Day 3

Task 1: Lazy page allocation

Step 1: mmap()

Implement a simple 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!.

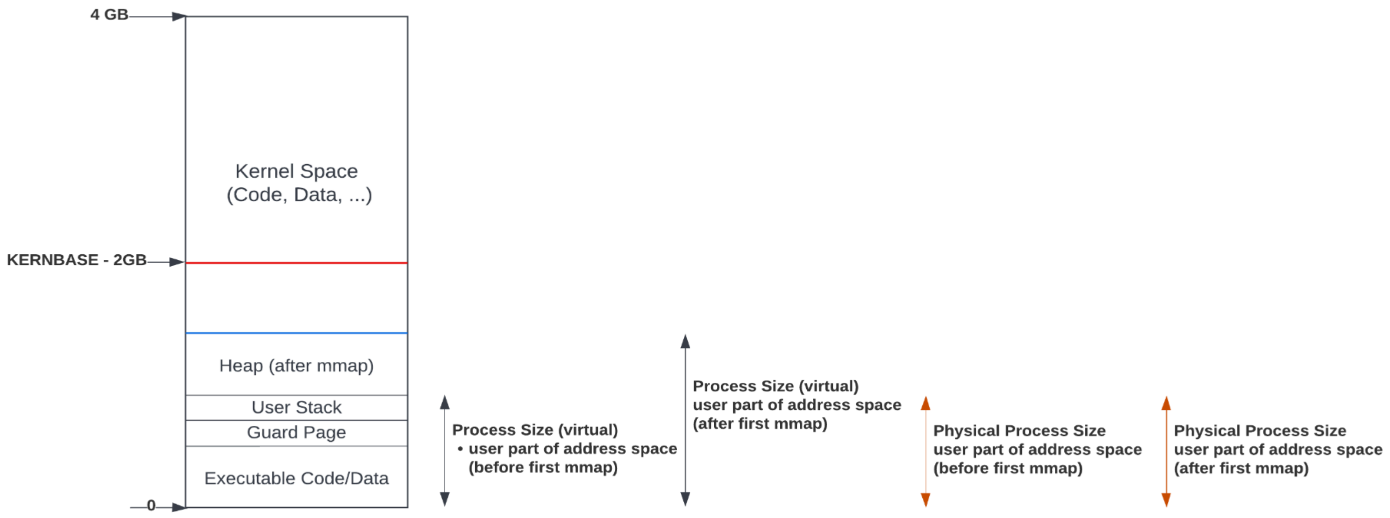

4 GB
Kernel Space (Code, Data, ...)
KERNBASE - 2GB
Heap (after mmap)
User Stack
Guard Page
Executable Code/Data
0
Process Size (virtual) • user part of address space (before first mmap)
Process Size (virtual) user part of address space (after first mmap)
Physical Process Size user part of address space (before first mmap)
Physical Process Size user part of address space (after first mmap)

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

The figure shows the working of mmap system call. when user call say mmap(1024) the virtual address space of the process increases however the physical address space remains the same.

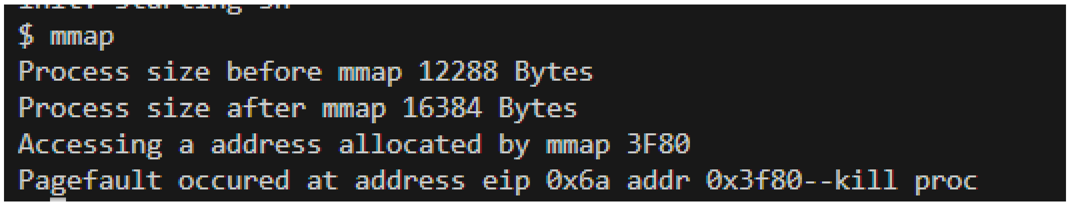
Assume that the <u>number of bytes is a positive number and is a multiple of the page size.</u> The system call should return a value of 0 if any invalid inputs are provided. In the valid case, the system call should expand the process's size by the specified number of bytes, and return the starting 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) … this Step 3.

Hints:
(i) **mmap**() system call is similar to **sbrk**() (with code related to memory allocation and mapping pages … kalloc, **growproc**, allocuvm, 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
```

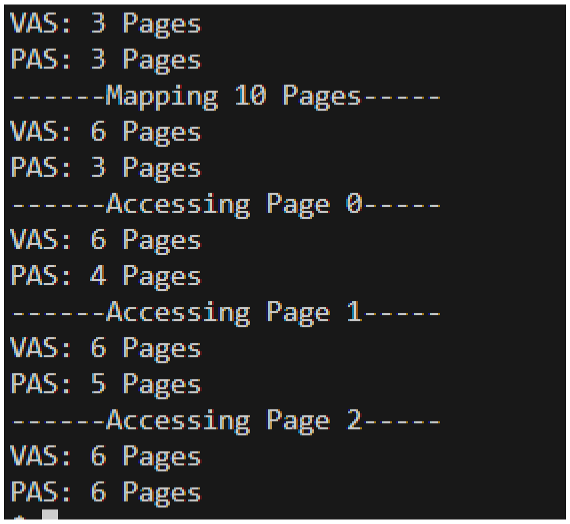
Step 2: the big reveal

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 not 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 **allocuvm** (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 switchuvm 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
```

