Lab 3 Report

Alp Can Aloglu, Najmeh Mohammadi Arani

SID : 862190749 / 862216499

E-mail: aalog001@ucr.edu / nmoha034@ucr.edu

- **All code changes you made on xv6 source code** (hint: `diff -r original_xv6 your_xv6`)

  Code diff is submitted through different submissions.

- **Detailed explanation about these changes**

  proc.c

```
201a202,203
>   //the child inherit the number of stack pages from its parent
>   np->stack_pages = curproc->stack_pages;
```

  The child process (np) inherits the number of stack pages from it parent (currproc)

  proc.h

```
51a52,53
>   int stack_pages;
>   uint my_sp;
```

  We added a stack_pages feature to the process object to keep track of the number of stack pages.

  exec.c

```
66,69c67,73
<   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
<     goto bad;
<   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
<   sp = sz;
```

```
---
>   sp = KERNBASE - 1;
>
>   if((allocuvm(pgdir, sp - 2*PGSIZE, sp)) == 0)
>       goto bad;
>   clearpteu(pgdir, (char*)(sp - 2*PGSIZE));
>   curproc->stack_pages = 1;
>   cprintf("Initial number of stack pages: %d\n", curproc->stack_pages);
101a106
>   curproc->my_sp = sp;
```

We assigned the stack pointer to KERNBASE - 1. This way the stack would locate at the top of the memory. Then we allocated one page under the stack and made it inaccessible by using clearpteu. This page would play the role of page guard.

Then we updated the number of stack pages of the process to one. We also saved the stack pointer into my_sp to have access to it outside of exec.c.

Makefile

```
diff -r xv6_original/Makefile xv6_lab3/Makefile
179a180,181
>         _test1\
>         _test2\
```

syscall.c

```
20d19
<   struct proc *curproc = myproc();
22c21,23
<   if(addr >= curproc->sz || addr+4 > curproc->sz)
---
>
>
>   if(addr >= (KERNBASE-1) || addr+4 > (KERNBASE-1))
35d35
```

```
<   struct proc *curproc = myproc();
37c37,39
<   if(addr >= curproc->sz)
---
>
>
>   if(addr >= (KERNBASE-1))
40c42,43
<   ep = (char*)curproc->sz;
---
>   //ep = (char*)curproc->sz;
>   ep = (char*)(KERNBASE-1);
62c65
<   struct proc *curproc = myproc();
---
>
66c69,70
<   if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
---
>
>   if(size < 0 || (uint)i >= (KERNBASE-1) || (uint)i+size > (KERNBASE-1))
```

In fetchint, fetchstr, and argptr, we changed curproc->sz to KERNBASE-1, because with the new memory layout the sz is not at the top of the memory anymore. So we need to check if we are under the KERNBASE - 1 instead.

trap.c

```
79a81,105
>
>     //to implement growable stack
>       case T_PGFLT:;
>           uint fa = rcr2(); //determine the faulting address
>           if (fa >KERNBASE-1){
>               cprintf("The address is beyond the user space");
>               exit();
>           }
>           fa = PGROUNDDOWN(fa);
>           if (fa < myproc()->my_sp) {
>
>
>                   if (allocuvm(myproc()->pgdir, fa - PGSIZE, fa + PGSIZE) == 0) {
```

```
>                   cprintf("allocuvm failed. Number of stack pages: %d\n",
myproc()->stack_pages);
>                   exit();
>               }
>
>         }
>
>         deallocuvm(myproc()->pgdir, fa, fa - PGSIZE) ;
>         clearpteu(myproc()->pgdir, (char*)(fa - PGSIZE));
>
>         myproc()->stack_pages++;
>         cprintf("Increased stack size to %d pages\n", myproc()->stack_pages);
>         break;
```

To implement a growable stack, we added one more case inside the trap function when a page fault occurs (T_PGFLT). We used rcr2() to determine the address that caused a page fault. If the address is above the KERNBASE -1, we cannot grow the stack, so we must exit. But if that is not the case, we first round down the address, and then check to see if the address is less than the process's stack pointer, and if that is true, then we allocate two pages under the stack. The page guard under the stack overlaps with the newly allocated pages and we need to make a new page guard. First we deallocate the second page and make it inaccessible by using clearpteu function. Lastly, we increase the number of stack pages by one.

