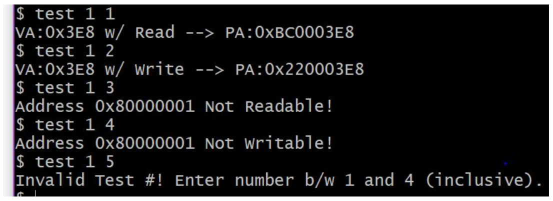
Testing the Syscall Implementations: Hypotheses & Test Data

How does the myV2p() implementation work?
→In order to get the physical address from a 32 bit virtual address it must first be broken up into parts. This is done in mmu.h using the PDX(va) , PTX(va) and PTE_FLAGS(va) macros. PDX(va) gives you the top 10 bits or the page directory index, PTX(va) gives the page table index. PTE_FLAGS(va) gives the offset of the physical address. The page directory for the process is obtained from the proc struct of the process. PDX(va) is then used as an index into this page table directory to obtain the appropriate page table entry (pde). The pte is then used to find the base address of the process page table by using the P2V() macro on the upper 20 bits (address) in the pte. The page table entry(pte) is found by using PTX(va) as an index into the page table. In this entry, the upper 20 bits are the physical address, and the lower 12 bits are the flags, which can be checked for the state of that page (user/supervisor, writable/non-writable, etc.). We use the second argument of myV2p, the w/r value, to check whether the page has the required permissions by seeing the appropriate flag bits. If it does have access, the physical address (PPN from the upper 20 bits of the page table entry OR'd with the page offset -- lower 12 bits of VA) is returned. On any error, -1 os returned.

How does the hasPages() implementation work?
→In order to identify what type pages the given pid has, hasPages() takes the pid and goes through each page table entry that is present. If the page table entry is present, it determines the number of different kinds of pages using the 12 flag bits. Additionally, two variables were added to the proc struct to store the top addresses of the code & stack segments (which were captured in the exec() function). These were used to determine the number of pages in each stack based on the memory layout given in the xv6 manual.

Test Results →

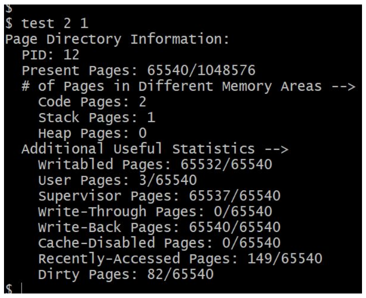
myV2p
Test Cases → #1: VA in User Space w/ Read Operation
#2: VA in User Space w/ Write Operation
#3: VA in Kernel Space w/ Read Operation
#4: VA in Kernel Space w/ Write Operation
For this system call, we would expect VAs in User Space (less than 0x80000000) to be both Readable & Writable (and therefore return valid PAs), but VAs in Kernel Space (greater than 0x80000000) to be neither (and return -1 -- thereby printing an error message). This was exactly what was observed in the test cases (see below):


```
$ test 1 1
VA:0x3E8 w/ Read --> PA:0xBC0003E8
$ test 1 2
VA:0x3E8 w/ Write --> PA:0x220003E8
$ test 1 3
Address 0x80000001 Not Readable!
$ test 1 4
Address 0x80000001 Not Writable!
$ test 1 5
Invalid Test #! Enter number b/w 1 and 4 (inclusive).
```

Note: Invalid test numbers display an error message.

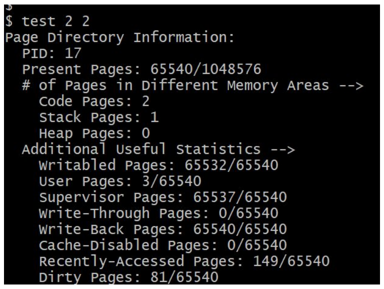
hasPages

Test Case #1 → Here, we only allocate a small local variable of size 10 bytes. This would be stored in the stack (of which there is only 1 page in xv6). This is seen in the image below (# of Stack Pages = 1):



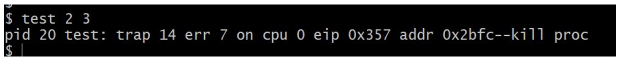
```
$
$ test 2 1
Page Directory Information:
  PID: 12
  Present Pages: 65540/1048576
  # of Pages in Different Memory Areas -->
    Code Pages: 2
    Stack Pages: 1
    Heap Pages: 0
  Additional Useful Statistics -->
    Writabled Pages: 65532/65540
    User Pages: 3/65540
    Supervisor Pages: 65537/65540
    Write-Through Pages: 0/65540
    Write-Back Pages: 65540/65540
    Cache-Disabled Pages: 0/65540
    Recently-Accessed Pages: 149/65540
    Dirty Pages: 82/65540
$
```

Test Case #2 → This time, we allocate a larger local variable of size 1000 bytes (still within the bounds of the stack). This would also show a single stack page as seen in the image below (# of Stack Pages = 1):



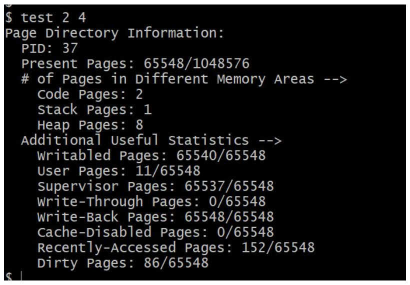
```
$
$ test 2 2
Page Directory Information:
  PID: 17
  Present Pages: 65540/1048576
  # of Pages in Different Memory Areas -->
    Code Pages: 2
    Stack Pages: 1
    Heap Pages: 0
  Additional Useful Statistics -->
    Writabled Pages: 65532/65540
    User Pages: 3/65540
    Supervisor Pages: 65537/65540
    Write-Through Pages: 0/65540
    Write-Back Pages: 65540/65540
    Cache-Disabled Pages: 0/65540
    Recently-Accessed Pages: 149/65540
    Dirty Pages: 81/65540
```

Test Case #3 → Now, we try to allocate a local variable larger than 1 page (500 bytes > 4096). Since there is only 1 stack page in xv6 (refer to the xv6 manual), the shell will throw an exception (see below).



