CS179F Report

Virtual Memory on XV6

Advisor: Heng Yin

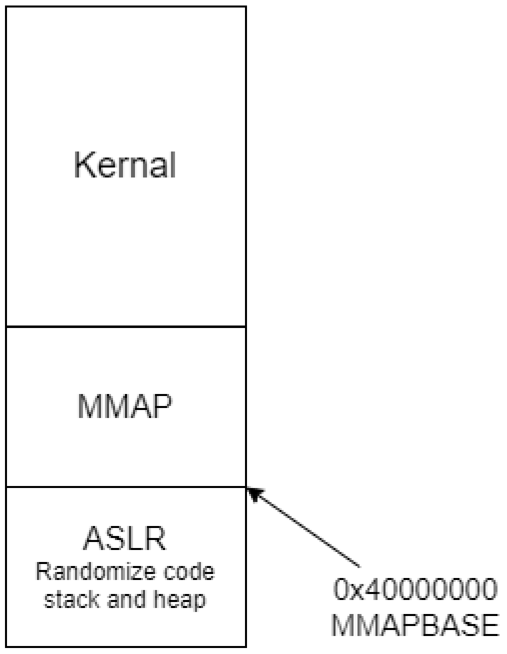
Student: Shuai Wang Yaming Zhang

Github Link: https://github.com/ws7474741/swang-ymzhang-CS179F.git

Content

1. Objectives

Our project is about virtual memory on XV6. We implemented Copy-on-Write, mmap and ASLR. Before we start, we need to understand the virtual memory in XV6 well, especially the virtual memory layout because all the three parts of our project have close connection with that. For Copy-on-Write, we copy the pages of the parent process when one of the processes makes changes on pages. For mmap, we leave the upper half of the user-level space for the mapping of the files. For ASLR, we randomize the base of code, stack and heap in the user space on the lower half. Below is what we aim to change on the xv6 virtual memory layout.



For the variables we need, we have added them into the process struct in proc.h to make our project easy going well.

```
struct proc {
    uint sz;                     // Size of process memory (bytes)
    pde_t* pgdir;                // Page table
    char *kstack;                // Bottom of kernel stack for process
    enum procstate state;        // Process state
    int pid;                     // Process ID
    struct proc *parent;         // Parent process
    struct trapframe *tf;        // Trap frame for current syscall
    struct context *context;     // swtch() here to run process
    void *chan;                  // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    struct file *ofile[NOFILE];  // Open files
•      struct inode *cwd;             // Current directory
•      char name[16];                 // Process name (debugging)
•   //----------cs179F------------//
•      uint mmapSz;                   // Total size of mmap area that used
•      struct mmapFile  *mfile[32]; // files that mapping with mmap()
•      int mfileIndex;                // Index to mfile array.
•      uint nextmmapAddr;             // next mmap page virutal address
•      int stackpos;                  // random position of the user stack
•      int stackpg;                   // number of stack pages of the
    process
•      int heappos;                   // random position of the user heap
•   };
```

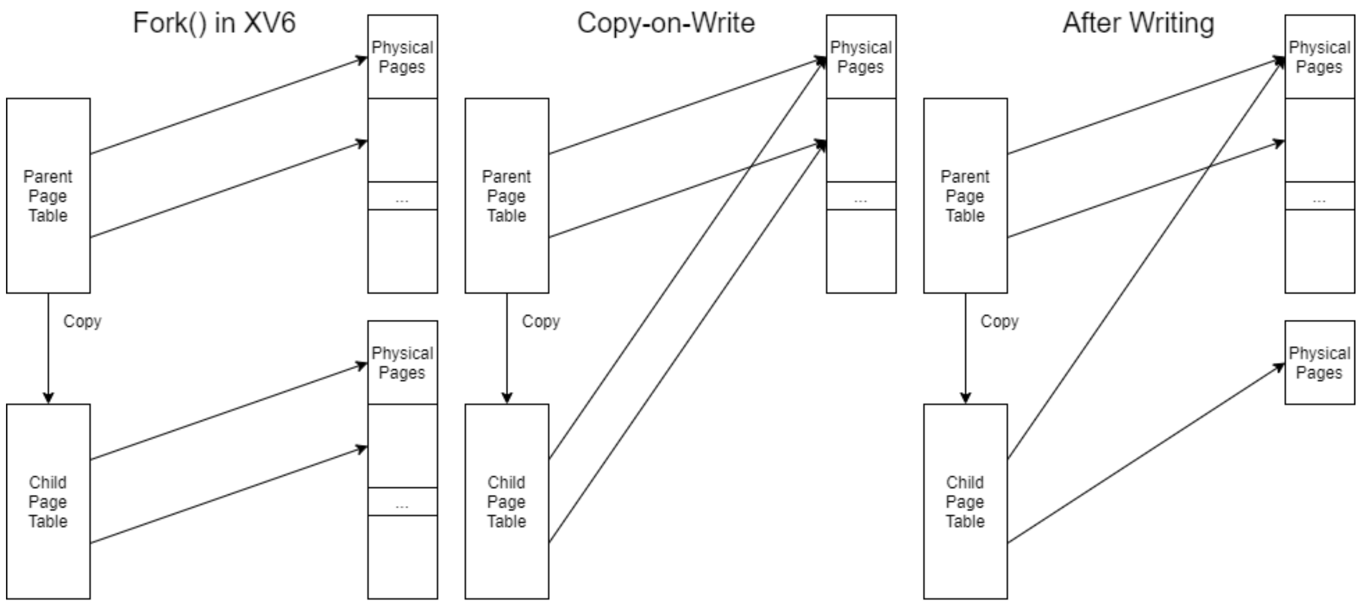
2. Copy-on-Write

1) Architecture

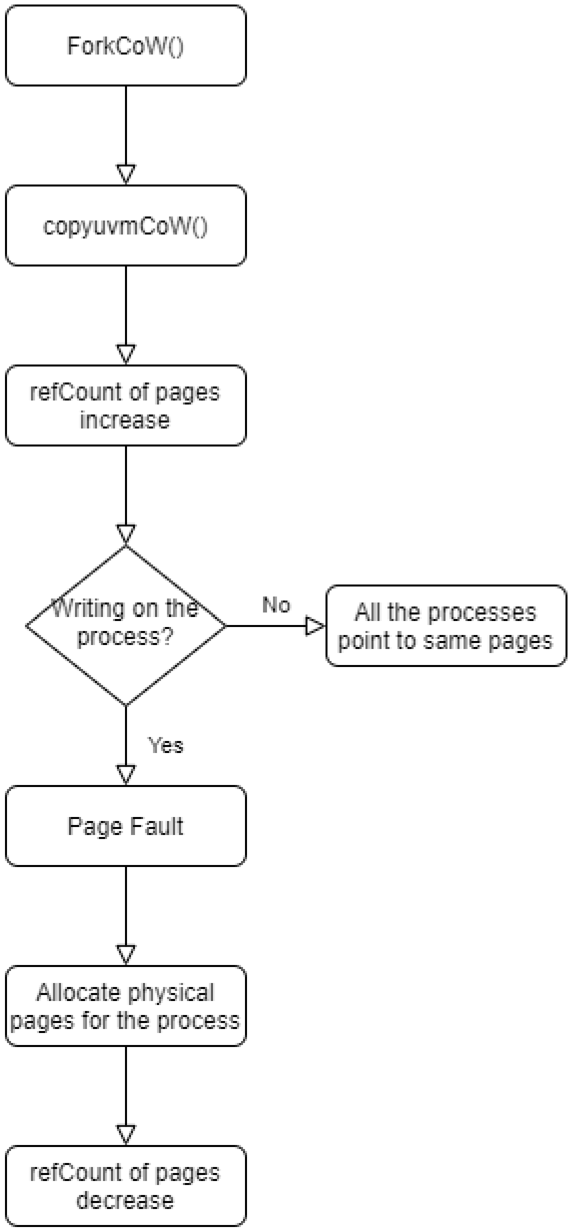
In XV6, fork() function uses copyuvm() method to copy parent processes' page tables and allocates new physical pages. It returns a new identical page directory for child process.

