Project 3.A

This project can be divided into 2 parts, one part is to implement null pointer dereference and another part is to implement the mprotect () and munprotect (). In this report, I presented several key steps to finish it.

To implement the null pointer, I tried to leave the first page table blank and boot the xv6 from the second page 0x1000. To implement this, the first thing I do is do some modification on Makefile:

```
_%: %.o $(ULIB)
    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
```

should be modified to

Here I changed entry point from 0 to 0x1000 to make the first page invalid.

```
_%: %.o $(ULIB)
    $(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o $@ $^
```

After updating the Makefile, I looked into exec () and fork (). I firstly trace back to file *exec.c* to look into exec ().

```
// Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
```

The sz is assigned with PGSIZE, it is defined in the mmu.h, the header file contains the x86 memory management unit definitions, and PGSIZE = 4096, the same as 0x1000, and it means that when we start our program, the beginning memory point of the program becomes second page.

I also made sure when we worked on the virtual memory and physical memory, beginning page is well defined, so allocuvm (), loaduvm (), switchuvm (), setupkvm () are also carefully looked into.

```
// Load program into memory.
  sz = PGSIZE;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
```

Fortunately, they are not relevant with beginning page of the program, so we do not need to change anything. Then according to the project instruction, I looked into the fork (), which is defined in file *proc.c*, and here is one function copyuvm () used, I needed to do some modification on it, and copyuvm () is also defined in vm.c:

```
……
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");
  ……
  }
```

should be changed to

```
……
for (i = PGSIZE; 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");
  ……
  }
```

This means when we walk through the page directory table, we start from virtual address 0x1000, the second memory page, thus ensuring when we invoked fork () to create a child process, the virtual memory is accurately copied. (starting from second page and end to the process size limit in bytes). Finally, I googled and found that I should do some optimization on functions fetchstr() and fetchint() of syscall.c file, but I did not show them here.

The second part of the project 3A is only to add kernel level functions and relevant user functions. Because I have have previous done a project of adding a system call, here I just presented the core code segments I did:

sysproc.c

```
void _mprotect(struct proc *p, void *addr, int len) {
  uint vpn, ad = (uint)addr;
  int size = ad + len - 1;
    for (vpn = ad; vpn < size; vpn += PGSIZE) {
        pte_t *pte;
        pde_t *pde = p->pgdir;
        if ((pte = walkpgdir(pde, (void*)vpn, 0)) == 0) {
        } else {
      if ((*pte)&PTE_W && (*pte&PTE_U) == 0) {
        *pte = (*pte)&(~PTE_W);
        lcr3(v2p(proc->pgdir));
      }
        }
    }
  cprintf("\n");
}
```

```
void _munprotect(struct proc *p, void *addr, int len) {
  uint vpn, ad = (uint)addr;
  int size = ad + (len * PGSIZE) + 1;
    for (vpn = ad; vpn < size; vpn += PGSIZE) {
        pte_t *pte;
        pde_t *pde = p->pgdir;
        if ((pte = walkpgdir(pde, (void*)vpn, 0)) == 0) {
        } else {
      if (!(*pte&PTE_W) && (*pte&PTE_U) == 0) {
        *pte = *pte|PTE_W;
        lcr3(v2p(proc->pgdir));
      }
        }
    }
  cprintf("\n");
}
```

```
int kern_mprotect(void *addr, int len) {
  int rv = -1;
  if ((int)addr%PGSIZE != 0 || (int)addr > proc->sz || (int)addr <= 0) {
    return rv;
  }
  if (len <= 0 || ((int)addr + (len * PGSIZE)) > proc->sz) {
    return rv;
  }
  _mprotect(proc, addr, len);
  rv = 0;
    return rv;
}

int kern_munprotect(void *addr, int len) {
  int rv = -1;
  if ((int)addr%PGSIZE != 0 || (int)addr > proc->sz || (int)addr <= 0) {
    return rv;
  }
  if (len <= 0 || ((int)addr + (len*PGSIZE)) > proc->sz) {
    return rv;
  }
  _munprotect(proc, addr, len);
  rv = 0;
    return rv;
}
```