---

to be lazy is human, to share is humane!
note: solving this question is optional, rest is not optional.

xv6 does not support shared memory by default in this Part. We would like you to implement a mechanism for Shared Memory. For this part we want you to implement Shared Memory support for xv6. You will need to implement 4 System Calls namely

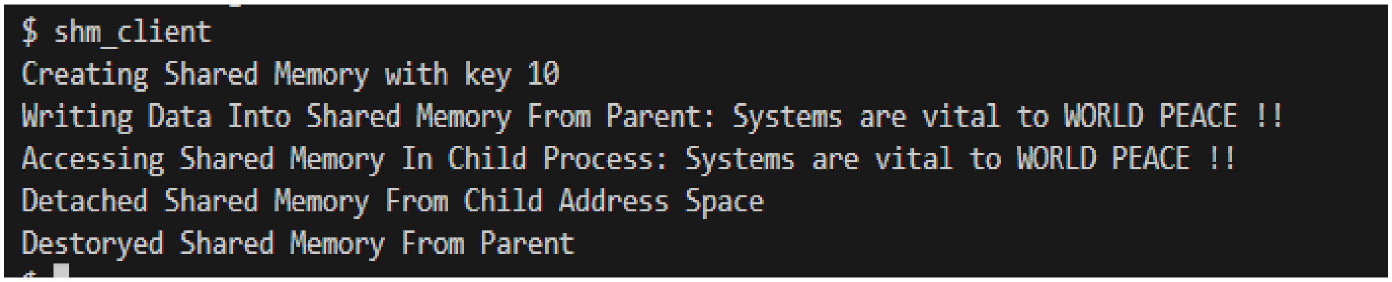
uint attach_shm(uint key); → Allocated one page shared memory identified by KEY to process address space.

uint create_shm(uint key); → Attaches Shared memory identified by KEY to process add space.

int detach_shm(uint key); → Deattaches Shared memory from process add space.

int destroy_shm(uint key); → Destroy Shared memory identified by KEY

The program in **t_shm_client.c** is provided for testing.



```
$ shm_client
Creating Shared Memory with key 10
Writing Data Into Shared Memory From Parent: Systems are vital to WORLD PEACE !!
Accessing Shared Memory In Child Process: Systems are vital to WORLD PEACE !!
Detached Shared Memory From Child Address Space
Destoryed Shared Memory From Parent
```

<u>**Hints:**</u>

(i) need to maintain some form of table in the kernel in order to keep track of all the shared memory identified by key and when you do call attach this table can be used to give you PA associated with the shared memory identified by key.

(ii) need to call **mappages**() from trap.c in order to map the physical address in shared memory table to virtual address for the shared memory in attach_shm() .In order to do this, you'll need to delete the static in the declaration of mappages() in vm.c, and you'll need to declare mappages() in the trap.c. Add this declaration to the trap.c before any call to **mappages**(): int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);

Optional Task:

entrypoint 2.0

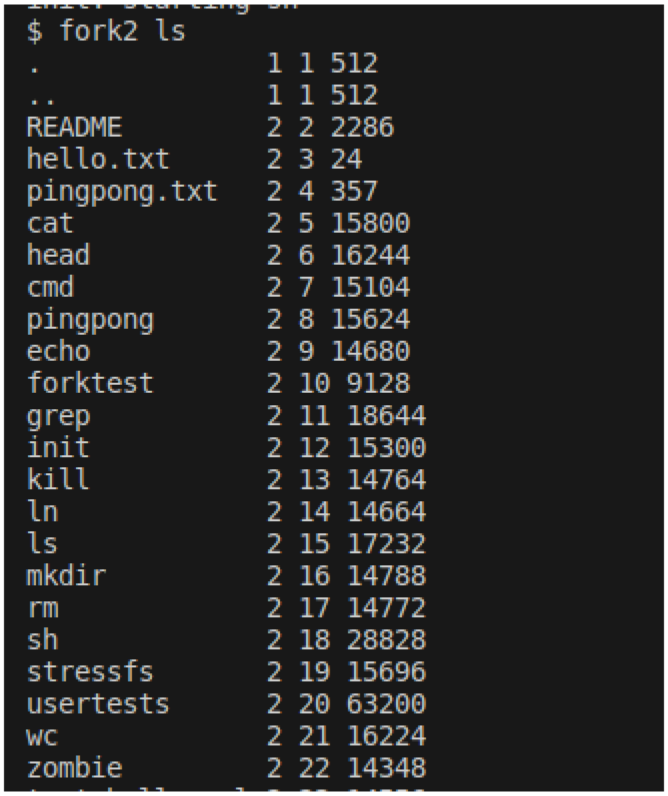
Implement a new type of **system_call_handler** which instead of handling TRAP NUMBER 64 handles trap number say 65. You should implement a new type of system call **fork2()** that uses a different trap number (**65**) instead of the commonly used trap number 64 (which corresponds to the traditional `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:



```
init: starting sh
$ 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
```

Task 2: Blast from the past

Step 1: Implement smalloc system call

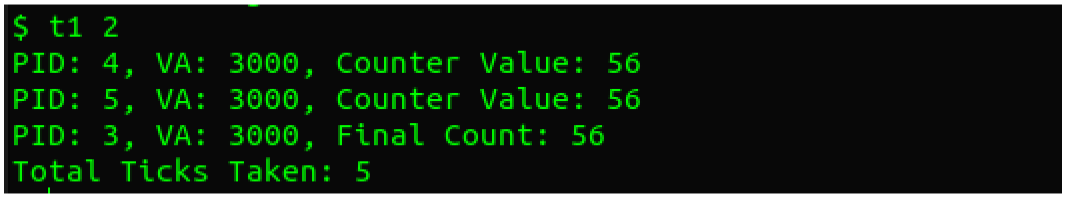
Implement a system call called **smalloc** with declaration **`char* smalloc(void)`** which increases the size of the process (virtual) space by **one** page (4096 bytes), and maps it to a physical page. The virtual address of the process if to be kept page aligned.

