

Lab 3: to infinity and beyond with xv6

Instructions

- This course is on a **no-plagiarism** diet. All parties involved in plagiarism harakiri will be penalized referred to DADAC and penalized to the maximum extent.
- The Moss Detective Agency has agreed to conduct all investigations.
<https://theory.stanford.edu/~aiken/moss/>
Byomkesh, Sherlock, Phryne, Marple, and Hercule are on standby.
- Hardcoding expected output of execution in source code will yield **negative** marks.
- Generative AI (ChatGPT, Gemini, etc.) is your friend, but is not you!
Analytical, critical and creative thinking are learning outcomes of this course and source code or any content via GenAI tools cannot be part of your submissions.
- **Note:** Submission guidelines to be strictly followed; otherwise, your submission will not count.
- Login with username your **CSE LDAP ID** or **labuser** on software lab machines for this lab.
- This file is part of the **lab2.tar.gz** archive which contains multiple directories with programs associated with the exercise questions listed below.
- Most questions of this lab are tutorial style and are aimed to provide an introduction to tools and files related to system and process information and control.

In this lab we will learn how to use the xv6 operating system, learn how system calls are implemented, explore examples of OS metadata and actions and dabble with the memory management subsystem.

Task 0: Setting up xv6

<https://www.cse.iitb.ac.in/~puru/courses/autumn2024/lectures/cs744-xv6.pdf>

Task 1: Adding new programs to the xv6 environment

(a) pingpong with xv6

Write a program named **pingpong** which reads a text file as an input argument and outputs **pong** to the standard output every time it finds the word **ping** in the input text file.

- I. Additions to the `Makefile` will be needed to add new programs for compilation and also to be included as part of the xv6 viewable disk image (to read/write files e.g., `abc.txt`, `hello.txt`) via `fs.img`
Look for the following keywords in the `Makefile`.

UPROGS=\

Lists names of all user programs which are available after xv6 boot up.

EXTRA=\

List of all files (source programs and other scripts and data files) available after xv6 bootup.

fs.img

List of files to be added to the xv6 startup disk (imagefile).

- II. xv6 OS itself does not have a text editor or compiler support, all source code of programs has to be written and compiled on the host machine, all its references added to the makefile and then via fs.img and xv6.img be used via QEMU emulator.
- III. You will need to include input text files for e.g., pingpong.txt or any other files in the xv6 OS image that you will use for running the program.
Refer to the README included in xv6 image.
- IV. Sample input file **pingpong.txt** is provided as part of this lab archive file.
Consider using **wc.c**, source code of the **wc** program as a starting point for this task.

```
≡ pingpong.txt
1 Hello this is the first occurrence of ping.
2 You must have used ping command in Linux.
3 A process could use the ping system call to check network connectivity.
4 In the xv6 operating system, system calls act as the bridge between user-level applications and the kernel.
5 A process could use a custom ping system call to send special messages between processes.
6
```

Sample usage

```
$ pingpong pingpong.txt
pong
pong
pong
pong
$
```

(b) inception (shell in a shell)

Write a program **cmd.c** that creates a child process — the child process executes a program, and the parent process waits till completion of the child process before terminating. This program should use the **fork** and **exec** system calls of xv6. The program to be executed by the child process can be any of the sample xv6 programs and should be specified at the command line.

Refer to Sheet 66,85 of [xv6 source code booklet](#) for `fork()`, `exec()` system calls in xv6.