vm.c

```
337a340,365
>
>    //for stack
>    uint u = KERNBASE-1;
>    u = PGROUNDDOWN(u);
>    struct proc *curproc = myproc();
>    for(i = u; i > u - (curproc->stack_pages) * PGSIZE; i -= PGSIZE){
>
>       if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
>          panic("copyuvm: pte should exist");
>       if(!(*pte & PTE_P))
>          panic("copyuvm: page not present");
>       pa = PTE_ADDR(*pte);
>       flags = PTE_FLAGS(*pte);
>         if((mem = kalloc()) == 0)
>            goto bad;
>         memmove(mem, (char*)P2V(pa), PGSIZE);
>         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
>            goto bad;
>     }
>
>     if((allocuvm(pgdir, i+PGSIZE, i)) == 0)
>       goto bad;
>     clearpteu(pgdir, (char*)(i));
>
```

Copyuvm() function is used to make a copy of the parent proc's virtual memory for the child proc. Previously the virtual memory was consecutive from 0 to curproc()->sz. With the new memory layout, memory from 0 to curproc()->sz contains code and heap, and a one-page stack grows from KERNBASE towards 0 followed by a page guard. Therefore, we added the implementation for copying the stack and also added a page guard at the end.

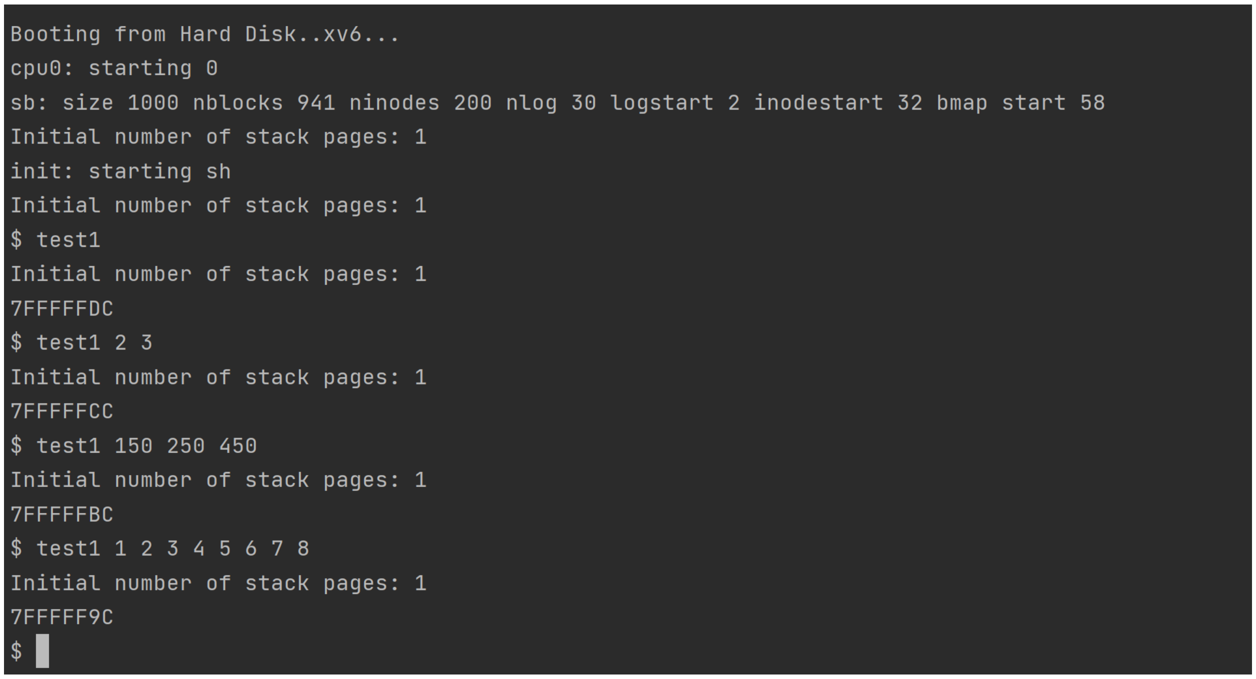
Screenshots about how you run the related program(s) and results

We had two provided test files for this lab.

test1.c:

```
#include "types.h"
#include "user.h"
#include "stat.h"

int
main(int argc, char *argv[])
{
    int v = argc;
    printf(1, "%p\n", &v);
    exit();
}
```

Screenshots of the result of running test1:



```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Initial number of stack pages: 1
init: starting sh
Initial number of stack pages: 1
$ test1
Initial number of stack pages: 1
7FFFFFDC
$ test1 2 3
Initial number of stack pages: 1
7FFFFFCC
$ test1 150 250 450
Initial number of stack pages: 1
7FFFFFBC
$ test1 1 2 3 4 5 6 7 8
Initial number of stack pages: 1
7FFFFF9C
$ 
```