The characteristic of this system call is that while every call changes the process (virtual) address space usage (size) the **same** physical page is used for mapping the virtual address range (in effect providing a shared memory region). Note that **smalloc** is a system call and can be invoked from across processes or multiple times in a single process and sets up multiple instances of execution with a shared physical page.

Implementation notes:

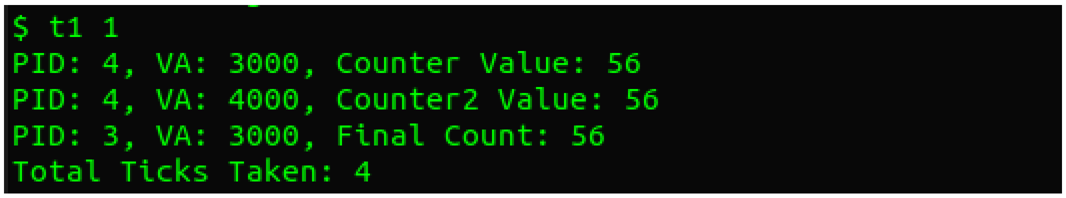
- Needs logic to store and reuse a reserved physical page for multiple mappings.
- Needs careful handling of freeing memory mappings to a shared page. By default xv6 has no support for page sharing and frees pages whenever a process exits.
  Support needs to be added to make sure that pages are not freed on page table cleanup or on any other action when the physical page may still be shared via other mappings.
- xv6 functions of interest —
  `growproc, allocuvm, deallocuvm, freevm, mappages, kalloc, …`

Uncomment the lines corresponding to counter1 and final, compile and execute the file **t1.c** to test your implementation.
Usage: **`t1 <number of processes to fork>`**
Sampe usage:



```
$ t1 2
PID: 4, VA: 3000, Counter Value: 56
PID: 5, VA: 3000, Counter Value: 56
PID: 3, VA: 3000, Final Count: 56
Total Ticks Taken: 5
```

Now, uncomment the two lines corresponding to counter2 present in t1.c to observe that different smalloc() calls return the same shared area even though the virtual addresses differ



```
$ t1 1
PID: 4, VA: 3000, Counter Value: 56
PID: 4, VA: 4000, Counter2 Value: 56
PID: 3, VA: 3000, Final Count: 56
Total Ticks Taken: 4
```

p.s.: PID values, Total Ticks taken need not match.

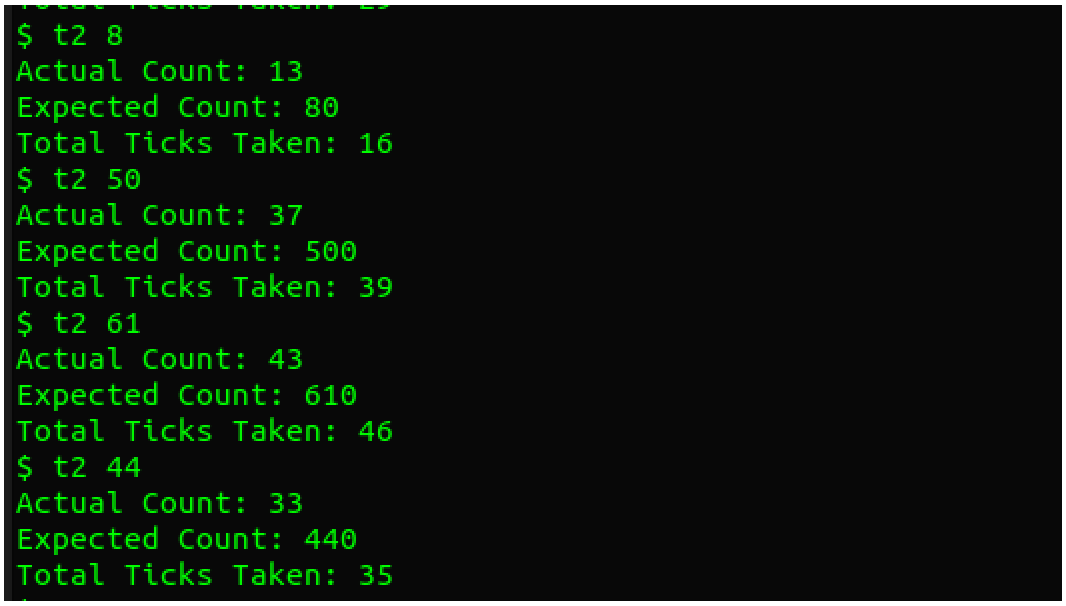
Step 2: race to the bottom

This task depends on and uses the **smalloc** system call.