Sample usage

```
$ cmd ls
 1 1 512
.. 1 1 512
README 2 2 2286
cat 2 3 15488
head 2 4 15844
cmd 2 5 14792
echo 2 6 14368
forktest 2 7 8812
grep 2 8 18332
init 2 9 14988
kill 2 10 14456
ln 2 11 14352
ls 2 12 16920
mkdir 2 13 14476
rm 2 14 14456
sh 2 15 28512
stressfs 2 16 15388
usertests 2 17 62888
wc 2 18 15912
zombie 2 19 14036
console 3 20 0
$ cmd echo hello xv6 os !!
hello xv6 os !!
$
```

Task 2: Adding new system calls to xv6

To understand and work with system calls and process related information and action, the following files of the xv6 OS are important —

`usys.S`, `user.h`, `defs.h`, `sysproc.c`, `syscall.h`, `syscall.c`, `proc.h`, `proc.c`

- **user.h** contains the xv6 system call declarations
- **usys.S** contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction
- **syscall.h** contains the mapping of system call name to system call number
- **syscall.c** contains helper functions to handle the system call entry, parse arguments, and pointers to the actual system call implementations
- **sysproc.c** contains the implementations of process related system calls
- **defs.h** is a header file with function declarations in the xv6 kernel
- **proc.h** contains the process abstraction related variable definitions
- **proc.c** contains implementations of various process related system calls, functions and the scheduler, also contains the declaration of `ptable`, and several examples of functions traversing/using the process list
- System call related functions are also listed in **sysfile.c**

All or most of these files will have to be used/updated to implement new system calls. New files, new programs, new data files need to be added to xv6 via the xv6 Makefile. All changes are to be followed by a clean compile and build, followed by executing xv6.

Note that xv6 itself does not have a text editor or compiler support, so all xv6 source code changes are on the host machine and then the **xv6.img** and **fs.img** files are used as inputs to QEMU.

(a) hello, system calls!

Implement a system call, with the following declaration **worldpeace()**, which prints the message “[Systems are vital to world peace !!](#)” in the kernel mode.

The function **cprintf** is used for printing in the kernel mode (refer to sheet 30 line 3026 of [xv6 source code booklet](#) for usage).

A simple test program **worldpeace.c** is also provided to test your implementation.

ChatGPT's take on systems + world peace is [here](#).

Sample usage

```
$ test-message
Systems are vital to world peace !!
$
```

(b) who all are ready?

Implement a system call, with the following declaration **numberofprocesses()** which returns the total number of processes in **READY**(xv6 naming is **RUNNABLE**) state to the user program.

Refer to the **PCB** structure defined on line 2336 sheet 23 of [xv6 source code booklet](#) and lines 10-14 in **proc.c** in the given source code to refer to **struct ptable**.

Refer to sheet 24 Line 2480 to understand how to iterate through the process table and sheet 23 Line 2334 to for the process state enum of [xv6 source code booklet](#) .

A simple test program **nump.c** is also provided to test your implementation.

Sample usage

```
$ test-numberofprocesses
ready processes: 2
ready processes: 1
ready processes: 0
$
```

(c) spawn — one call, many processes!

Implement a system call, with the following declaration `int spawn(int n, int* pids)` which creates `n` child processes when invoked.

- The system call must return 0 to each of the child processes and number of children created to the parent process.
- Additionally, the `pids` array should contain the pids of the spawned children after the `spawn` system call. The parent should gracefully reap all the child processes which are present in the `pids` array.

A simple test program `spawn.c` is also provided to test your implementation.

Note: `argint`, `argstr`, `argptr` are helper functions for handling system calls arguments. (Refer to sheet 65 line 6557 of [xv6 source code booklet](#) for `argptr` usage)

Refer to `fork()` system call implementation in `proc.c` to understand how a child process is created and how the call handles return values for parent and child processes.

Sample usage

```

$ test-spawn 3
[P] Child PID list: 8 9 10
[C] Spawned child pid 8
[C] Spawned child pid 9
[C] Spawned child pid 10
[P] reaped process with id 9
[P] reaped process with id 10
[P] reaped process with id 8
```

Task 3: what is your address?—virtual and physical !

(a) is the virtual address space real?