Copy on Write is a kind of resource management technique. The kernel will not allocate new physical pages for child process after creation. It will point the child process to the same pages that parent process holds. Additionally, both the pages of parent and child process will be set to read-only. Once one of the processes tries to write the read-only page, the kernel will detect the behavior and trap into a page fault, which will create a new physical page for the child process. That is Copy-on-Write, which can save resources significantly for unmodified copies.

2) Diagram



3) Flowchart


ForkCoW()
copyuvmCoW()
refCount of pages increase
Writing on the process?
No
All the processes point to same pages
Yes
Page Fault
Allocate physical pages for the process
refCount of pages decrease

4) Design Detail

a. We add a data structure kmem with an integer number called numberOfUnusedPage to trace the number of free pages in the OS. In kinit1, numberOfUnusedPage is set to 0. In kalloc() and kfree(), it is minored 1 and added 1 respectively.

```
struct {
    struct spinlock lock;
    int use_lock;
    struct run *freelist;

//----------cs179F-------------//
//Add a new varaible called 'numberOfUnusedPage' to trace how many free page.
    int numberOfUnusedPage;
} kmem;
```

b. We add a data structure to count how many child processes are pointing to the parent process's pages. While a new process is pointing to the parent process, the number will be incremented.

```
//----------cs179F------------//
//New data structure to keep track of the reference count of pages
struct {
    struct spinlock plock;
    // Use the page physical address as index to track every page.
    // PHYSTOP >> 12 is the number of pages that we could have in xv6.
    // If we have a page which page memory is x,
    // then refCount[x>>12] will index this page.
    int refCount[PHYSTOP>>12];
} pcounter;
```

c. We write the copyuvmCoW() function. When a child process is created, we do not create new pages. We just map the same pages of the parent process and set them to read-only.

```
//-------------cs179F-----------------//
// Given a parent process's page table, create a copy of it for a child.
// But this is a copy on write version of fork.
// We make child process and parent process point to the same page and
// set this page unwritable.
// We only allocate a page to the child process when one of the processes
// try to change the page.
```

```
pde_t*
copyuvmCoW(pde_t *pgdir, uint sz) {
    pde_t *d;
    pte_t *pte;
    uint pa, i, flags;
    //char *mem;

    if((d = setupkvm()) == 0)
        return 0;
    for(i = 0; i < sz; i += PGSIZE) {
        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
            panic("copyuvm: pte should exist");
        if(!(*pte & PTE_P))
            panic("copyuvm: page not present");


        //-------------cs179F--------------//
        *pte &= ~PTE_W;
        pa = PTE_ADDR(*pte);
        flags = PTE_FLAGS(*pte);
        if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
            goto bad;
        acquire(&pcounter.plock);
        pcounter.refCount[pa>>12]++;
        release(&pcounter.plock);
    }

    // Above code is to copy the code and text
    // The code below is to copy the stack in the virtual address space
    for(i = myproc()->stackpos - myproc()->stackpg*PGSIZE; i <
myproc()->stackpos; i += PGSIZE) {
        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
            panic("copyuvm: pte should exist");
        if(!(*pte & PTE_P))
            panic("copyuvm: page not present");
        *pte &= ~PTE_W;
        pa = PTE_ADDR(*pte);
        flags = PTE_FLAGS(*pte);
        if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
            goto bad;
        acquire(&pcounter.plock);
        pcounter.refCount[pa>>12]++;
        release(&pcounter.plock);
    }

    //This must be out of the 'for' loop.
    //Because we have to reinstall it when all pages are updated.
    lcr3(V2P(pgdir));
    return d;
bad:
    freevm(d);
    lcr3(V2P(pgdir));
    return 0;
}
```

Note that the function of lcr3(V2P(pgdir)) is significant. Once the page table of a process is changed, it has to be re-installed by writing the page table address to lcr3() function.

d. We write the forkCoW() function, using the copyuvmCoW() function we write above

```
//--------------cs179F-----------------//
// Copy-on-write system call
// Very similar with normal fork() function
// But we use copyuvmCoW instead of copyuvm
// So we don't actually allocate a new page
// for the child
// We simply point the child and parent process
// to the same page.
int
forkCoW(void) {
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

    if((np = allocproc()) == 0) {
        return -1;
    }

    //Use copyuvmCow instead of copyuvm
    if((np->pgdir = copyuvmCoW(curproc->pgdir, curproc->sz)) == 0) {
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;

    np->tf->eax = 0;

    for(i = 0; i < NOFILE; i++)
        if(curproc->ofile[i])
            np->ofile[i] = filedup(curproc->ofile[i]);
    np->cwd = idup(curproc->cwd);

    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
    pid = np->pid;
    acquire(&ptable.lock);
    np->state = RUNNABLE;
    release(&ptable.lock);

    return pid;
}
```

e. Once the child process is being written, the OS will detect the page fault. We need to implement the page fault handler to copy the page after the process is modified.

```
//CoW Page Fault, write to a read-only page.
else if(pte && (!(*pte & PTE_W))) {
    pa = PTE_ADDR(*pte);
    acquire(&pcounter.plock);

    //Only one ref to this page
    if(pcounter.refCount[pa>>12] == 1) {
        *pte |= PTE_W;
        release(&pcounter.plock);
        lcr3(V2P(curproc->pgdir));
        return;
    }
    //No one ref to this page
    if(pcounter.refCount[pa>>12] <= 0) {
        release(&pcounter.plock);
        cprintf("pgfHandler: page reference counter error. ( <= 0)\n");
        kill(curproc->pid);
        return;
    }
    //Multiple processes ref to this page
    if(pcounter.refCount[pa>>12] > 1) {
        release(&pcounter.plock);
        //this part is what we deleted in 'copyuvm'
        if((mem = kalloc()) == 0) {
            cprintf("pgfHandler: can not allocate a memory.\n");
            kill(curproc->pid);
            return;
        }
        memmove(mem, (char*)P2V(pa), PGSIZE);
        *pte = V2P(mem) | PTE_P | PTE_W | PTE_U;

        acquire(&pcounter.plock);
        pcounter.refCount[pa>>12]--;
        pcounter.refCount[V2P(mem)>>12]++;
        release(&pcounter.plock);
●           lcr3(V2P(curproc->pgdir));
●           return;
●       }
●   }
●   cprintf("Unknown Page Fault\n");
●   kill(curproc->pid);
●   return;
●   }
```

The rcr2() will return the virtual address of the fault page. If it is out of the range of virtua address, we have to kill the process. Otherwise, we make a copy for the child process.