Understand the program specified in the file **t2.c** and execute it in xv6 to observe its outputs. Uncomment all the lines present in t2.c before compiling

Usage: `t2 <number of processes>`

Sample usage



```
$ t2 8
Actual Count: 13
Expected Count: 80
Total Ticks Taken: 16
$ t2 50
Actual Count: 37
Expected Count: 500
Total Ticks Taken: 39
$ t2 61
Actual Count: 43
Expected Count: 610
Total Ticks Taken: 46
$ t2 44
Actual Count: 33
Expected Count: 440
Total Ticks Taken: 35
```

Do we have a race condition yet?

Task 3: Context switching mechanism

Step 1: count switch-in and switch-out in xv6

xv6 uses a per−CPU process scheduler. Each CPU calls a **`void scheduler(void)`** after setting itself up. The scheduler keeps looping infinitely, doing the following:

- ➔ choose a process to run
- ➔ **`swtch`** to start running that process
- ➔ Eventually, that process transfers control via `swtch` back to the scheduler.

Refer to sheet 27 to understand the existing scheduler function (present in `proc.c`) in xv6 and get familiar with it.

The functioning of the `swtch` system call is as follows:
`void swtch(struct context **old, struct context *new);`
Save the current registers on the stack (populating) struct context, and save its address in *old (this is the old context).
Switch stacks to newcontext (new→esp) and pop previously saved registers.
Refer to implementation in **swtch.S**

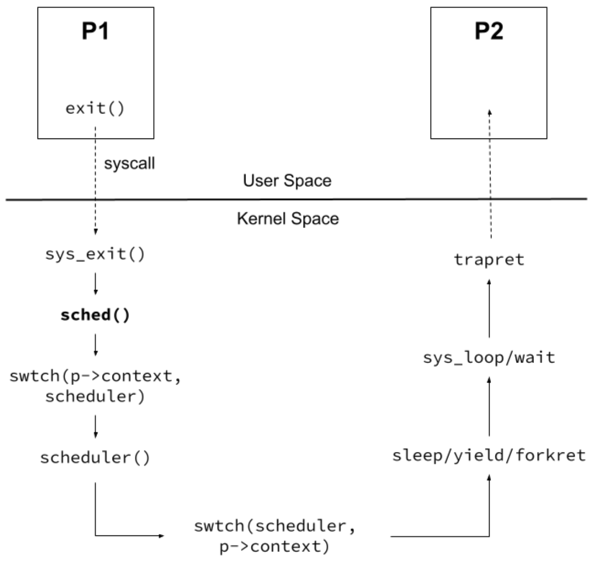


Figure 1: Depiction of system call flow in process scheduling

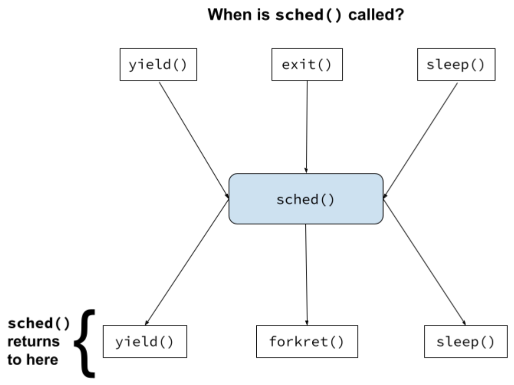


Figure 2: Locations in xv6 code where sched() is invoked and where sched() returns to.

Refer to `swtch.S` in the xv6 source code to familiarize yourself with the underlying assembly code for saving and loading registers while context switching.
Figure 1 describes the sequence of **scheduler**, **sched** and **swtch** and Figure 2 shows the location in xv6

code where it is called from where it returns to when rescheduled.

Implement a system called **`cscount(int pid)`** to count the number of context switches for a process with a given pid, i.e. you need to count both the number of switch-in and switch-out of that particular process.

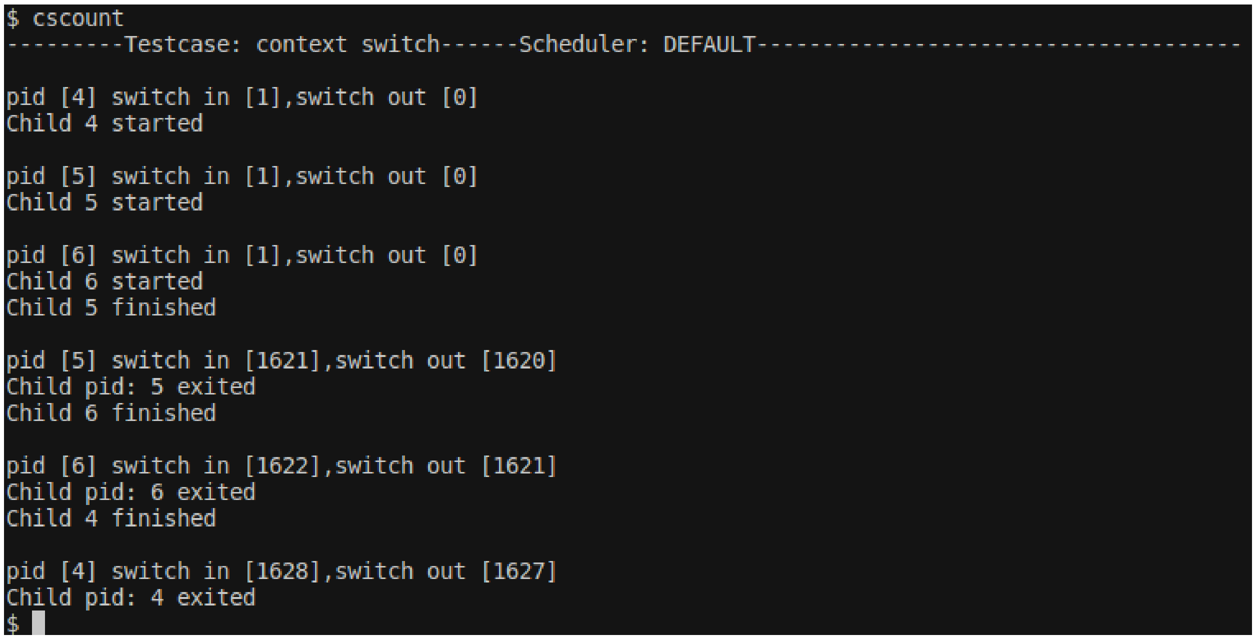
Hint: Can add fields in proc struct to store the per process context switch in and out values. Understand the logic of **sched** in proc.c and when it is called. Note that the **scheduler** picks up a user space process to run on the CPU whereas **sched** switches back to the scheduler again once the process is done. **sched** is called after a timer interrupt when a process becomes a zombie or when a process goes to a blocked state.