Write a system call `getvasize` that returns the size of the address space used by a process. Specifically, the system call should have the following interface:

```
int getvasize(int pid);
// pid of a process is argument to the call and amount of virtual
// memory used by the process is the return value
```

Hints:

- (i) Look up and understand implementation of the **sbrk** system call in **proc.c**. Also, check the **struct proc** data structure in **proc.h**
- (ii) Refer to Sheet 38 of [xv6 source code booklet](#) for sbrk() system call in xv6.
- (iii) Refer to discussion on Page 34 of the [xv6 book](#) .

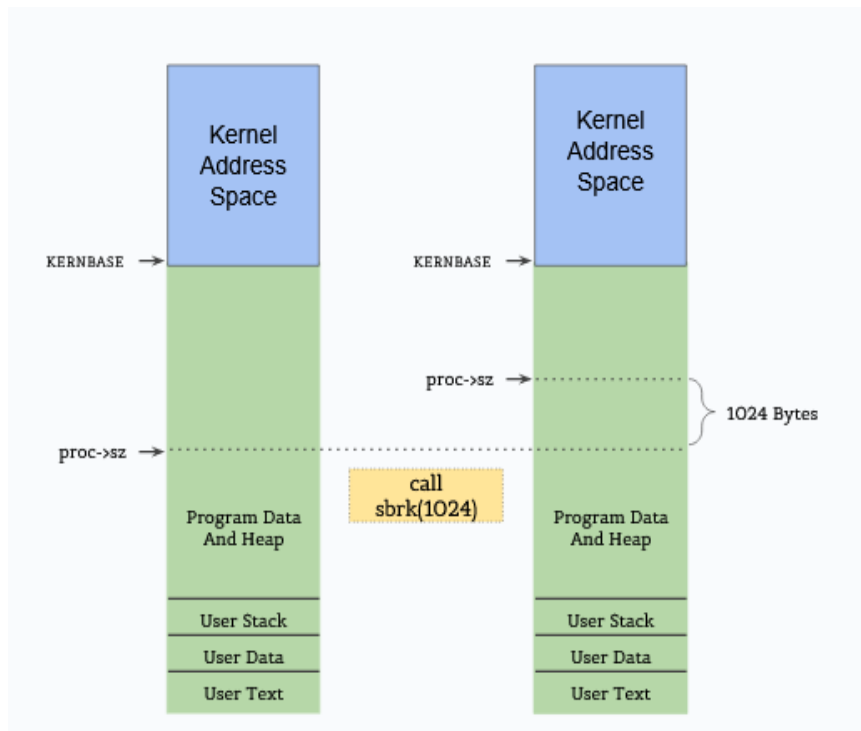


Figure 1. After calling **sbrk(1024)** to increase the process size by 1024.

The **sbrk(n)** system call is implemented in the function **sys_sbrk()** in [sysproc.c](#) that allocates physical memory and maps it into the process's virtual address space. The **sbrk(n)** system call grows the process's memory size by *n* bytes, and then returns the start of the newly allocated region (i.e., the old size).

Figure1: Address space layout of process in xv6. User stack is of one page, followed by the heap up to KERNBASE. **What is the value of the constant KERNBASE?**

A sample program `t_getvasize.c` is available for testing.

```
$ t_getvasize
Pid of the process is 4
Size of process:      12288 Bytes
Address returned by sbrk: 0x3000
Size of process:      13312 Bytes
Address returned by sbrk: x03000
```

(b) What is your postal address?

Write a system call **va2pa** that returns the virtual address to physical address mapping from the page table of the current process. Specifically, the system call should have the following interface:

```
uint va2pa(uint virtual_addr);  
// virtual address is the argument  
// corresponding physical address is the return value
```

Hints:

- (i) Lookup and understand the **walkpgdir()** function and understand usage of this function in the system calls implemented in **vm.c**
- (ii) Refer to Sheet 17 of [xv6 source code booklet](#) for **sbrk()** system calls in xv6.
- (iii) Refer to discussion on Page 29-32 of [the xv6 book](#).