5) Test

a. Function Test: Use fork() and ForkCoW() respectively to see how many pages they use for every fork.

```
●   pid = forkCoW();
●
●   if(pid == 0) {
●       printf(1,"This is CoW fork!\n");
●       printf(1,"C1:Before change data, data = %d, we have %d Unused
    Pages\n", data, getNumberOfUnusedPage());
●       data = 10;
●       printf(1,"C1:After change data, data = %d,  we have %d Unused
    Pages\n",data , getNumberOfUnusedPage());
●       exit();
●   }
●
●
●   int pid2;
●   pid2 = fork();
●
●   if(pid2 == 0) {
●       printf(1,"This is normal fork!\n");
●       printf(1,"C1:Before change data, data = %d, we have %d Unused
    Pages\n", data, getNumberOfUnusedPage());
●       data = 20;
●       printf(1,"C1:After change data, data = %d,  we have %d Unused
    Pages\n",data , getNumberOfUnusedPage());
●       exit();
●   }
●   wait();
●   wait();
●   printf(1,"\nfunction test success!!!\n\n");
```

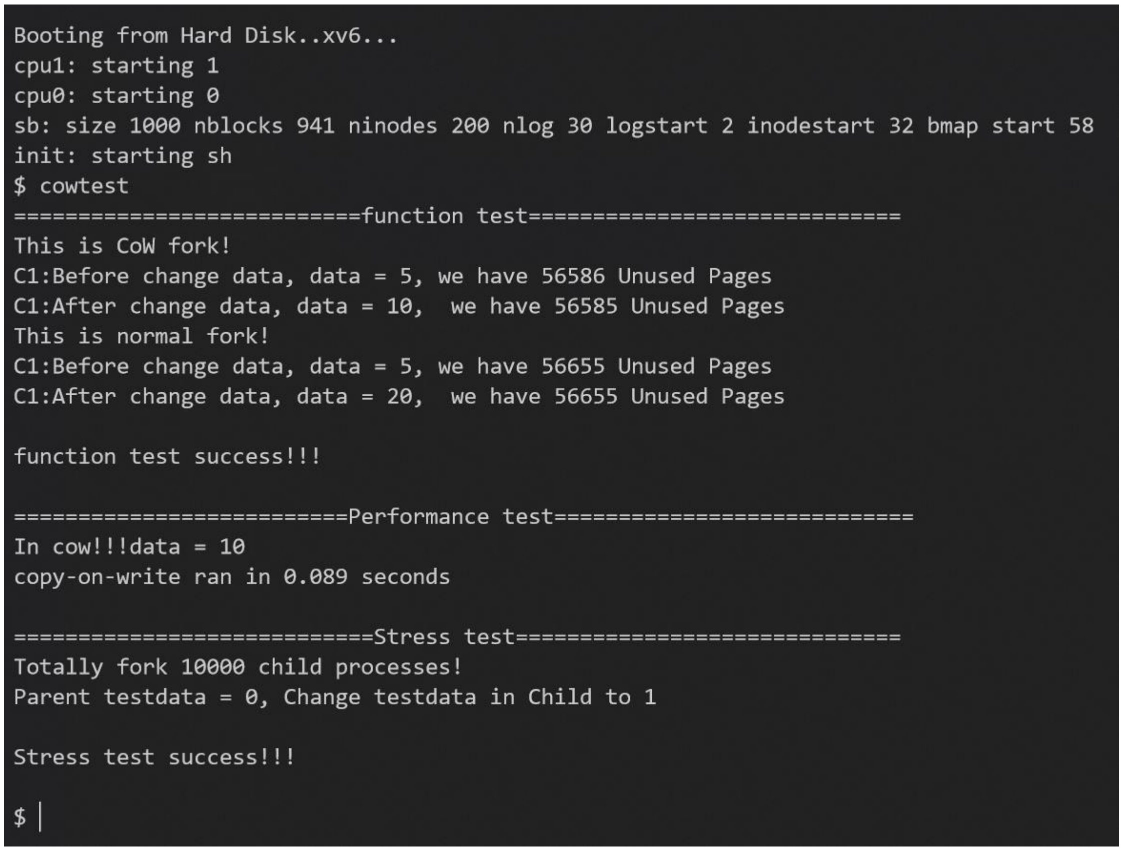
b. Performance Test: test the running time of Copy-on-Write.

```
int timeInit = uptime();
int retPid = fork();
if(retPid == 0) {
    exec("cow",argv);
    exit();
}


if(retPid > 0) {
    wait();
    int time = uptime();
    time = time - timeInit;
    int decim = time % 1000;
    time /= 1000;
    if(decim < 10)
        printf(1, "copy-on-write ran in %d.00%d seconds\n", time,
decim);
    else if(decim < 100 && decim >= 10)
        printf(1, "copy-on-write ran in %d.0%d seconds\n", time,
decim);
    else
        printf(1, "copy-on-write ran in %d.%d seconds\n", time, decim);
}
```

c. Stress test: Fork 1000 processes using forkCoW() system call to see if Copy-on-Write works fine under such pressure.

```
printf(1,"Totally fork 10000 child processes!\n");
int pidarray[10000];
int i;
int testdata = 0;
printf(1,"Parent testdata = %d, Change testdata in Child to 1\n",
testdata);

for(i = 0; i < 10000; i++) {
    if((pidarray[i] = forkCoW()) == 0) {
        testdata = 1;
        sleep(100);
        exit();
    }
}
int j;
for(j = 0; j < 10000; j++)
    wait();

printf(1,"\nStress test success!!!\n\n");
```

d. Test Result



```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ cowtest
===========================function test=============================
This is CoW fork!
C1:Before change data, data = 5, we have 56586 Unused Pages
C1:After change data, data = 10,  we have 56585 Unused Pages
This is normal fork!
C1:Before change data, data = 5, we have 56655 Unused Pages
C1:After change data, data = 20,  we have 56655 Unused Pages

function test success!!!

==========================Performance test============================
In cow!!!data = 10
copy-on-write ran in 0.089 seconds

============================Stress test==============================
Totally fork 10000 child processes!
Parent testdata = 0, Change testdata in Child to 1

Stress test success!!!

$ |
```

For Fork(), it will allocate pages to the child process immediately. So even after a writing operation, the total unused page will not change.

