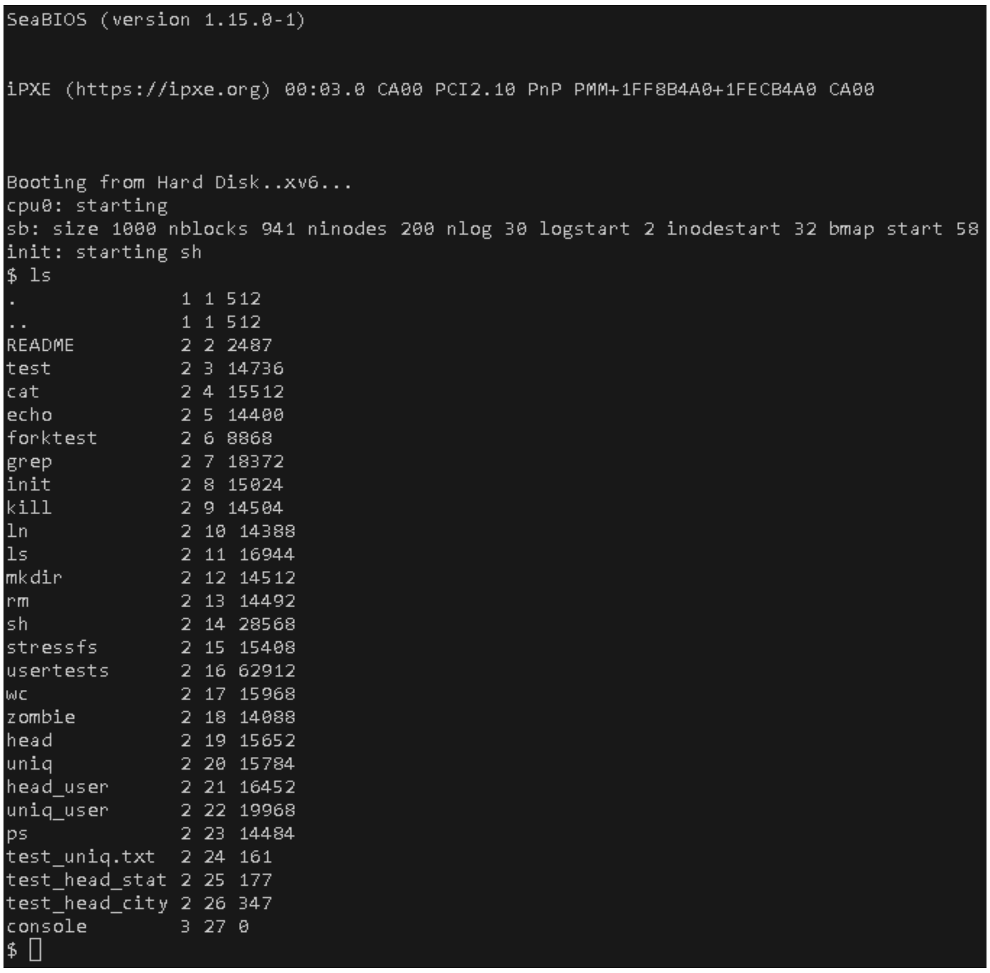


```
venkatarahulc@RAHUL:~/xv6$ make
make: 'xv6.img' is up to date.
venkatarahulc@RAHUL:~/xv6$ 
```

- **Access the Xv6 Environment**
  - Boot or launch the Xv6 operating system on the system or in an emulator, such as QEMU

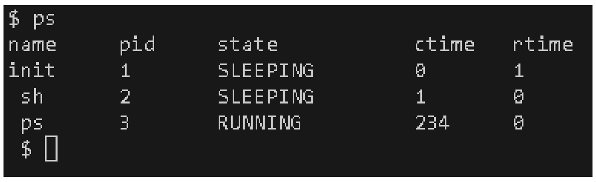


```
venkatarahulc@RAHUL:~/xv6$ make qemu-nox
```



