

Day 2

Task 1: Implementing the spawn system call

spawn — one call, many processes!

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

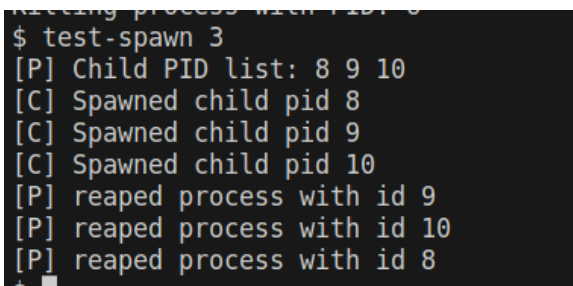
- The system call must return 0 to the child processes and number of children created to the parent process.
- Additionally, the **pids** array should contain the pids of the spawned children after the **spawn** system call. The parent should gracefully reap all the child processes which are present 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 2 : Memory handling with xv6

Following is a partial list of important files for the task:

syscall.c, **syscall.h**, **sysproc.c**, **user.h**, **usys.S**, **vm.c**, **proc.c**, **trap.c**, **defs.h**, **mmu.h**, **memlayout.h**, **kalloc.c**

- **sysproc.c**, **syscall.c**, **syscall.h**, **user.h**, **usys.S** link user system calls to system call implementation code in the kernel.
- **mmu.h**, **memlayout.h** and **defs.h** are header files with various useful definitions pertaining to memory management.

- **vm.c** contains most of the logic for memory management in the xv6 kernel, and **proc.c** contains process-related system call implementations.
- **trap.c** contains trap handling code for all traps including memory access related exceptions (page faults).

2.1. Is the virtual address space real?

Write a system call **getvasize()** that returns the size of the virtual memory used by a process. Specifically, the system call should have the following interface:

```
int getvasize(int pid);    // pid is argument to the call and amount of virtual memory
                           // used by the process as 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](#).

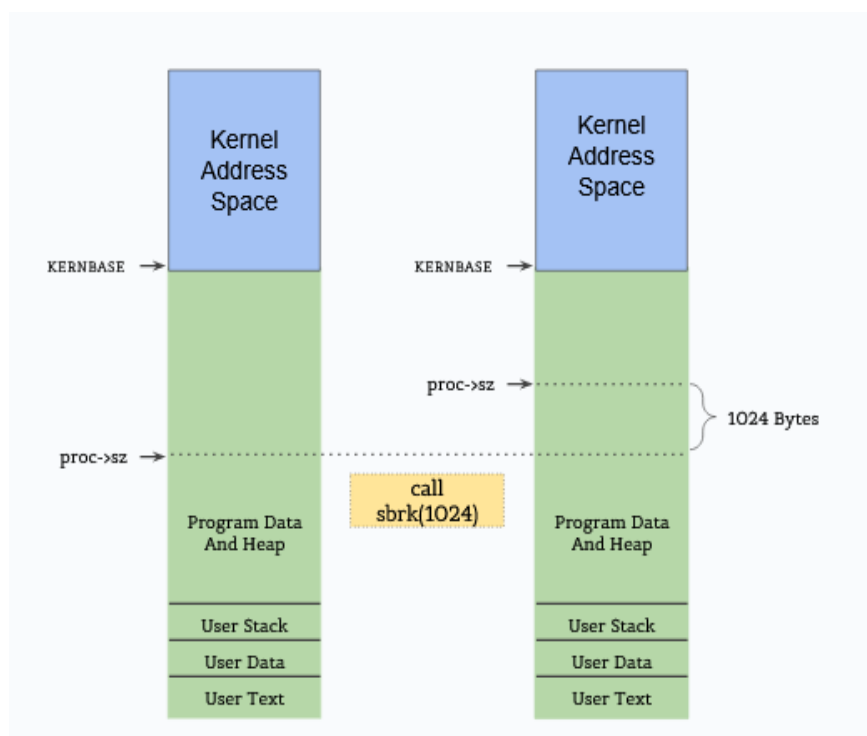


Figure 1. After calling **sbrk(1024)** to increase the process size by 1024.

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: Show memory image of process in XV6. User Stack in XV6 is 1 Page Follow by heap 2 upto KERNBASE.

A sample program **t_getvsize.c** is available for testing.

