Lab Report 2

June 5th, 2017

Bradley Evans and Dharti Tarapara

CS153 Operating Systems

Part 1: Finding Physical Addresses from Virtual

We are given a virtual address and knowledge of how xv6 does addressing, and need to derive the physical address from there. We eventually accomplished this by way of the following code in `proc.c`.

```
void
v2p(int virtual, int* physical)
{

  unsigned short int dir = virtual>>22;
  unsigned short int table = (virtual>>12)&0x3ff;
  unsigned short int offset = virtual & 0xfff;

  pde_t *pde;
  pte_t *pgtab;

  // Find the page directory entry.
  pde = &proc->pgdir[PDX(virtual)];
  // Use the page directory entry to find the page table entry.
  pgtab = (pte_t*)V2P(PTE_ADDR(*pde));

  cprintf("Virtual Address: dir:0x%x table:0x%x offset:0x%x \n",dir,table,offset);
  cprintf("Page Table Address: 0x%x \n",*pgtab);

  *physical = (PTE_ADDR(pgtab[PTX(virtual)])) | (virtual & 0xFFF);
  cprintf("Physical Address: 0x%x \n",*physical);

}
```

This function is further implemented as a system call in the usual way (as it was in lab 1).

So, what does this do? We refer to figure 2-1 of the x86 textbook.