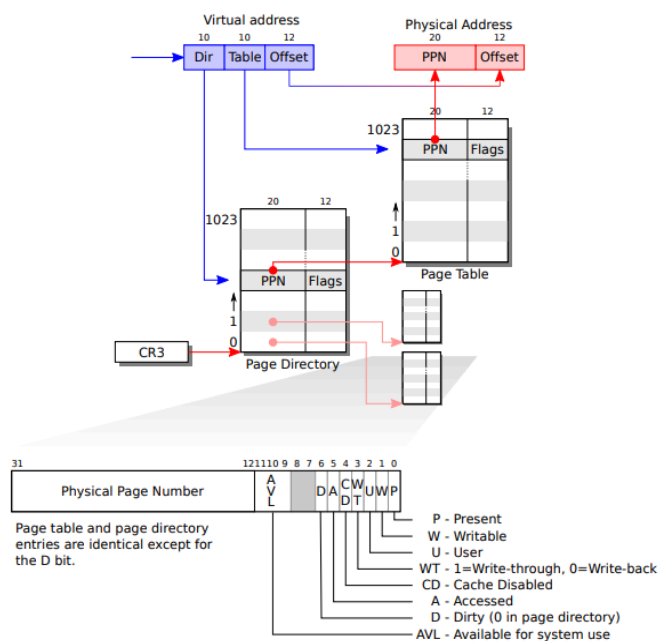


Figure 3: Page table layout

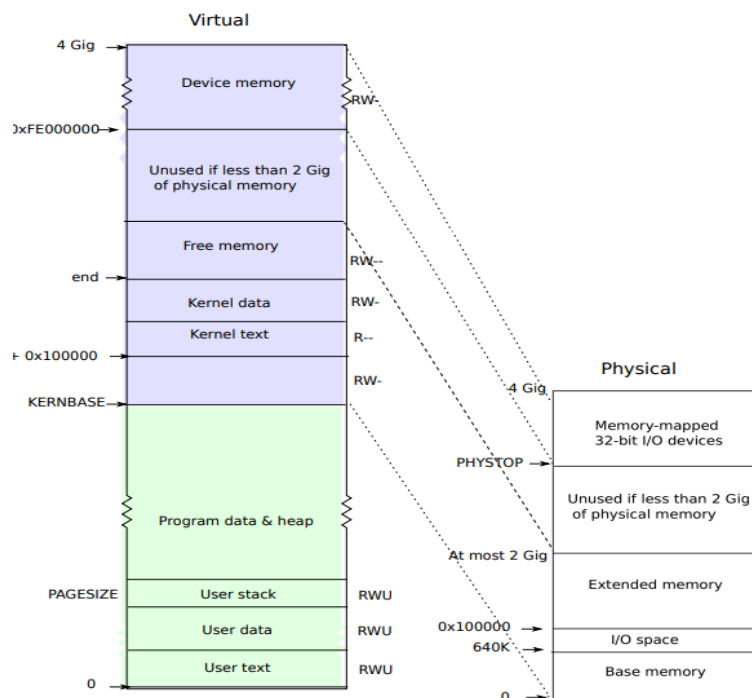


Figure 2: Virtual to physical address mapping

Figure 2. A page table is stored in physical memory as a **two-level tree**. The root of the tree is a 4096-byte page directory that contains 1024 PTE references to page table pages. Each page table page is an array of 1024 32-bit PTEs. The paging hardware uses the MSB 10 bits of a virtual address to select a page directory entry. If the page directory entry is present, the paging hardware uses the next 10 bits of the virtual address to select a PTE from the page table page that the page directory entry refers to. **If either the page directory entry or the PTE is not present, the paging hardware raises a fault.**

Figure 3. A process's address space starts at virtual address zero and can grow up to **KERNBASE**, allowing a process to address up to 2 GB of memory. The file **memlayout.h** declares the constants for xv6's memory layout, and macros to convert virtual to physical addresses. When a process asks xv6 for more memory, xv6 first finds **free physical pages** from the free page list and then adds PTEs to the process's page table that point to the new physical pages. xv6 sets the **PTE_U**, **PTE_W**, and **PTE_P** flags in these PTEs. xv6 includes all mappings needed for the **kernel** to run in every process's page table; these mappings all appear above **KERNBASE**.