```
$ t_getvsize
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
$
```

2.2. 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.

Figure 2: Page table layout

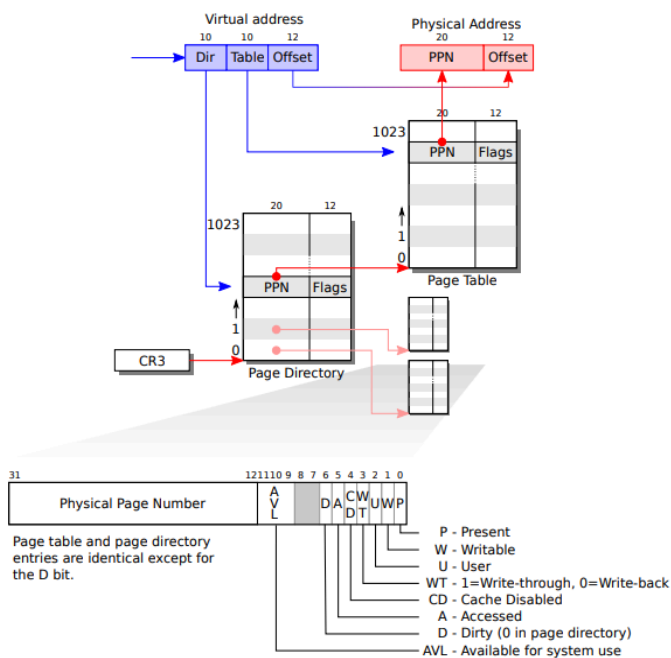
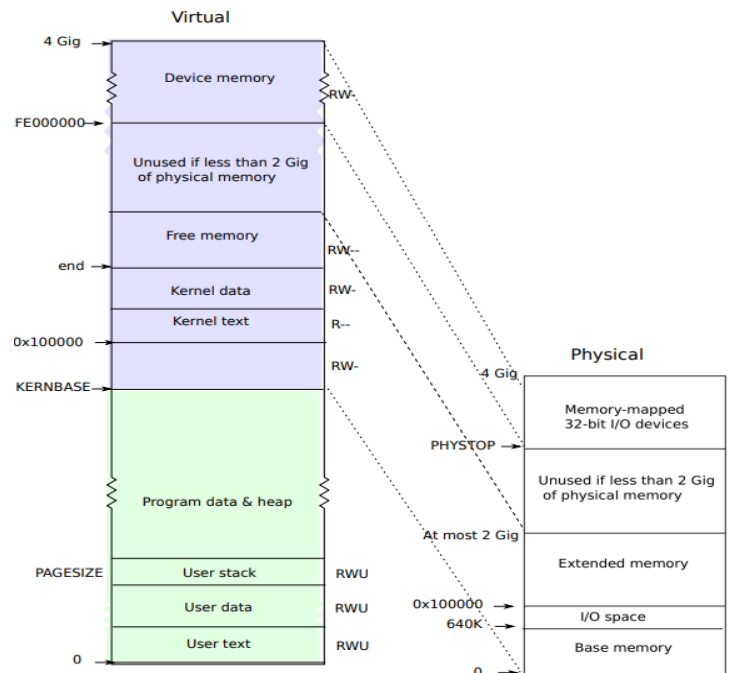


Figure 3: Virtual to physical address mapping



- (iii) Refer to discussion on Page 29-32 of [the xv6 book](#).

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 top 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_va_to_pa1.c** and **t_va_to_pa2.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
```

Q. Can you see what is interesting in the two outputs?

2.3. 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 by using the **walkpgdir** function and consider only those entries which indicate mapping 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
```

2.4. no escape from reality!

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

int getpaside(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_getpaside.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 of Task 4 .

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: Page fault handling

Step 1: faulting page, page faulting, who is handling?

The default xv6 distribution does not handle the page fault trap explicitly.

Extended implementation of 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

```
Physical address on fault in parent process
$ 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.
-

More Exercises (Optional)

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: **syscall.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
```