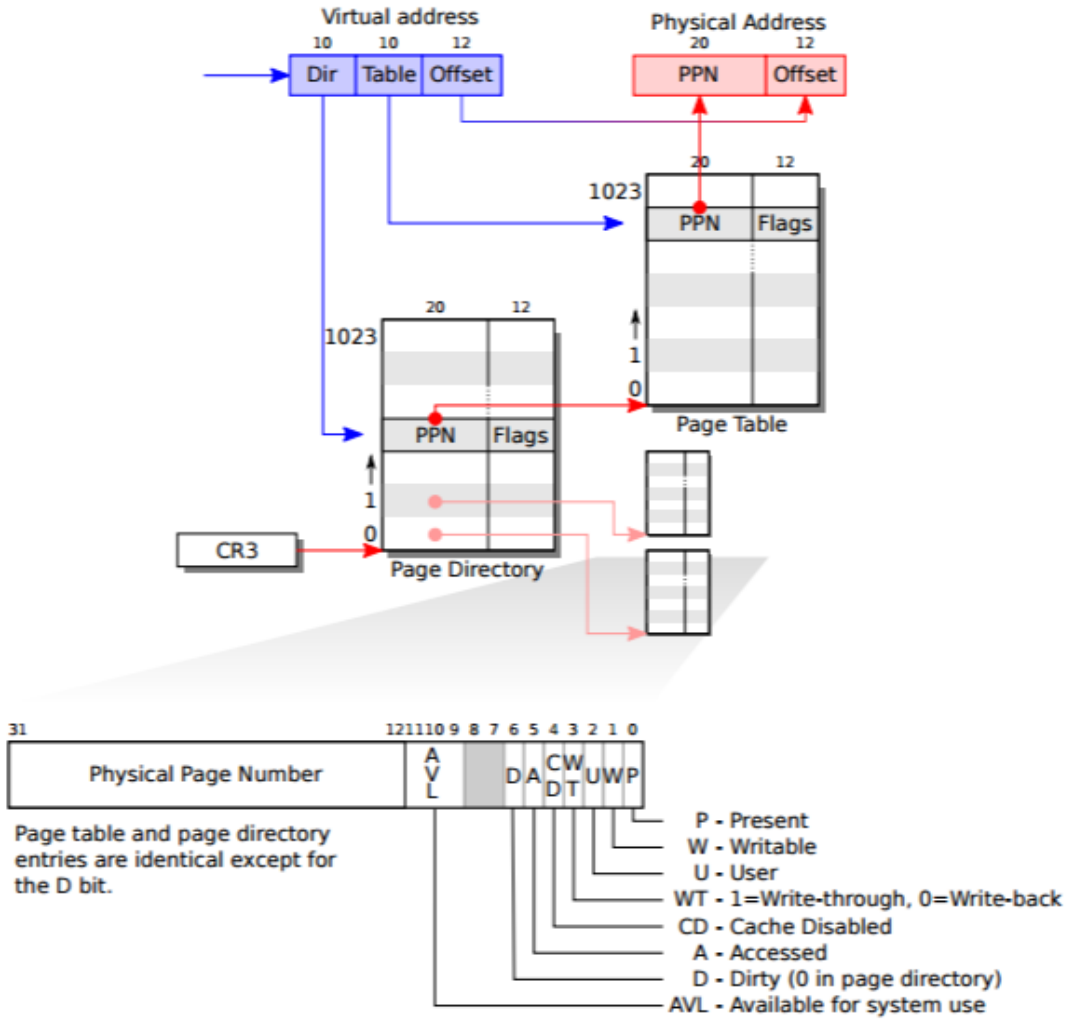

Figure 2-1. x86 page table hardware.

First we read in the virtual address `virtual`.

To derive the page directory entry `pde`, we reference the processes' page directory `proc->pgdir` with the first 10 bits of `virtual`. The mask `PDX(virtual)` strips out the first ten bits for us, providing the index within `proc->pgdir` that gives up the page directory entry we're looking for.

Now we need to find the appropriate page table entry. The first 20 bits of `pde` are the PPN (physical page number), and the middle 10 bits of `virtual` refer to the page table we're interested in.

We set `pgtab` by taking a mask of the first 20 bits of `pde` then casting this as a pointer.

Finally, we set `physical` to the PPN found in `pgtab` and the offset bits from the original `virtual` address.

We had to make use of an number of functions in `memlayout.h`, especially `V2P()`, `PTE_ADDR()`, and so on to make the functionality happen without writing a lot of unnecessary code.

Part 2: Dereferencing Null Pointers

To get this done, we had to create a reserved page of data that began at memory address zero and ensure that memory addressing began at this new "start point" (one page after address 0). Any attempt to access that first page would then produce an error. In `exec.c`, we reserve this space with the following code snippet (lines 74-78, `exec.c`):

```
if((sz = allocuvm(pgdir, sz, PGSIZE)) == 0)
  goto bad;
```

Additionally, we need to change particular limits in `syscall.c`. On lines 20 and 34, in `fetchint` and `fetchstr`, we add a sentinel to look for attempts to access memory address `0x00` and return an error state.

In `syscall.c`:

```
int
fetchint(uint addr, int *ip)
{
  if(addr >= proc->sz || addr+4 > proc->sz || addr==0)    // MOD : LAB2
    return -1;
  *ip = *(int*)(addr);
  return 0;
}

int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;

  if(addr >= proc->sz || addr==0)  // MOD : LAB2
    return -1;
  *pp = (char*)addr;
  ep = (char*)proc->sz;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
```

Finally, we need to modify the Makefile so that xv6 compiles correctly -- that is, it doesn't start by using address `0x00`.

In `Makefile`, lines 142-151:

```
_%: %.o $(ULIB)
        $(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o $@ $^
        $(OBJDUMP) -S $@ > $*.asm
        $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym
```

```
_forktest: forktest.o $(ULIB)
        # forktest has less library code linked in - needs to be small
        # in order to be able to max out the proc table.
        $(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o _forktest forktest.o ulib.o usys.o
        $(OBJDUMP) -S _forktest > forktest.asm
```

xv6 now throws errors when a null pointer is dereferenced.

Part 3: Stack Rearrangement

Our implentation of this part caused conflicts with Part 1 and 2, and so is seperately implemented (in its own xv6 instance).

The goal for this part of the lab is to rearrange the `xv6` address space to mimic Linux.

`memlayout.h`

In this file, we defined USERTOP.

```
#define USERTOP 0xDD4E000     // line 3
```

`syscall.c`

Next, we made sure that the current process' address stayed within the USERTOP threshold by replacing every instance of `proc-sz` with `USERTOP` or `KERNBASE`.

```
int
fetchint(uint addr, int *ip)
{
  if(addr >= USERTOP || addr+4 >= USERTOP)      // line 20
    return -1;
  *ip = *(int*)(addr);
  return 0;
}

int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;

  if(addr >= USERTOP)     // line 34
    return -1;
  *pp = (char*)addr;
  ep = (char*)USERTOP;      // line 37
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
```

```
}

int
argptr(int n, char **pp, int size)
{
  int i;
  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= KERNBASE || (uint)i+size > KERNBASE)      // line 60
    return -1;
  *pp = (char*)i;
  return 0;
}
```