Use the above information to traverse the page table of a process and convert virtual address to physical address.

Sample programs **t_va2pa.c** and **t_va2pa2.c** are provided for testing.

Sample usage:

```
$ t_va_to_pa1
Physical Address of user   virtual address 2FCC is DF24FCC
Physical Address of Kernel virtual address 80100400 is 100400
Physical Address of Kernel virtual address 80100800 is 100800
Physical Address of Kernel virtual address 80101000 is 101000
Physical Address of Kernel virtual address 80101448 is 101448
```

```
$ t_va_to_pa2
Virtual address of Var N in child    2FCC
Physical address on Var N in child    DF1FFCC
Virtual address of Var N in parent    2FCC
Physical address on Var N in parent    DF76FCC
```

*** These outputs seem interesting. Why so?**

(c) enter the page table!

Implement the following system calls to get details of the page table of a process.

int get_pgtb_size();

The system call has no arguments and returns the number of page table pages allocated to the current process.

int get_usr_pgtb_size();

The system call has no arguments and returns the number of page table pages allocated for user space memory for the current process.

int get_kernel_pgtb_size();

The system call has no arguments and returns the number of page table pages allocated for kernel space memory for the current process. **Recall** kernel pages are mapped for virtual addresses above **KERNBASE**.

A user-level program **t_getpgtables.c** is provided for testing. This program will call all the above system calls before and after multiple **sbrk()** system calls.

Hint:

Walk the page table of a process based on/using the **walkpgdir** function and consider only those entries which indicate mappings that are present (the present bit is set).

Sample usage

```
$ t_getpgtables
Page Table Size 65 Pages
User Pages Table Size 1 Pages
kernel Pages Table Size 64 Pages
-----Doing sbrk-----
Page Table Size 69 Pages
User Pages Table Size 5 Pages
kernel Pages Table Size 64 Pages
```

(d) no escape from reality!

Next, report the physical memory (in pages) allocated for a process via a system call

int getpasize(int pid);

The call takes **pid** as an argument and prints the number of physical pages mapped to the virtual addresses of a process (process virtual addresses).

NOTE: Count the number of mapped pages by walking the process page table and counting the number of page table entries that have a valid physical address assigned.

You are provided with **t_getpasize.c** for testing,

Sample usage

```
$ t_getpasize
Virtual Address Space of process: 3 Pages
Number of physical pages allocated to process: 3 Pages
Virtual Address Space of process after sbrk: 14 Pages
Number of physical pages allocated to process after sbrk: 14 Pages
```

We will use these system calls to test your implementation in the next task.

Hints:

(i) You can walk the page table of the process by using the **walkpgdir** function which is present in **vm.c**. You can look up **loadvm** and **deallocvm** in **vm.c** to see how to invoke the **walkpgdir** function. To compute the number of physical pages in a process, you can write a function that walks the page table of a process in **vm.c** and invoke this function from the system call handling code.

(ii) xv6 has a 2-level page table organization. You need to calculate the size of the page table (total level 0 and level 1 pages). You need to iterate over the Page Directory Entries (PDEs) to check if a page is assigned for storing Page Table Entries (PTEs) for that PDE.

Task 3: the scam revealed

(a) faulting page, page faulting, who is handling?

The default xv6 distribution does not handle the page fault trap explicitly. Extend implementation of the trap handler function in **trap.c** to explicitly handle a page fault. The handler should print details of the page fault — pid of the process and faulting address which was accessed for the trap. The page fault trap defined in **traps.h** is **T_PGFLT**.