NOTE: The user program provided `cscount.c` has a simple implementation to check the `cscount()` system call.
The given user program forks child processes and calls the cscount() just before the child starts executing, and calls the system call cscount() again just before the child exits.

The output format is pid [**process pid**] switch in [**context switch in count**], switch out [**context switch out count].**

Your output should give a switch in count = 1 and switch out count = 0 just before the process starts executing since this is the first time the process is getting the share of the CPU. The switch in and out count at the end can be any number depending on the time taken by the process to execute and the number of context switches it went through.

Sample usage



```
$ cscount
---------Testcase: context switch------Scheduler: DEFAULT------------------------------------

pid [4] switch in [1],switch out [0]
Child 4 started

pid [5] switch in [1],switch out [0]
Child 5 started

pid [6] switch in [1],switch out [0]
Child 6 started
Child 5 finished

pid [5] switch in [1621],switch out [1620]
Child pid: 5 exited
Child 6 finished

pid [6] switch in [1622],switch out [1621]
Child pid: 6 exited
Child 4 finished

pid [4] switch in [1628],switch out [1627]
Child pid: 4 exited
$
```

Step 2: Waiting for execution statistics

Implement a new system call `wait2` similar to wait, but with more functionalities, in order to check the performances of xv6 scheduling algorithms. Specifically, it will have the following interface:

```
int wait2(int *wtime, int *runtime);
```

The call takes two arguments, pointers to variables that denote the amount of time the process spent waiting for the CPU and the time spent executing on the CPU. It waits for a child process to exit and fills the waiting time and run time (both are in terms of ticks counted in the trap handler. Note that xv6 is configured to generate 200 ticks per second for a 2GHz CPU, which corresponds to a tick every 5ms. You may however report just the tick count.) in `wtime` and `runtime` buffer, respectively, for the process that is exiting and return its **pid**. Return **-1** if the calling process has no children.

The **waiting time** of a process is defined as the time spent by the process in the RUNNABLE state (ready to run and waiting for CPU), and **run time** is the time spent by the process on the CPU (time is in ticks).

NOTE: The user programs provided `task1b.c` have a simple implementation to check the wait2() system call.

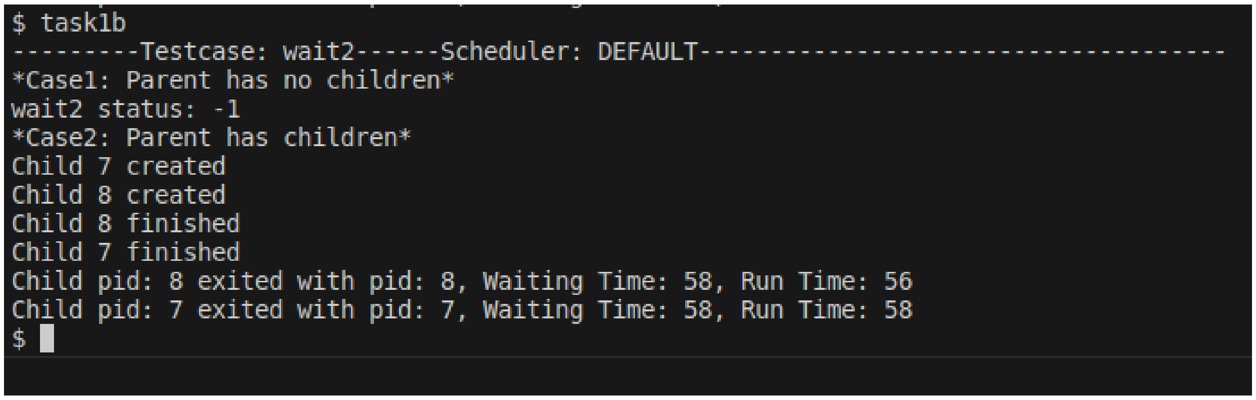
The wait2 call will be useful for understanding how a scheduling policy affects the times of every Process.