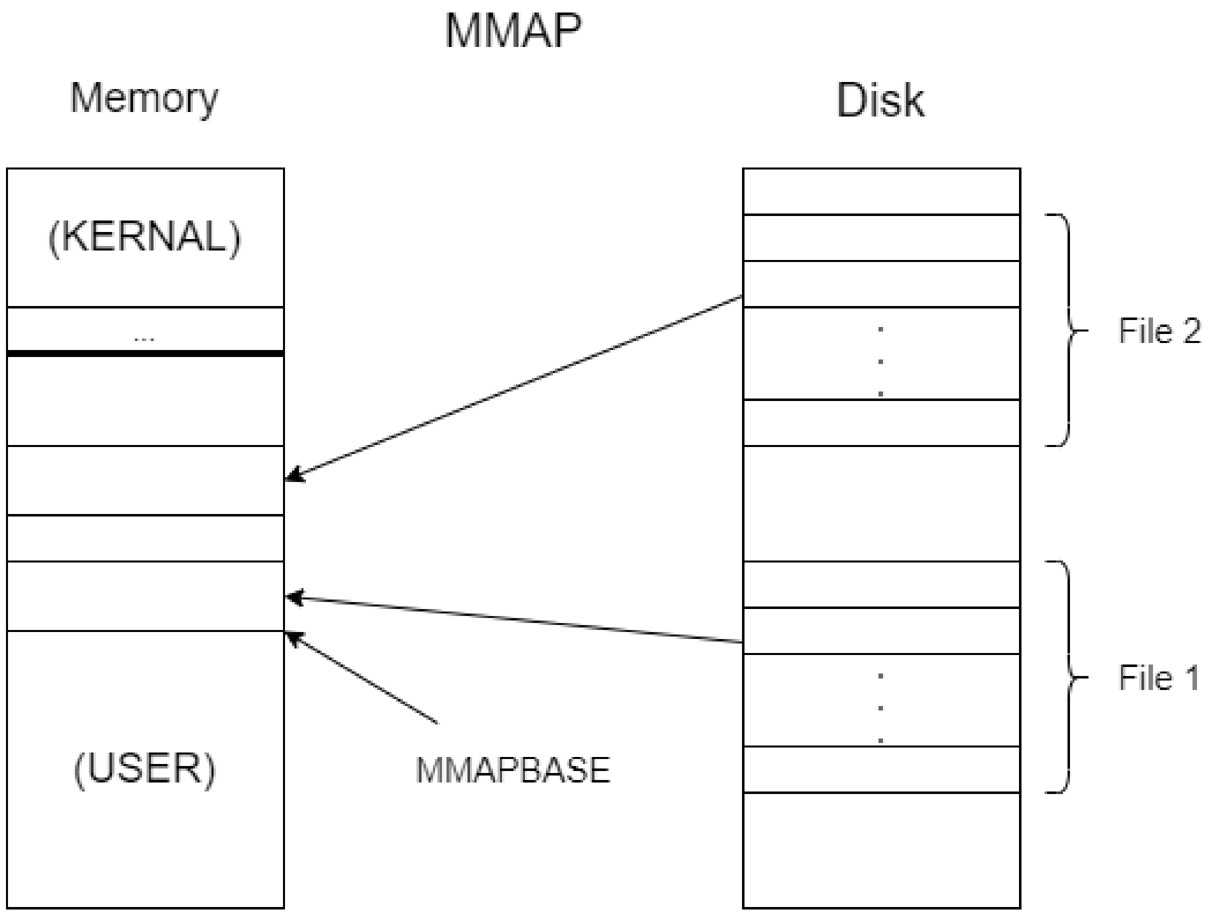
For forkCoW(), the parent process and child process will point to the same pages until one of processes has written the page. The OS will create a new page for the child process. So, the unused pages after writing are 1 page less than before.

3. Mmap

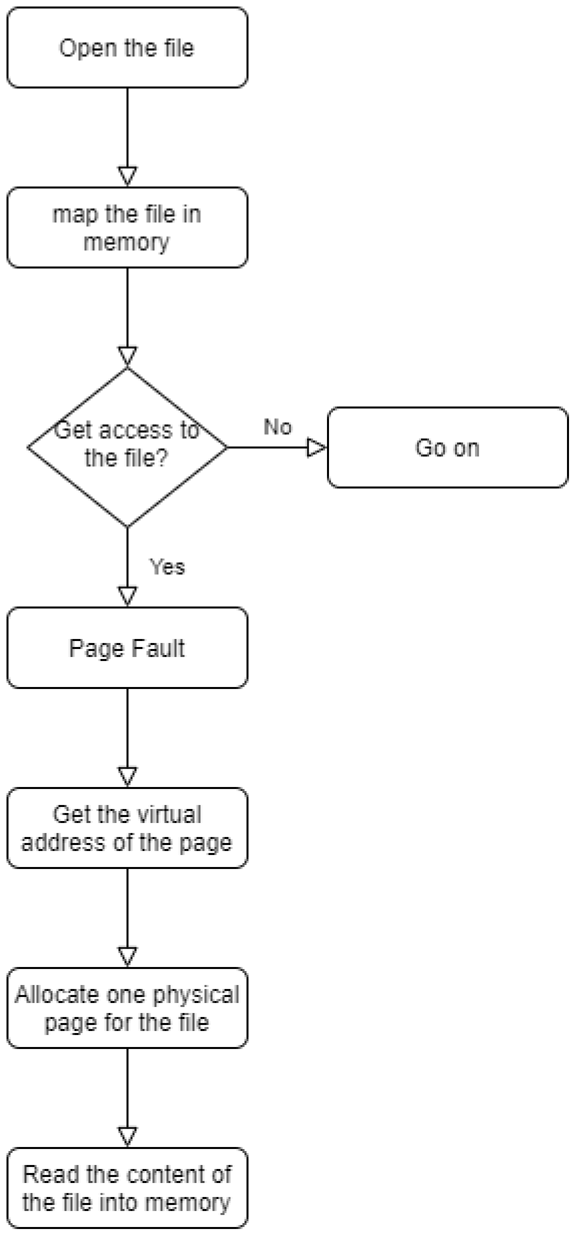
1) Architecture

Mmap maps files from disk into memory. It is a method of memory-mapped file I/O. It implements demand paging, because file contents are not read from disk directly and initially do not use physical RAM at all. The actual reads from disk are performed in a "lazy" manner, after a specific location is accessed.

2) Diagram



3) Flowchart


Open the file
map the file in memory
Get access to the file?
No
Go on
Yes
Page Fault
Get the virtual address of the page
Allocate one physical page for the file
Read the content of the file into memory

4) Design Details

a. Add a data structure for every process to store the information of files which are mapped to the memory.

```c
//---------cs179F---------//
//mmapFile struct stores info about files that used mmap to mapping
struct mmapFile {
    uint fileStartAddr;           // Address of this file's start place.
    uint fileEndAddr;             // Address of this file's end place.
    int fd;                       // File descriptor
    struct file *f;
};
```

b. Write the mmap system call to implement the lazy allocation of the file. While the user tries to access the file, it will be mapped to the user address space. We use the struct we created before to store the information about start address, end address, etc.

```c
//-----------cs179F-------------//
// mmap system call for the user to map a file into memory.
// This is a lazy allocation.
// Only allocate memory when the user tries to access the file content.
int
mmap(int fd, struct file *f) {
    struct proc *curproc = myproc();
    uint mPointer = curproc->mmapSz + MMAPBASE;
    uint startAddr = mPointer;
    uint endAddr;

    //update record informations
    curproc->mfile[curproc->mfileIndex]->fileStartAddr = startAddr;
    endAddr = curproc->mfile[curproc->mfileIndex]->fileStartAddr + f->ip->size;
    curproc->mfile[curproc->mfileIndex]->fileEndAddr = PGROUNDUP(endAddr) - 1;
    curproc->mfile[curproc->mfileIndex]->fd = fd;
    curproc->mfile[curproc->mfileIndex]->f = f;
    curproc->mmapSz = curproc->mmapSz + PGROUNDUP(curproc->mfile[curproc->mfileIndex]->fileEndAddr - curproc->mfile[curproc->mfileIndex]->fileStartAddr);
    curproc->mfileIndex++;
    return mPointer;
}
```