Refer to Sheet 34 of [xv6 source code booklet](#) for **trap()** handler in xv6.

Sample program **t_pagefault.c** is provided for testing.

Sample usage

```
$ t_pagefault
Pid of the process is 8
Simulating a page_fault on addr 4000
```

Hints:

- Look at the arguments to the **cprintf** statements in **trap.c** to figure out how one can find the virtual address that caused the page fault.
 - Once you correctly handle the page fault, do break or return in order to avoid the **cprintf** and the **add proc->killed = 1** statement.
-

(b) mmap()

Implement a version of the `mmap` system call in xv6. The `mmap` system call should take one argument: the number of bytes to add to the **size** of the process. The process size in this context refers to the heap size. The `mmap` call grows the size of the process (virtual) address space and expects a mapped physical address.

However, mapping from a virtual to physical address is required only when the virtual address is accessed!.

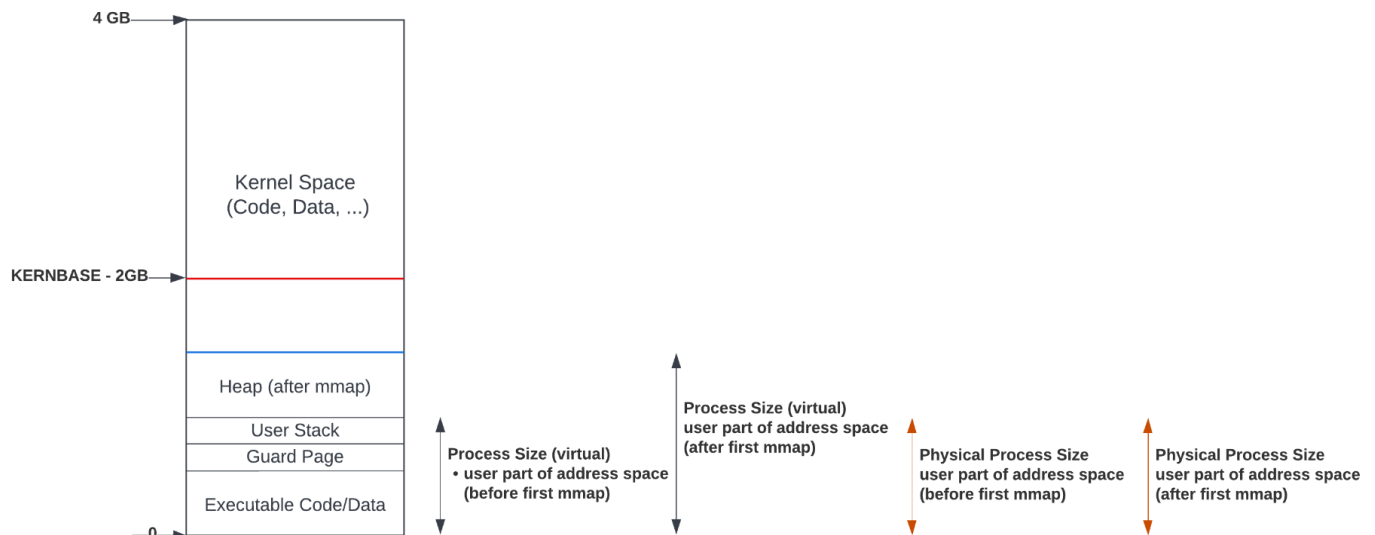


Figure 4: Virtual and physical addresses before and after `mmap()`.

Figure 4 shows the working of the `mmap` system call. The `mmap(1024)` call increases the virtual address space of the process, however the physical address space remains the same.

Assume that the number of bytes is a positive number and is a multiple of the page size. The system call should return a value of 0 if any invalid inputs are provided.