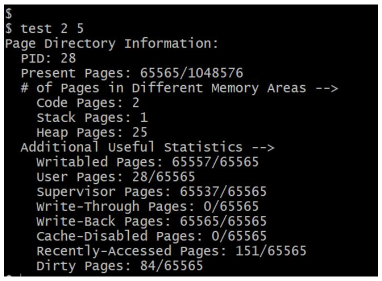
```
$
$ test 2 3
pid 20 test: trap 14 err 7 on cpu 0 eip 0x357 addr 0x2bfc--kill proc
$
```

Test Case #4 → Now we move on to the heap. So far, there were no heap pages allocated (so it was 0). Therefore, we allocate a small local variable of size 10 bytes as well as a dynamic variable of the same size using a call to malloc. Therefore, we would have 1 stack page, and some number of heap pages. This can be verified in the image below (# of heap pages = 8):



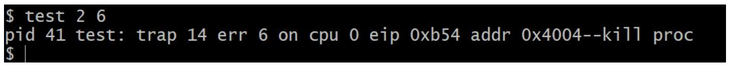
```
$
$ test 2 4
Page Directory Information:
  PID: 37
  Present Pages: 65548/1048576
  # of Pages in Different Memory Areas -->
    Code Pages: 2
    Stack Pages: 1
    Heap Pages: 8
  Additional Useful Statistics -->
    Writabled Pages: 65540/65548
    User Pages: 11/65548
    Supervisor Pages: 65537/65548
    Write-Through Pages: 0/65548
    Write-Back Pages: 65548/65548
    Cache-Disabled Pages: 0/65548
    Recently-Accessed Pages: 152/65548
    Dirty Pages: 86/65548
$
```

Test Case #5 → If we allocate a larger dynamic variable (1000 bytes), more heap pages will be needed (see below: # of heap pages = 25).



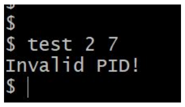
```
$
$ test 2 5
Page Directory Information:
  PID: 28
  Present Pages: 65565/1048576
  # of Pages in Different Memory Areas -->
    Code Pages: 2
    Stack Pages: 1
    Heap Pages: 25
  Additional Useful Statistics -->
    Writabled Pages: 65557/65565
    User Pages: 28/65565
    Supervisor Pages: 65537/65565
    Write-Through Pages: 0/65565
    Write-Back Pages: 65565/65565
    Cache-Disabled Pages: 0/65565
    Recently-Accessed Pages: 151/65565
    Dirty Pages: 84/65565
```

Test Case #6 → Finally, we try to allocate a dynamic variable larger than the heap size (greater than 0x80000000 bytes). Yet again, the shell raises an exception (see below).



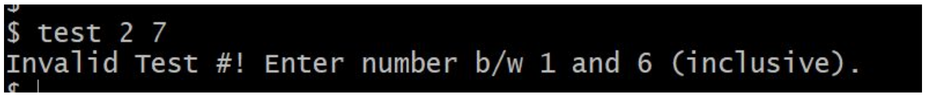
```
$ test 2 6
pid 41 test: trap 14 err 6 on cpu 0 eip 0xb54 addr 0x4004--kill proc
$ |
```

Test Case #7 → An invalid PID would cause hasPages() to return -1 and the test function will print an error message as shown below:



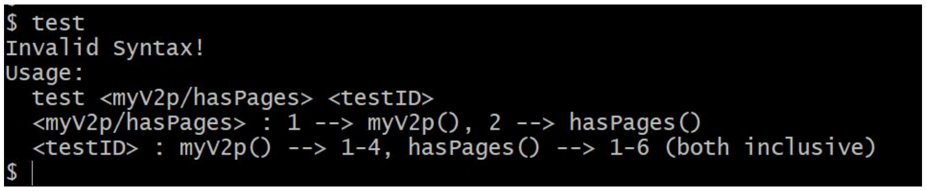
```
$
$ test 2 7
Invalid PID!
$ |
```

Also, invalid test numbers will display an error message:



```
$ test 2 7
Invalid Test #! Enter number b/w 1 and 6 (inclusive).
```

Finally, providing invalid arguments to the test function displays the following usage msg:



```
$ test
Invalid Syntax!
Usage:
  test <myV2p/hasPages> <testID>
  <myV2p/hasPages> : 1 --> myV2p(), 2 --> hasPages()
  <testID> : myV2p() --> 1-4, hasPages() --> 1-6 (both inclusive)
$ |
```

Additional Inferences →

In the tests for hasPages(), note that the # of code pages (.txt/.data) is always the same. This is expected because the code segment depends on the program size, and is loaded from memory only once before starting the execution of the program.

Also, the number of stack pages was always 1 (never 0), because even for an empty main() function, the argc, argv[] values are stored on the stack (which at least contains the program/command name).