c. When the user tries to access the file, kernel will be trapped into a page fault. So, we write a page fault handler to allocate physical pages for the file and read the content into the memory.

```
//--------------cs179F-----------------//
// Page fault handler
// 1.Handle copy on write page fault
// 2.Handle mmap page fault
// 3.Handle growing stack page fault
void
pgfHandler(void) {
    uint pa;
    uint va = rcr2();   //rcr2() stores fault page's virtual address
    pte_t *pte;
    struct proc *curproc = myproc();
    char *mem;
    int i = 0;
    uint a;
    int sp;


    pte = walkpgdir(curproc->pgdir, (void*)va, 0);

    //Search PTE in pgdir
    //So if PTE not exist or pte value = 0 means we may caused by
mmap()
    if(!pte || *pte == 0) {
        for(i = 0; i < curproc->mfileIndex; i++) {
            if(va >= (uint)curproc->mfile[i]->fileStartAddr && va <=
(uint)curproc->mfile[i]->fileEndAddr) {
                if( va >= KERNBASE)
                    panic("mmap() tries to access kernel region!!");
                a = PGROUNDDOWN(va);
                mem = kalloc();
                if(mem == 0) {
                    cprintf("allocuvm out of memory\n");
                    kfree(mem);
                    return;
                }
                memset(mem, 0, PGSIZE);
                pte = walkpgdir(curproc->pgdir,(char*) a, 1);
                *pte = V2P(mem) | PTE_W | PTE_U | PTE_P;

                readi(curproc->mfile[i]->f->ip, (char*)a, a -
(uint)curproc->mfile[i]->fileStartAddr, PGSIZE);
                acquire(&pcounter.plock);
                pcounter.refCount[a >>12] = 1;
                release(&pcounter.plock);
•                   lcr3(V2P(curproc->pgdir));
•                   return;
•               }
•           }
•           cprintf("pgfHandler: pte should exist or mmap should be in
    range\n");
•           kill(curproc->pid);
•           return;
•       }
```

5) Test

a. Function Test: to test if we get access and read the file, how the number of pages changes.

```
•   int fd = open("CS179F1",0);
•   char* text = mmap(fd);
•   printf(1,"before we open the file, we have %d
    pages\n",getNumberOfUnusedPage());
•
•   printf(1,"===========================================================\n"
    );
•   printf(1,"\n%s\n",text);
•   printf(1,"\n===========================================================\n
    ");
•   printf(1,"After we mmap the file, we have %d
    pages\n",getNumberOfUnusedPage());
•
•   int fd2 = open("CS179F2",0);
•   char *text2 = mmap(fd2);
•   printf(1,"before we open the file, we have %d
    pages\n",getNumberOfUnusedPage());
•   printf(1,"============================================================\n
    ");
•   printf(1,"\n%s\n",text2);
•   printf(1,"\n===========================================================\n
    ");
•   printf(1,"After we mmap the file, we have %d
    pages\n",getNumberOfUnusedPage());
•
•   printf(1,"function test success!!!\n");
•   close(fd);
•   close(fd2);
```

b. Performance Test: test the running time of mmap.

```
•
•   int timeInit = uptime();
•   int retPid = fork();
```

```
if(retPid == 0) {
    exec("mmap",argv);
    exit();
}

if(retPid > 0) {
    wait();
    int time = uptime();
    time = time - timeInit;
    int decim = time % 1000;
    time /= 1000;
    if(decim < 10)
        printf(1, "mmap ran in %d.00%d seconds\n", time, decim);
    else if(decim < 100 && decim >= 10)
        printf(1, "mmap ran in %d.0%d seconds\n", time, decim);
    else
        printf(1, "mmap ran in %d.%d seconds\n", time, decim);
}
```

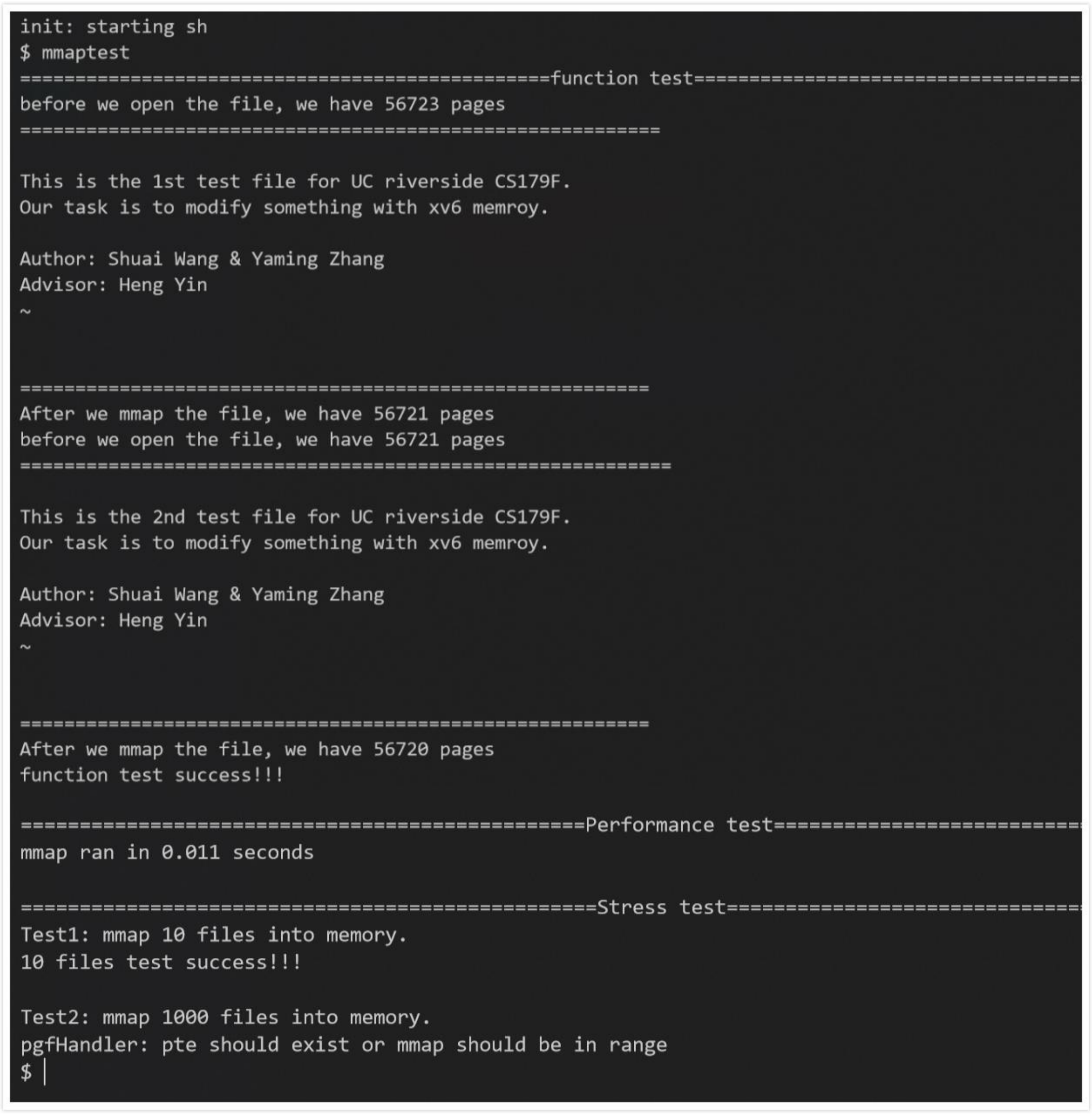
c. Stress Test: We map 10 files into the memory to see if the mmap works fine. Then we mmap 1000 files into the memory to see if the mmap works fine (It suppose to run out of memory).

```
printf(1,"Test1: mmap %d files into memory.\n",n);

int fdx[n];
int i;
for(i = 0; i < n; i++) {
    fdx[i] = open("CS179F1",0);
    char* Stresstext = mmap(fdx[i]);
    Stresstext[80] = 0;
}
for(i = 0; i < n; i++) {
    close(fdx[i]);
}

printf(1,"%d files test success!!!\n\n",n);

printf(1,"Test2: mmap %d files into memory.\n",m);
int fdx2[m];
int k;
for(k = 0; k < m; k++) {
    fdx2[k] = open("CS179F1",0);
    char* Stresstext = mmap(fdx2[k]);
    Stresstext[80] = 0;
}
for(k = 0; k < m; k++) {
    close(fdx2[k]);
}
```

```
•
•   printf(1,"%d files test success!!!\n",m);
```