In the valid case, the system call should expand the process's size by the specified number of bytes, and return the starting virtual address of the newly added memory region.

However, the system call should **NOT allocate any physical memory** corresponding to the new virtual pages. When the user accesses a memory-mapped page, a page fault will occur, and physical memory should only be allocated as part of the page fault handling (lazy page allocation and mapping).

Hints:

- (i) **mmap()** system call is similar to **sbrk()** (with code related to memory allocation and mapping pages ... **kalloc**, **growproc**, **allocvm**, **mappages** etc.)
- (ii) Understand the implementation of the **sbrk** system call and **mmap()** system call will follow a similar logic.
- (iii) Refer to Sheets 19, 25, 38 of [xv6 source code booklet](#) for the related system calls.
- (iv) Refer to Page 34 of [the xv6 book](#).

Source file `t_mmap.c` is provided for testing.

Sample usage

```
$ mmap
Process size before mmap 12288 Bytes
Process size after mmap 16384 Bytes
Accessing a address allocated by mmap 3F80
Pagefault occured at address eip 0x6a addr 0x3f80--kill proc
```

(c) being lazy pays (sometimes)!

Next, modify the page fault handler logic, to allocate memory on demand for the page (need to check if the page faulting address is a valid address!). Once a physical page is allocated and mapped for the virtual address being accessed, the handler returns and the access is re-attempted and should no longer result in a page fault.

Hints:

- Look at the arguments to the **cprintf** statements in [trap.c](#) to figure out how one can find the virtual address that caused the page fault.
- Use **PGROUNDDOWN(va)** to round the faulting virtual address down to the start of a page boundary.
- You may invoke **allocvm** (or write another similar function) in [vm.c](#) in order to allocate physical memory upon a page fault.
- You can add your page fault handler in `vm.c` and call it from `trap.c`.
- Check whether the page fault was actually due to a lazy allocated page or an actual page fault (For example - illegal memory access).

Note: it is important to call **switchvm** to update the **CR3** register and **TLB** every time you **change the page table of the process**. This update to the page table will enable the process to resume execution when you handle the page fault correctly.

A user program `t_lazy.c` is provided for testing.

Sample usage

```
VAS: 3 Pages
PAS: 3 Pages
-----Mapping 10 Pages-----
VAS: 6 Pages
PAS: 3 Pages
-----Accessing Page 0-----
VAS: 6 Pages
PAS: 4 Pages
-----Accessing Page 1-----
VAS: 6 Pages
PAS: 5 Pages
-----Accessing Page 2-----
VAS: 6 Pages
PAS: 6 Pages
```

Submission Instructions

- All submissions are via moodle. Name your submission as **<rollnumber_lab3>.tar.gz** (e.g 190050096_lab3.tar.gz)
- The tar should contain the xv6 source code files in the following directory structure: **<rollnumber_lab3>/xv6**
- The directory must contain the new programs **pingpong.c** and **cmd.c** which you will implement as a part of 1a and 1b respectively and the entire xv6 source code with necessary changes to it.
- Your code should be well commented and readable.
- **tar -czvf <rollnumber_lab3>.tar.gz <rollnumber_lab3>**

Deadline: Thursday 13th September 2024 5:00 PM via moodle.

More Exercises (Optional)

1. you got siblings?

Implement a system call `int get_sibling()`

to print the details of siblings of the calling process to the console and to return the number of siblings of the calling process. A sibling is all processes with the same parent process.

The output should be in the format of:

<pid> <process status>

<pid> <process status>

....

Sample usage

```
$ my_siblings 6 1 2 1 0 2 0
```

```
4 RUNNABLE
```

```
5 ZOMBIE
```

```
6 RUNNABLE
```

```
7 SLEEPING
```

```
8 ZOMBIE
```

```
9 SLEEPING
```

Sample user program **my_siblings.c** is provided. The program takes an integer **n**, followed by a combination of 0, 1 and 2 of length **n**, as command line arguments— 0/1/2 specify the process state of the **n** child processes. The (**n**+1) th child process executes the `get_sibling()` system call and displays the output.