`proc.h`

Then in this file, we defined the variable `stackTop` in `struct proc`. This variable will hold the top of the stack.

```
uint stackTop;      // line 55
```

`exec.c`

Here, we then defined another variable also called `stackTop`to use internally.

```
uint stackTop;      // line 15
```

We allocated an inacessible page and and a second page for the user stack.

```
stackTop = USERTOP - (2 * PGSIZE);      // lines 67 - 72
  if ((sp = allocuvm(pgdir, stackTop, USERTOP)) == 0) {
    cprintf("exec.c 69\n");
    goto bad;
  }
  clearpteu(pgdir, (char*)stackTop);
```

We also set the process' `stackTop` equal to the internal `stackTop` value.

```
proc->stackTop = stackTop;      // line 109
```

`proc.c`

In this file, we set the process' initial `stackTop` value to zero in the `userinit` function.

```
p->stackTop = 0;      // line 93
```

Next, we added code to copy the process state from `p` into the new process with the new variable `proc->stackTop`.

```
// Copy process state from p
if((np->pgdir = copyuvm(proc->pgdir, proc->sz, proc->stackTop)) == 0){      // line 153 - 162
  kfree(np->kstack);
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
np->sz = proc->sz;
np->parent = proc;
np->stackTop = proc->stackTop;      // line 161
*np->tf = *proc->tf;
```

`trap.c`

Here, we added a case for page faults when the OS kills a process.

```
case T_PGFLT:     // line 80 - 88
        if(growstack(proc->pgdir, proc->tf->esp, proc->stackTop) == 0)
                break;
        cprintf("pid %d %s: page fault on %d eip 0x%x ",proc->pid, proc->name, cpu->apicid, tf->eip);
        cprintf("stack 0x%x sz 0x%x addr 0x%x\n", proc->stackTop, proc->sz, rcr2());
        if(proc->tf->esp > proc->sz)
                deallocuvm(proc->pgdir, USERTOP, proc->stackTop);
        proc->killed = 1;
        break;
```

`vm.c`

In this file, we wrote a function `growstack` to grow the stack based on whether memory is already present or new memory needs to be allocated.

```
int     // line 391 - 413
growstack(pde_t *pgdir, uint sp, uint stackTop)
{
  pte_t *pte;
        uint newTop = stackTop - PGSIZE;
  cprintf("vm.c 392\n");
        if (sp > (stackTop + PGSIZE))
                return -1;


        // don't allocate new memory if already present
        if((pte = walkpgdir(pgdir, (void *) newTop, 1)) == 0)
                return -1;
        if(*pte & PTE_P)
                return -1;
        if(allocuvm(pgdir, newTop, stackTop) == 0)
                return -1;
```

```
        proc->stackTop = proc->stackTop - PGSIZE;
        setpteu(proc->pgdir, (char *)(proc->stackTop + PGSIZE));
        clearpteu(proc->pgdir, (char *)proc->stackTop);
        return 0;
}
```

We also needed a helper function `setpteu` to set PTE_U on a page.

```
void      // line 320 - 330
setpteu(pde_t *pgdir, char *uva)
{
  cprintf("vm.c 323\n");
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if(pte == 0)
    panic("setpteu");
  *pte |= PTE_U;
}
```

Then, we modified `copyuvm` to take in a third parameter, `stack_top`, and use it to copy memory within the function for when there's a child process.

```
pde_t*
copyuvm(pde_t *pgdir, uint sz, uint stack_top)      /// line 336
{
  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");
    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;
  }
  //cprintf("vm.c 358\n");
```

```
  // part 3 - start
  // for copying memory for child process
  if (stack_top == 0) return d;     // line 361 - 381

  // copy stack
  for(i = stack_top; i < USERTOP; i += PGSIZE){
    cprintf(".");
                if((pte = walkpgdir(pgdir, (void *) i, 1)) == 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;
        }
        cprintf("\n");
  // part 3 - end
  cprintf("vm.c 378\n");
  return d;

bad:
  cprintf("vm.c 382\n");
  freevm(d);
  return 0;
}
```

`defs.h`

The following functions were modified or added into the `defs.h` file.

```
pde_t*          copyuvm(pde_t*, uint, uint);      // line 179
void            setpteu(pde_t *pgdir, char *uva);       // line 184
int             growstack(pde_t*, uint, uint);      // line 185
```