Project 3.B

In this project, I implemented clone () and join () as system calls, the core codes are attached below:

```
int sys_clone(void)↵
{↵
  void *arg1, *arg2, *stk;↵
  void(*fnc) (void *, void *);↵
  if(argptr(0, (void *)&fnc, 0) < 0)↵
    return -1;↵
  if(argptr(1, (void *)&arg1, sizeof(void*)) < 0)↵
    return -1;↵
  if(argptr(2, (void *)&arg2, sizeof(void*)) < 0)↵
    return -1;↵
  if(argptr(3, (void *)&stk, PGSIZE) < 0)↵
    return -1;↵
  if((uint)stk % PGSIZE)↵
    return -1;↵
  return clone(fnc, arg1, arg2, stk);↵
}↵
```

```c
int sys_join(void)
{
  void **stackPointer = 0;
  if(argptr(0, (void*)&stackPointer, sizeof(void *)) < 0)
    return -1;
  return join(stackPointer);
}
```

The user-level calls are defined in file *proc.c*

```c
int join(void **stkp)
{
  struct proc* proc = myproc();
  struct proc *p;
  int thexist, pid;
  acquire(&ptable.lock);
  while(1){
    thexist = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(proc->pgdir != p->pgdir || proc->pid == p->pid)
        continue;
      thexist = 1;
      if(p->state == ZOMBIE){
        pid = p->pid;
        kfree(p->kstack);
        p->pid = 0;
        p->parent = 0;
        p->killed = 0;
        p->name[0] = 0;
        p->kstack = 0;
        p->state = UNUSED;
        *stkp = p->stack;
        release(&ptable.lock);
        return pid;
      }
    }
    if(!(thexist && proc->killed)){
      release(&ptable.lock);
      return -1;
    }
    sleep(proc, 0);
  }
}
```

```c
int clone(void(*func) (void *, void *), void*
arg1, void* arg2, void* stack)
{
  struct proc *proc = myproc();
  int pid, i;
  struct proc *np;
  char *stackaddr;
  if(!(np = allocproc()))
    return -1;
  np->pgdir = proc->pgdir;
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;
  np->tf->eax = 0;
  np->tf->ebp = 0;
  np->tf->eip = (int)func;
  np->stack = stack;
  np->tf->esp = (int)(stack) + PGSIZE - 12;
  stackaddr = uva2ka(np->pgdir, (char*)stack);
  *(void **)(stackaddr + PGSIZE -12) =
(void*)0xffffffff;
  *(void **)(stackaddr + PGSIZE -8) = (void*)arg1;
  *(void **)(stackaddr + PGSIZE -4) = (void*)arg2;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);
  pid = np->pid;
  np->state = RUNNABLE;
  safestrcpy(np->name, proc->name,
sizeof(proc->name));
  return pid;
}
```

I also implemented several frequently used user-level functions:

```
void lock_init(lock_t *lock) {
  lock->ticket = 0;
  lock->turn = 0;
}

void lock_acquire(lock_t *lock) {
  int myturn = fadd(&lock->ticket, 1);
  while (lock->turn != myturn);
}

void lock_release(lock_t *lock) {
  fadd(&lock->turn, 1);
}
```

```
int
thread_join()
{
  void *stk;
  int cpid;
  //Need to free stack
  cpid = join(&stk);
  if (cpid != -1) {
    for(int i=0; i<ARRSZ; i++) {
      if(val_array[i] == cpid) {
        free(userstkaddr[i]);
        userstkaddr[i] = 0;
        val_array[i] = -1;
      }
    }
  }
  return cpid;
}
```

```
#define ARRSZ 64
int val_array[ARRSZ];
void* userstkaddr[ARRSZ];
int thread_create(void
(*start_routine)(void *, void *),
void *arg1, void *arg2)
{
  void* _stkaddr, *_stkpassed;
  _stkaddr = malloc(2*PGSIZE);
  if(_stkaddr == 0) {
    return -1;
  }
  //align the page
  int extspace =
((int)(_stkaddr))%PGSIZE;
  _stkpassed = (_stkaddr) + PGSIZE -
extspace;
  int cpid = clone(start_routine,
arg1, arg2, _stkpassed);
  if (cpid != -1) {
    for(int i=0; i<ARRSZ; i++ ) {
      if(userstkaddr[i] == 0) {
        userstkaddr[i] = _stkaddr;
        val_array[i] = cpid;
        break;
      }
    }
  }
  else {
    free(_stkaddr);
  }
  return cpid;
}
```