Hints:

- Logging of durations/timings will need tracking down all events where the state of the process changes between WAITING, RUNNABLE, RUNNING etc. and appropriate duration updates via variables in the PCB entry for the process.
- An implementation of the `wait` system call exists in xv6 and can be the starting point for the `wait2` implementation — a copy of the code of wait is a good starter for the implementation of `wait2`.
- You can initialize `wtime` and `runtime` in the `allocproc` function in the `proc.c` file.

Note: It is important to keep in mind that the process table struct `ptable` is protected by a lock.
You must acquire the lock before accessing this structure for reading or writing and must release the lock after you are done.

Sample usage



```
$ task1b
---------Testcase: wait2------Scheduler: DEFAULT------------------------------------
*Case1: Parent has no children*
wait2 status: -1
*Case2: Parent has children*
Child 7 created
Child 8 created
Child 8 finished
Child 7 finished
Child pid: 8 exited with pid: 8, Waiting Time: 58, Run Time: 56
Child pid: 7 exited with pid: 7, Waiting Time: 58, Run Time: 58
$
```

Task 4: Process scheduling and extensions

Step 1: *actions express priorities …* or is it the other way round?

A priority-based scheduler selects the process with the highest priority for execution. In case two or more processes have the same priority, we choose them in a round-robin fashion. The priority of a process can be in the range [0,100]. The **smaller value will represent a higher priority**. Set the default priority of a process as 60. To change the default priority add a new system call `set_priority` which can change the priority of a process.

`int set_priority (int pid, int priority)`

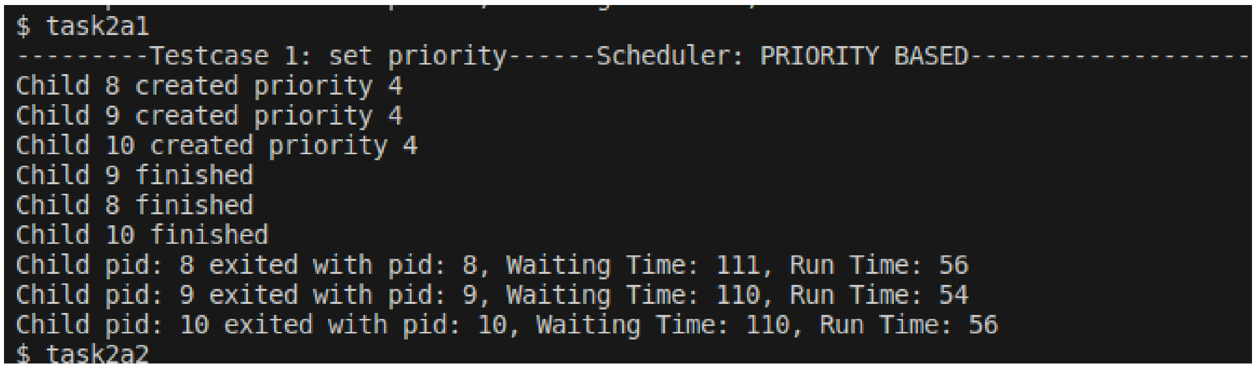
The system call should set the priority of a process with a given pid and return the pid of the process. The scheduler should then schedule the process based on the set priority.

Hint: You also need to slightly modify the `scheduler()` function in `proc.c` to change the logic to choose the process having higher priority to schedule instead of the one used by the default round robin in xv6.

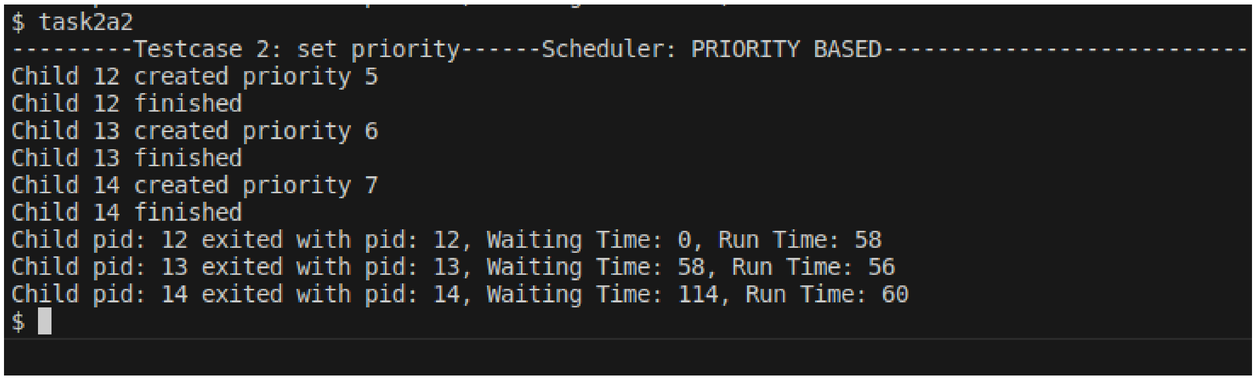
The user programs provided `task4a.c` have a simple implementation to check the `set_priority()` system call.