The result is that if we map 1000 files into the memory, the mmap will fail because the mmap space will run out of the space that we allocate for it.

d. Test Result



```
init: starting sh
$ mmaptest
================================================function test===================================
before we open the file, we have 56723 pages
==========================================================

This is the 1st test file for UC riverside CS179F.
Our task is to modify something with xv6 memroy.

Author: Shuai Wang & Yaming Zhang
Advisor: Heng Yin
~


=========================================================
After we mmap the file, we have 56721 pages
before we open the file, we have 56721 pages
===========================================================

This is the 2nd test file for UC riverside CS179F.
Our task is to modify something with xv6 memroy.

Author: Shuai Wang & Yaming Zhang
Advisor: Heng Yin
~


=========================================================
After we mmap the file, we have 56720 pages
function test success!!!

===================================================Performance test===============================
mmap ran in 0.011 seconds

====================================================Stress test================================
Test1: mmap 10 files into memory.
10 files test success!!!

Test2: mmap 1000 files into memory.
pgfHandler: pte should exist or mmap should be in range
$ |
```

First we open the 1st file. Then we call mmap() to this file. We see that after we access the file, we allocate two pages of memory to the file. The first is the page directory and the second is the page in the memory. Then we open the 2nd file and do the same things like the first file. This time we only need to allocate one page because we do not need the page directory. For the stress test, we can see that we can map 10 files into memory but when it comes to 1000, the mmap will handle the error that there is no more memory for the files to allocate.

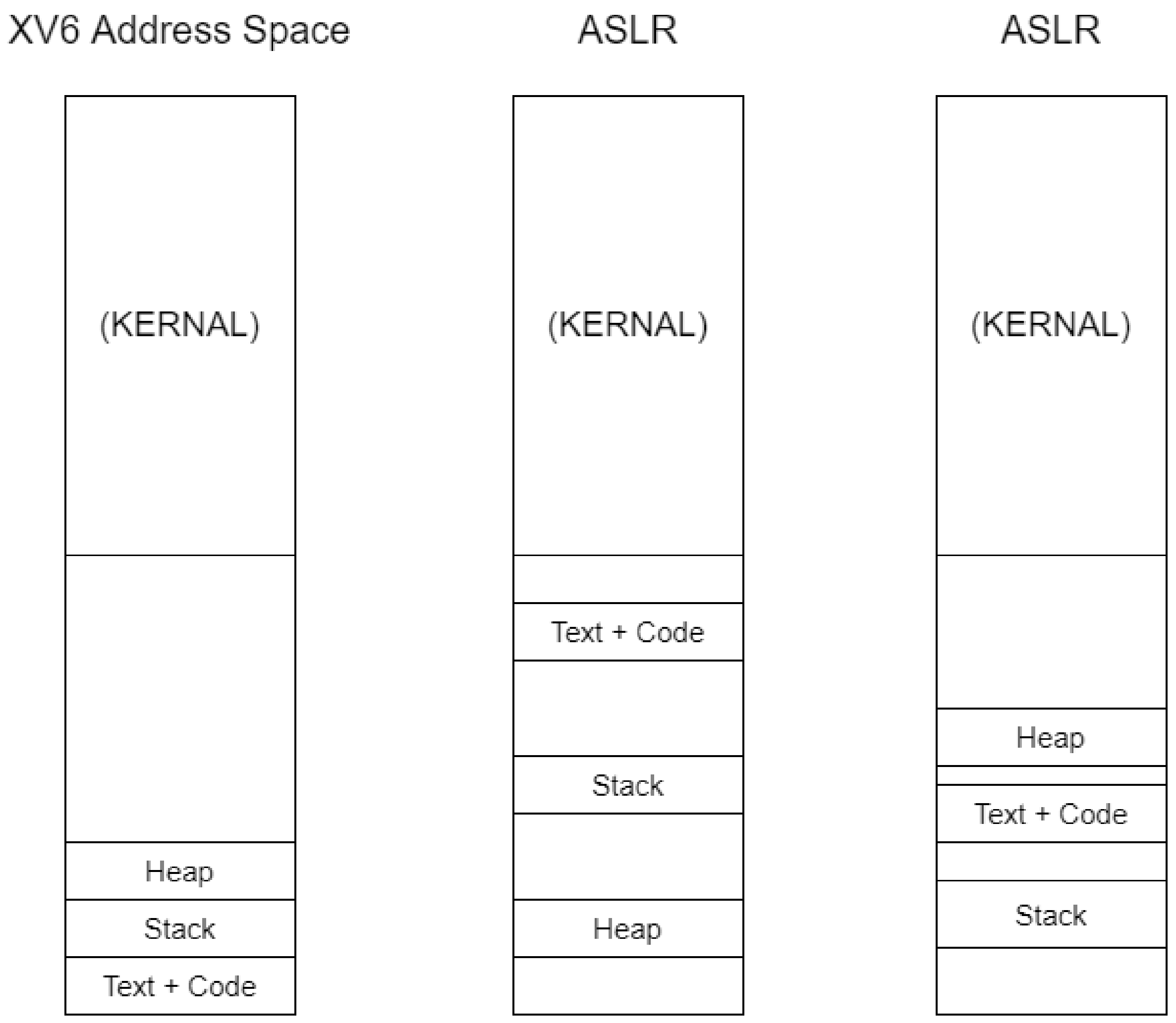
4. **Address Space Layout Randomization (ASLR)**

1) Architecture

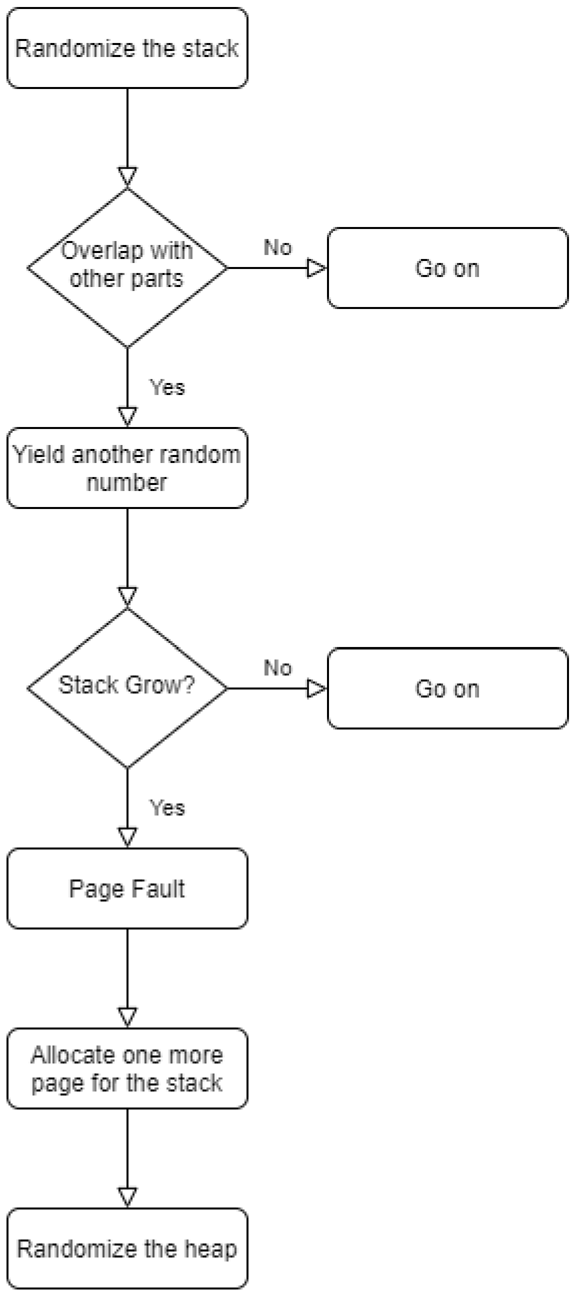
In XV6, Text and Code grow from 0 of a process's address space. The stack grows right above the code and text and the heap grows right above the stack.