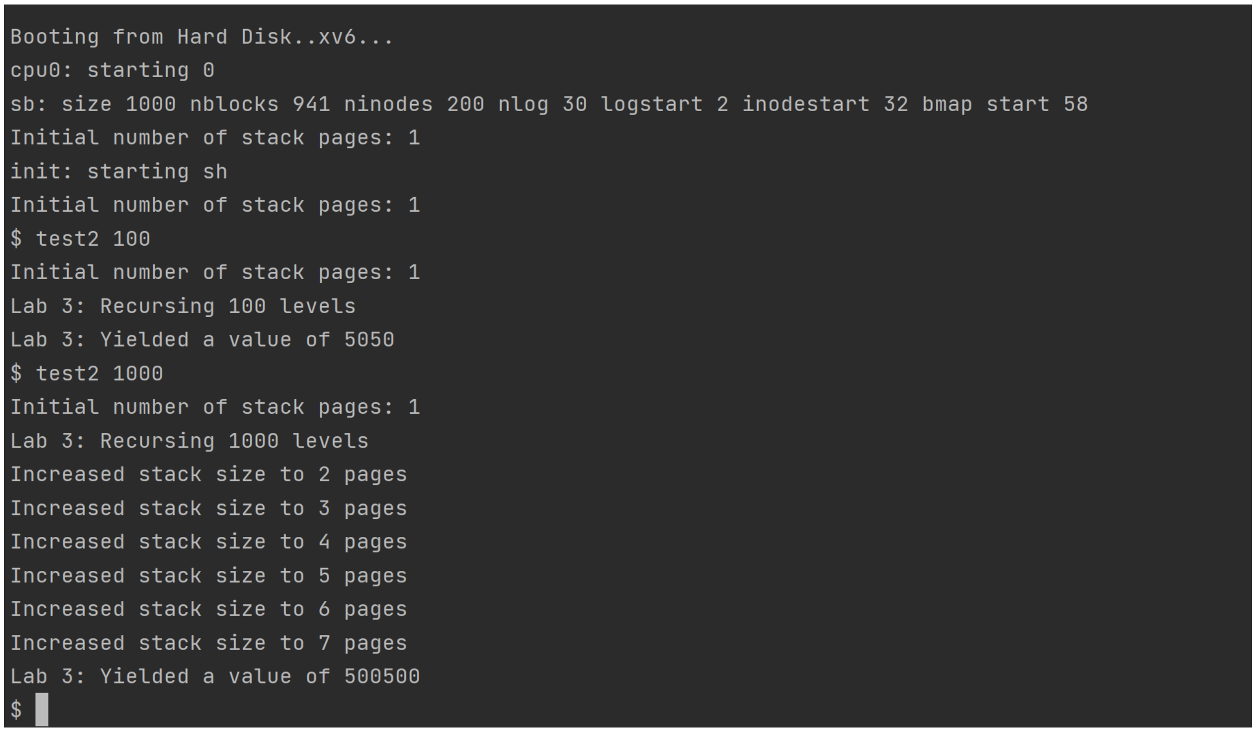
Analyze the result for test1:

The result for test1 shows that as we give the bigger inputs to the program, the returned address would decrease toward 0 since the stack grows downward.

test2.c:

```c
#include "types.h"
#include "user.h"

// Prevent this function from being optimized, which might give it closed form
#pragma GCC push_options
#pragma GCC optimize ("O0")
static int
recurse(int n)
{
    if(n == 0)
        return 0;
    return n + recurse(n - 1);
}
#pragma GCC pop_options

int
main(int argc, char *argv[])
{
    int n, m;

    if(argc != 2){
        printf(1, "Usage: %s levels\n", argv[0]);
        exit();
    }

    n = atoi(argv[1]);
    printf(1, "Lab 3: Recursing %d levels\n", n);
    m = recurse(n);
    printf(1, "Lab 3: Yielded a value of %d\n", m);
    exit();
}
```

Screenshots of the result of running test2:



```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Initial number of stack pages: 1
init: starting sh
Initial number of stack pages: 1
$ test2 100
Initial number of stack pages: 1
Lab 3: Recursing 100 levels
Lab 3: Yielded a value of 5050
$ test2 1000
Initial number of stack pages: 1
Lab 3: Recursing 1000 levels
Increased stack size to 2 pages
Increased stack size to 3 pages
Increased stack size to 4 pages
Increased stack size to 5 pages
Increased stack size to 6 pages
Increased stack size to 7 pages
Lab 3: Yielded a value of 500500
$
```

Analyze the result for test2:

Inside of the test2, there is a call to a recursive function. As the input given by us increases, the incursion function needs to use more of the memory space. (stack). If we give the input of 100, one stack page would be enough to keep the data. But if we give it a value of 1000, one stack page is not enough to keep the data, and the stack grows to 7 pages instead of returning a page fault because of the added implementation for stack growth.