Sample usage

Note: If multiple processes have the same priority, they will be executed in the default round-robin fashion (first sample output) and if processes have different priorities (second sample output), the one with a lower priority number (i.e. higher priority) will execute before the ones with a higher priority number (i.e. lower priority).
This effect is visible in the increased waiting time of lower priority processes.



```
$ task2a1
---------Testcase 1: set priority------Scheduler: PRIORITY BASED------------------
Child 8 created priority 4
Child 9 created priority 4
Child 10 created priority 4
Child 9 finished
Child 8 finished
Child 10 finished
Child pid: 8 exited with pid: 8, Waiting Time: 111, Run Time: 56
Child pid: 9 exited with pid: 9, Waiting Time: 110, Run Time: 54
Child pid: 10 exited with pid: 10, Waiting Time: 110, Run Time: 56
$ task2a2
```



```
$ task2a2
---------Testcase 2: set priority------Scheduler: PRIORITY BASED--------------------------
Child 12 created priority 5
Child 12 finished
Child 13 created priority 6
Child 13 finished
Child 14 created priority 7
Child 14 finished
Child pid: 12 exited with pid: 12, Waiting Time: 0, Run Time: 58
Child pid: 13 exited with pid: 13, Waiting Time: 58, Run Time: 56
Child pid: 14 exited with pid: 14, Waiting Time: 114, Run Time: 60
$ 
```

Step 2: The big slice

Implement the following system call **`int set_quanta(int pid, int quanta),`** which sets the time slice quanta of a process with the given pid. This system call will allow you to give more time to a process overriding the **default slice** of 1 quanta/tick implemented in xv6.

You need to add a specific field to the process structure (e.g. current slice, extra_slice) at `proc.h` to hold the current time slice value. You should create the syscall to set quanta and modify the scheduler function to reset current_slice to extra_slice at process wakeup.

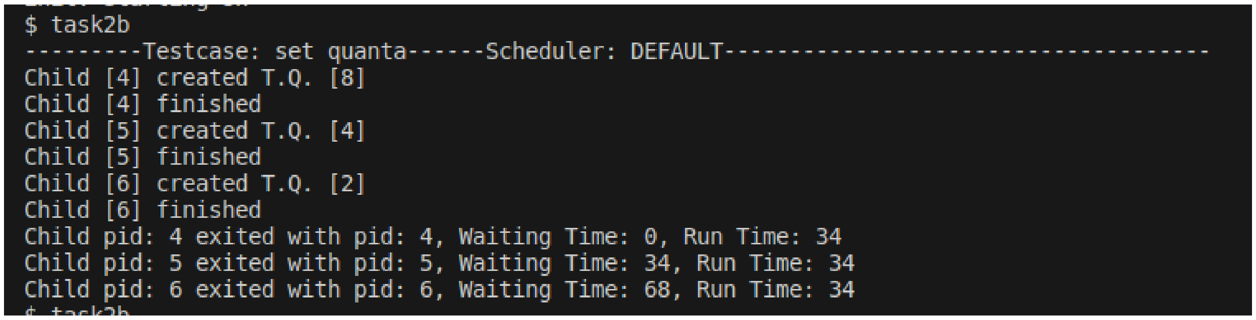
You also need to modify `trap.c` to handle the time slice logic when timer interrupts come for a process to give up the CPU on the clock tick. Look at the case of what happens when
`T_IRQ0 + IRQ_TIMER` interrupt occurs.

Note:. Change the modification in `scheduler()` in proc.c made for the previous task back to the default scheduling logic and add a line to update the process's current slice value.

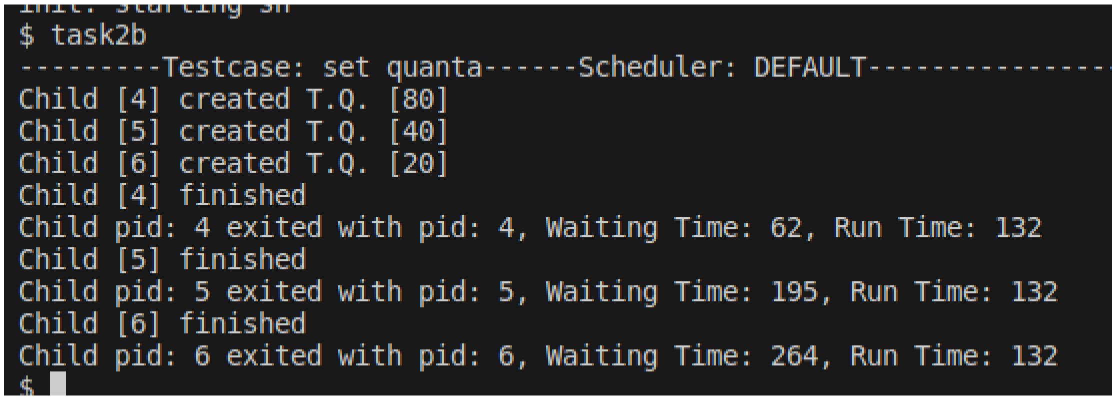
The user programs provided `task4b.c` have a simple implementation to check the `set_quanta()` system call.