Address Space Layout Randomization (ASLR) is a computer security technique which involves randomly positioning the base address of an executable and the position of libraries, heap, and stack, in a process's address space.

2) Diagram



3） Flowchart


Randomize the stack
Overlap with other parts
No
Go on
Yes
Yield another random number
Stack Grow?
No
Go on
Yes
Page Fault
Allocate one more page for the stack
Randomize the heap

3) Design Details

a. Write a random function to yield a random number for the base of the stack and the heap.

```
#include "rand.h"

//--------------------cs179F-------------//
// Yield the random number for ASLR base position

static unsigned int next = 1;

void srand(unsigned int seed) {
    next = seed;
}

int rand(void) {
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
```

b. Change the position of the stack and make the stack grow.

```
//----------------cs179F--------------//
// randomize the stack
cmostime(r);
srand(r->second);   // Yield the random number as the base position of the stack
stackpos = rand()*0x8001;   // Make the number between 0 to 0x40000000
while(stackpos <= sz + 2*PGSIZE)   // If the base position overlaps with other parts, yield another one.
    stackpos = rand()*0x8001;
stackpos = PGROUNDUP(stackpos);
if((sp = allocuvm(pgdir, stackpos - PGSIZE, stackpos)) == 0)   // Allocate user space for the stack
    goto bad;
```

c. Randomize the heap

```
// randomlize the heap
heappos = rand()*0x8001;
while(heappos <= sz || (heappos >= stackpos - PGSIZE && heappos <= stackpos + PGSIZE))
    heappos = rand()*0x8001;
heappos = PGROUNDDOWN(heappos);
```

d. Write a page fault handle to solve the page fault caused by the growing stack. If the virtual address of the page fault is under the MMAPBASE, it must be the fault caused by the growing stack.

```
// If the stack is growing, the page fault will get into this if sentence
if(va < MMAPBASE) {
    // Get the stack top
    sp = curproc->stackpos - curproc->stackpg*PGSIZE;
    // Allocate another page for the growing stack
    if(allocuvm(curproc->pgdir, sp - PGSIZE, sp) == 0)
        panic("No space for the growing stack!");
    // Number of stack page increase
    curproc->stackpg++;
    return;
}
```

e. Write three system calls to get the position of the stack and heap and the number pf pages of the stack for following tests.

```
// Get the current stack's position
int
sys_getstackpos(void) {
    return myproc()->stackpos;
}

// Get the current heap's position
int
sys_getheappos(void) {
    return myproc()->heappos;
}

// Get the number of current stack's pages
int
sys_getstackpg(void) {
    return myproc()->stackpg;
}
```

5) Test

a. Function Test: I write a recursive function to test the growing stack. The recursive function is to calculate the sum from one to a number input by the user. If the number is big, the operating system will be trapped into the page fault and grow the stack automatically. We use the system calls above to print the position of the stack and the heap. We can find that they are random.

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
// recursive function to test the growing of the stack
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;
    printf(1, "----------ASLR----------\n");
    printf(1, "stack position: 0x%x\n", getstackpos());
    printf(1, "heap position: 0x%x\n", getheappos());
    printf(1, "----------ASLR----------\n");

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

    n = atoi(argv[1]);
    m = recurse(n);
    printf(1, "The sum from 1 to %d is %d\n", n, m);
    printf(1, "This process uses %d stack pages\n", getstackpg());
    printf(1, "The stack top position 0x%x\n", getstackpos() -
getstackpg()*4096);
    printf(1,"function test success!!!\n");
```

b. Performance Test: test the running time of ASLR.

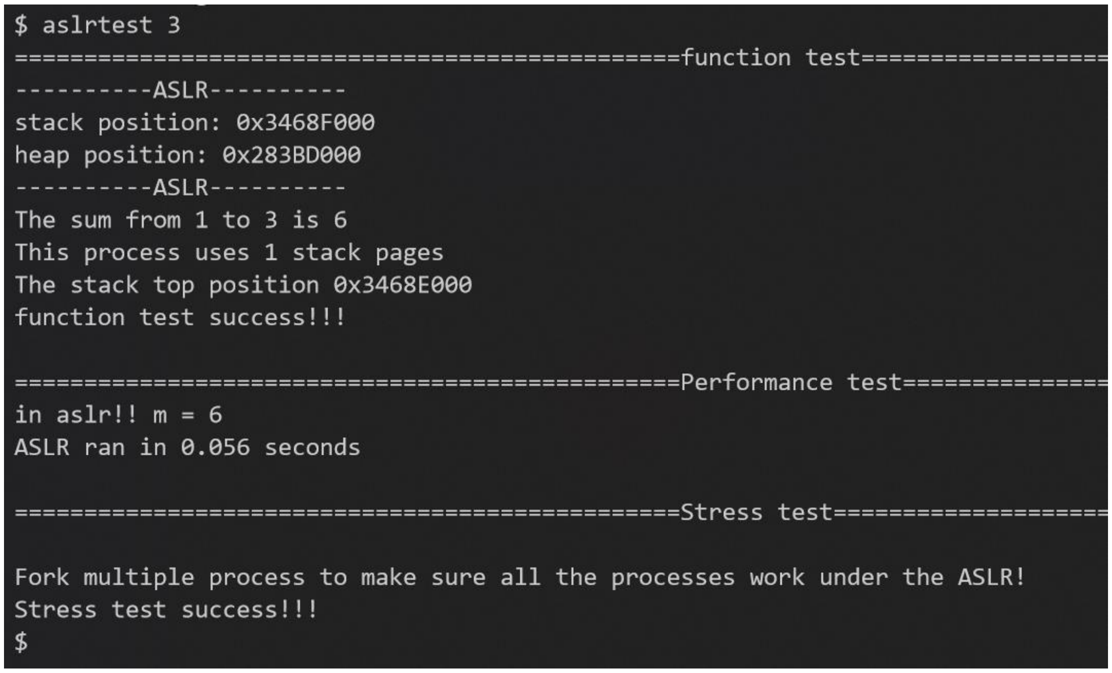
```
int timeInit = uptime();
int retPid = fork();
if(retPid == 0) {
    exec("aslr",argv);
    exit();
}

if(retPid > 0) {
    wait();
    int time = uptime();
    time = time - timeInit;
    int decim = time % 1000;
```