Hint: You need to find the process ID of the calling process, and process ID of its parent and traverse all the PCBs and compare their parent PID with the parent of the calling process.

Sample usage

```
$ mysiblings 6 0 1 2 2 1 0
4 SLEEPING
5 RUNNABLE
6 ZOMBIE
7 ZOMBIE
8 RUNNABLE
9 SLEEPING
10 RUNNING
```

2. entrypoint 2.0

Implement a new type of **system_call_handler** which instead of handling TRAP NUMBER 64 handles trap number say 65. You should implement a new type of system call **fork2()** that uses a different trap number (**65**) instead of the commonly used trap number 64 (which corresponds to the traditional `int $0x64` instruction for making system calls in x86 assembly). Using a different trap number, such as 65 in this exercise, allows you to define and handle your custom system calls independently from the standard ones.

In order to achieve this you should first need to look at how system calls are handled in xv6. List of files you will need to refer to:

sysc.all.c syscall.h defs.h user.h proc.c sysproc.c proc.h trap.c trap.h usys.S

Note: First you need to write a trap handler and after that you can implement a new type of system call. (**TRAP NUMBER for SYSCALL is 64 but the actual system call number of system call like fork in xv6 is 1**).

Refer to sheet 32 33 and 34 of [xv6 source code booklet](#)

Hint: usys.s has the entry point from where `int n` is called. :)

One way to implement this is to change only the entry point to the trap handler and use the same underlying system call implementation for all system calls.

A simple test program **userfork2.c** is provided to test your implementation.

Sample usage:

```
$ fork2 ls
.          1 1 512
..         1 1 512
README    2 2 2286
hello.txt  2 3 24
pingpong.txt 2 4 357
cat        2 5 15800
head       2 6 16244
cmd        2 7 15104
pingpong   2 8 15624
echo       2 9 14680
forktest   2 10 9128
grep       2 11 18644
init       2 12 15300
kill       2 13 14764
ln         2 14 14664
ls         2 15 17232
mkdir      2 16 14788
rm         2 17 14772
sh         2 18 28828
stressfs   2 19 15696
usertests  2 20 63200
wc         2 21 16224
zombie     2 22 14348
```

3. status check

Implement a system call, with the following declaration **whatsthestatus(int pid)**, which returns the **parent pid**, prints the **name of the parent process** and the **current state of the process** given the pid of the process.

The function **cprintf** is used for printing in the kernel mode (refer to sheet 30 line 3026 of [xv6 source code booklet](#) for usage).

Please refer to the PCB structure defined on line 2336 sheet 23 of [xv6 source code booklet](#) and lines 10-14 in `proc.c` in the given source code to refer to **ptable** struct.

Refer to sheet 24 Line 2480 to understand how to iterate through the process table and sheet 23 Line 2334 to understand the process enum structure of [xv6 source code booklet](#) .

Refer to sheet 36 line 3631-3632 to refer **argint** usage of [xv6 source code booklet](#)

Note: **argint**, **argstr**, **argptr** are helper functions for handling system calls arguments.

A simple test program **status.c** is also provided to test your implementation.

Input should be in the format **status 3 0 1 2** when “3” denotes the number of children to fork and 0 signifies **Sleeping**, 1 signifies **Runnable** and 2 signifies **Zombie** states.

Output should be in the format **<pid> <status> <ppid> <parent_name>** where **pid** represents the pid of the child process for which status needs to be checked, **ppid** represents the parent pid and **parent_name** represents the name of the parent.

Sample usage

```
$ test-whatsthestatus 3 0 1 2
0 4 SLEEPING 3 test-whatsthest
0 5 RUNNABLE 3 test-whatsthest
0 7 ZOMBIE 3 test-whatsthest
```