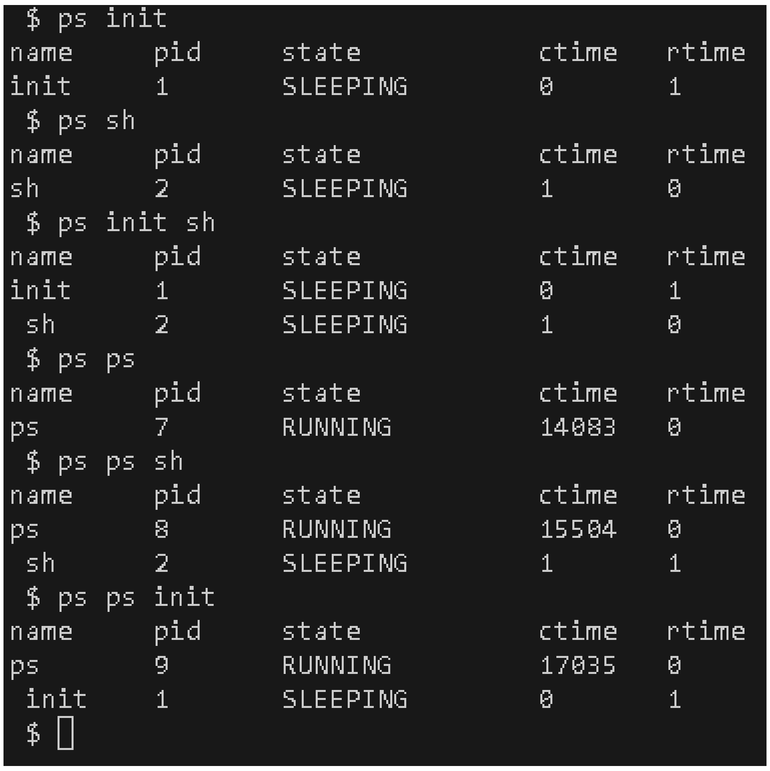
```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2487
test           2 3 14736
cat            2 4 15512
echo           2 5 14400
forktest       2 6 8868
grep           2 7 18372
init           2 8 15024
kill           2 9 14504
ln             2 10 14388
ls             2 11 16944
mkdir          2 12 14512
rm             2 13 14492
sh             2 14 28568
stressfs       2 15 15408
usertests      2 16 62912
wc             2 17 15968
zombie         2 18 14088
head           2 19 15652
uniq           2 20 15784
head_user      2 21 16452
uniq_user      2 22 19968
ps             2 23 14484
test_uniq.txt  2 24 161
test_head_stat 2 25 177
test_head_city 2 26 347
console        3 27 0
$ 
```

- **Run the Program**
  - ps



```
$ ps
name     pid      state            ctime     rtime
init     1        SLEEPING         0         1
 sh      2        SLEEPING         1         0
 ps      3        RUNNING          234       0
 $ 
```

- ps “pname” or ps “pname1” “pname2” ……..



```
 $ ps init
name      pid      state          ctime    rtime
init      1        SLEEPING       0        1
 $ ps sh
name      pid      state          ctime    rtime
sh        2        SLEEPING       1        0
 $ ps init sh
name      pid      state          ctime    rtime
init      1        SLEEPING       0        1
 sh       2        SLEEPING       1        0
 $ ps ps
name      pid      state          ctime    rtime
ps        7        RUNNING        14083    0
 $ ps ps sh
name      pid      state          ctime    rtime
ps        8        RUNNING        15504    0
 sh       2        SLEEPING       1        1
 $ ps ps init
name      pid      state          ctime    rtime
ps        9        RUNNING        17035    0
 init     1        SLEEPING       0        1
 $ 
```

Resources Used

- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjl-7jmo8qBAxU6mWoFHSwbBCkQFnoECBoQAQ&url=https%3A%2F%2Fwww.cse.iitb.ac.in%2F~mythili%2Fos%2Fanno_slides%2Flecture22.pdf&usg=AOvVaw3lDBVzi8fhmAkFwIcV_E2K&opi=89978449
- https://medium.com/@harshalshree03/xv6-implementing-ps-nice-system-calls-and-priority-scheduling-b12fa10494e4
- https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjer7yfpMqBAxVolWoFHSDkDpgQFnoECCUQAQ&url=https%3A%2F%2Fwww.cse.iitb.ac.in%2F~mythili%2Fos%2Fanno_slides%2Flecture23.pdf&usg=AOvVaw0aTV1MCpTPr7e9TlPkxIdD&opi=89978449
- XV6 Documentation