```
        time /= 1000;
        if(decim < 10)
            printf(1, "ASLR ran in %d.00%d seconds\n", time, decim);
        else if(decim < 100 && decim >= 10)
            printf(1, "ASLR ran in %d.0%d seconds\n", time, decim);
        else
            printf(1, "ASLR ran in %d.%d seconds\n", time, decim);
    }
```

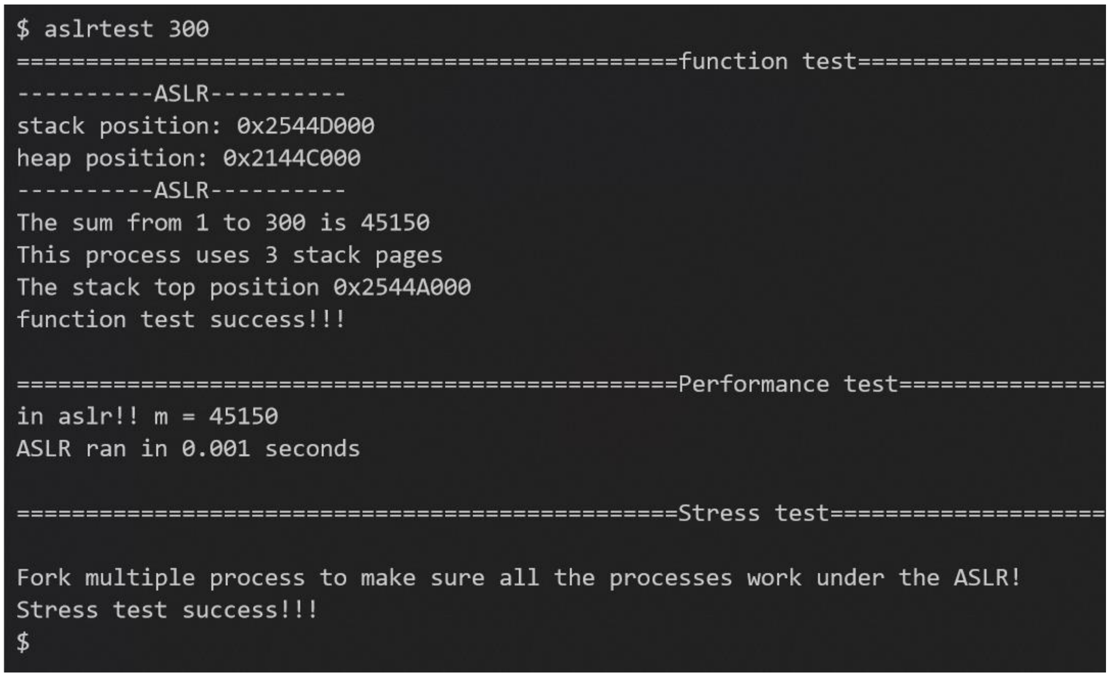
c. Stress Test: fork multiple processes to test if all the processes work fine on the ASLR.

```
printf(1,"\nFork multiple process to make sure all the processes work
under the ASLR!\n");
int i;
int pid;
for(i = 0; i < num; i++) {
    if((pid = fork()) == 0) {
        sleep(100);
        exit();
    }
}
for(i = 0; i < num; i++) {
    wait();
}
printf(1,"Stress test success!!!\n");

exit();
```

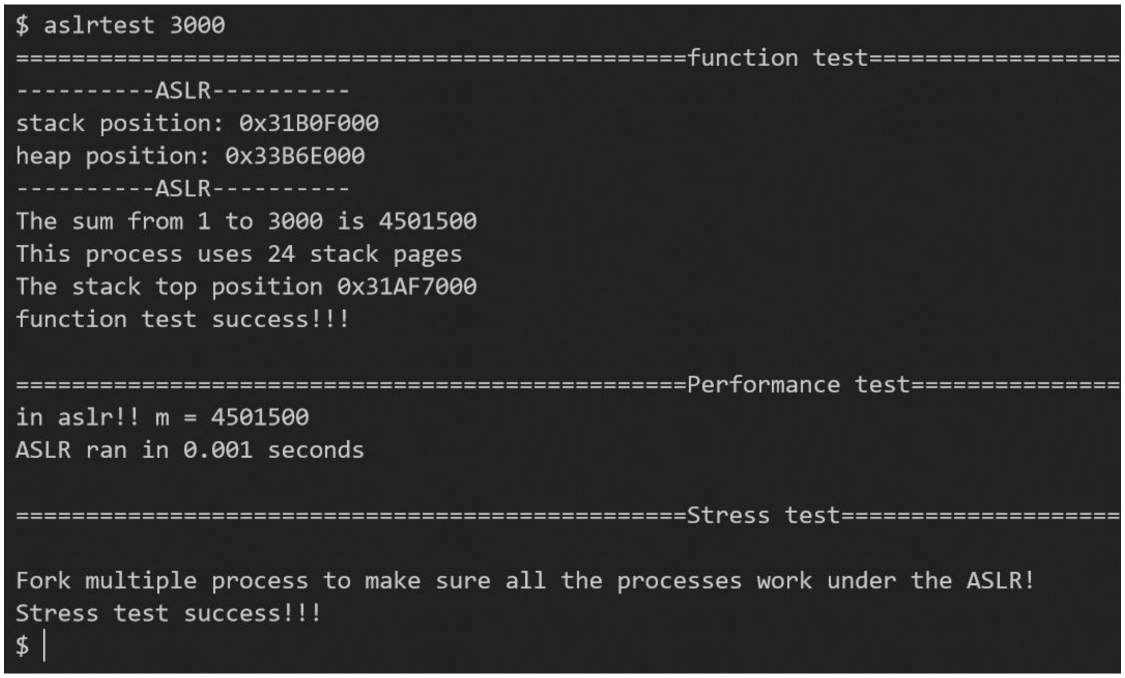
d. Test Result



```
$ aslrtest 3
================================================function test==================
----------ASLR----------
stack position: 0x3468F000
heap position: 0x283BD000
----------ASLR----------
The sum from 1 to 3 is 6
This process uses 1 stack pages
The stack top position 0x3468E000
function test success!!!

================================================Performance test===============
in aslr!! m = 6
ASLR ran in 0.056 seconds

================================================Stress test====================

Fork multiple process to make sure all the processes work under the ASLR!
Stress test success!!!
$
```



```
$ aslrtest 300
================================================function test==================
----------ASLR----------
stack position: 0x2544D000
heap position: 0x2144C000
----------ASLR----------
The sum from 1 to 300 is 45150
This process uses 3 stack pages
The stack top position 0x2544A000
function test success!!!

================================================Performance test===============
in aslr!! m = 45150
ASLR ran in 0.001 seconds

================================================Stress test====================

Fork multiple process to make sure all the processes work under the ASLR!
Stress test success!!!
$
```


```
$ aslrtest 3000
===============================================function test===================
----------ASLR----------
stack position: 0x31B0F000
heap position: 0x33B6E000
----------ASLR----------
The sum from 1 to 3000 is 4501500
This process uses 24 stack pages
The stack top position 0x31AF7000
function test success!!!

===============================================Performance test===============
in aslr!! m = 4501500
ASLR ran in 0.001 seconds

===============================================Stress test====================

Fork multiple process to make sure all the processes work under the ASLR!
Stress test success!!!
$ |
```

Every time we execute aslrtest, the stack position and the heap position are both random. For different numbers, the process will have a different number of pages in stack. We can see that the stack has been growing down.

5. Team

Shuai Wang is responsible for Copy-on-Write, mmap, performance & stress test and the installation of the Docker container.

Yaming Zhang is responsible for ASLR, document in README.md and the final report.