Note: The process with a higher quanta should have a lesser waiting time than others since it gets more time (opportunity) to execute at a time (and T.Q. in output implies "Time Quanta")

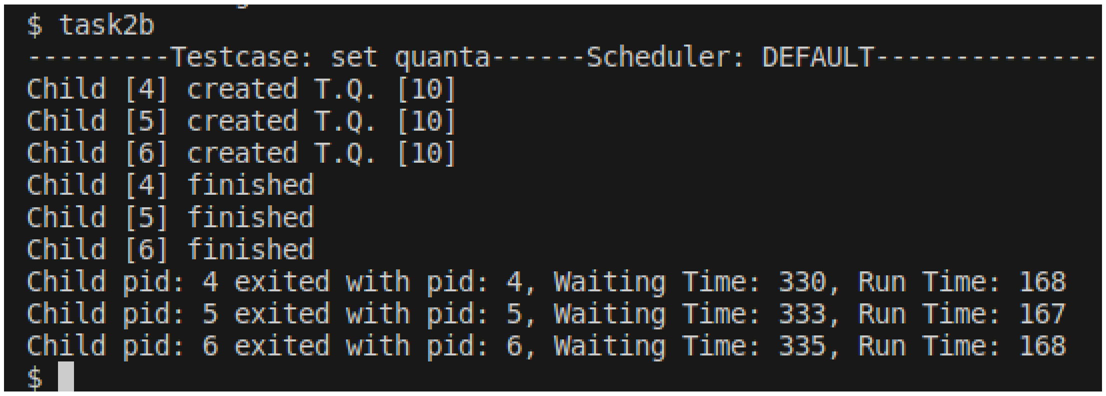
Sample usage:
The process with a higher time quanta (pid = 4) finishes before every one as it has the largest time quanta (T.Q. = 8) and has a waiting time 0 since it utilizes its full quanta to complete its execution. Similar is the case process with pid = 5 with respect to the process with pid = 6. ***So you need to make sure that the process with a higher time quanta has a lesser waiting time than those with lesser time quanta.***



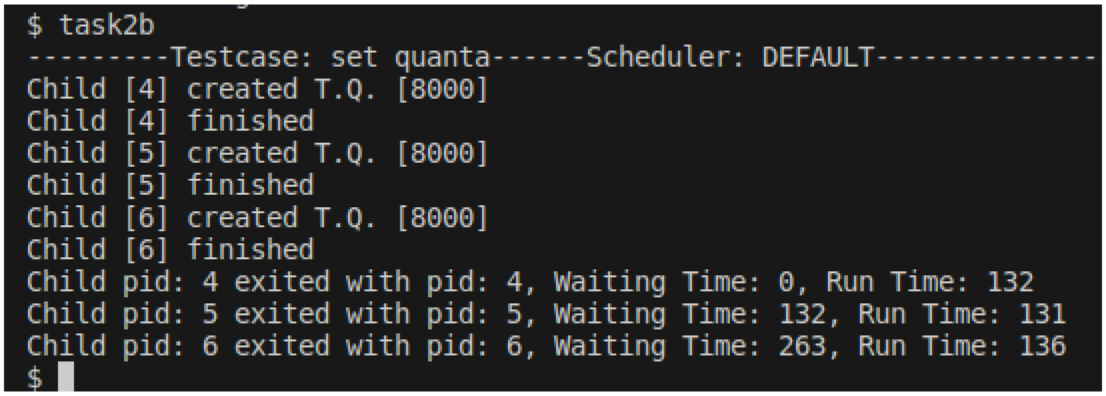
```
$ task2b
---------Testcase: set quanta------Scheduler: DEFAULT------------------------------------
Child [4] created T.Q. [8]
Child [4] finished
Child [5] created T.Q. [4]
Child [5] finished
Child [6] created T.Q. [2]
Child [6] finished
Child pid: 4 exited with pid: 4, Waiting Time: 0, Run Time: 34
Child pid: 5 exited with pid: 5, Waiting Time: 34, Run Time: 34
Child pid: 6 exited with pid: 6, Waiting Time: 68, Run Time: 34
```



```
$ task2b
---------Testcase: set quanta------Scheduler: DEFAULT---------------
Child [4] created T.Q. [80]
Child [5] created T.Q. [40]
Child [6] created T.Q. [20]
Child [4] finished
Child pid: 4 exited with pid: 4, Waiting Time: 62, Run Time: 132
Child [5] finished
Child pid: 5 exited with pid: 5, Waiting Time: 195, Run Time: 132
Child [6] finished
Child pid: 6 exited with pid: 6, Waiting Time: 264, Run Time: 132
$
```



```
$ task2b
---------Testcase: set quanta------Scheduler: DEFAULT--------------
Child [4] created T.Q. [10]
Child [5] created T.Q. [10]
Child [6] created T.Q. [10]
Child [4] finished
Child [5] finished
Child [6] finished
Child pid: 4 exited with pid: 4, Waiting Time: 330, Run Time: 168
Child pid: 5 exited with pid: 5, Waiting Time: 333, Run Time: 167
Child pid: 6 exited with pid: 6, Waiting Time: 335, Run Time: 168
$
```



```
$ task2b
---------Testcase: set quanta------Scheduler: DEFAULT--------------
Child [4] created T.Q. [8000]
Child [4] finished
Child [5] created T.Q. [8000]
Child [5] finished
Child [6] created T.Q. [8000]
Child [6] finished
Child pid: 4 exited with pid: 4, Waiting Time: 0, Run Time: 132
Child pid: 5 exited with pid: 5, Waiting Time: 132, Run Time: 131
Child pid: 6 exited with pid: 6, Waiting Time: 263, Run Time: 136
$
```