CSCI 350 Project 4

PART 1

Kivilcim Cumbul

Task 1: Enhanced process details viewer

Will be done in proc.c

Task 1

For task 1, you are asked to print:

virtual page number -> physical page number, writable?

...

virtual page number -> physical page number, writable?

For example, in a system with 2 processes, the information should be displayed as follows:

```
1 sleep init 80104907 80104647 8010600a 80105216 801063d9 801061dd
1 -> 300, y
200 -> 500, n
2 sleep sh 80104907 80100966 80101d9e 8010113d 80105425 80105216 801063d9
1 -> 306, y
200 -> 500, n
```

In mmu.h, you can use constants and flags for page table.

```c
// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page

#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
```

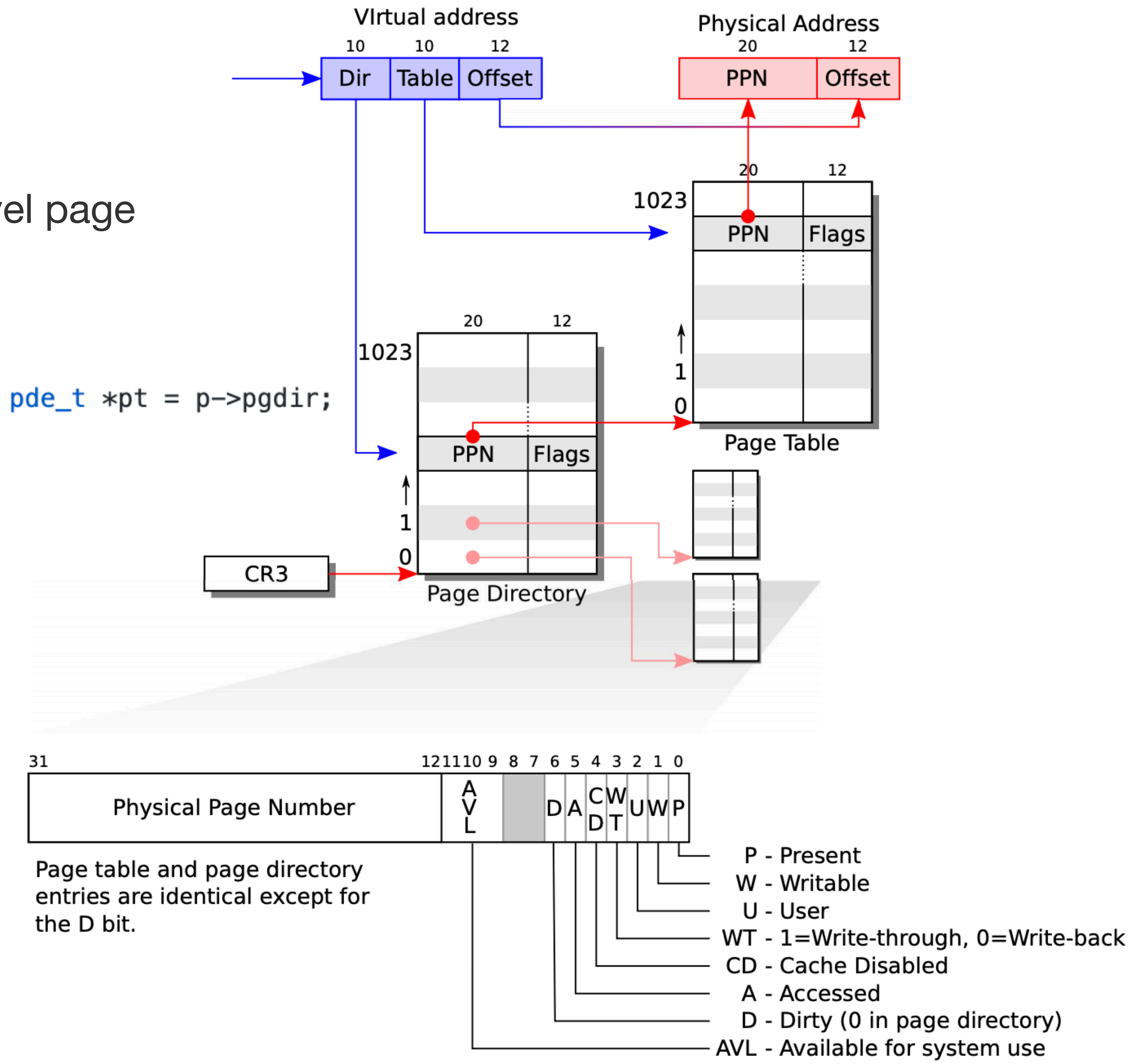
xv6 uses 2-level page tables.

p->pgdir is the first address of the first level page table of process p

This is called page directory in xv6

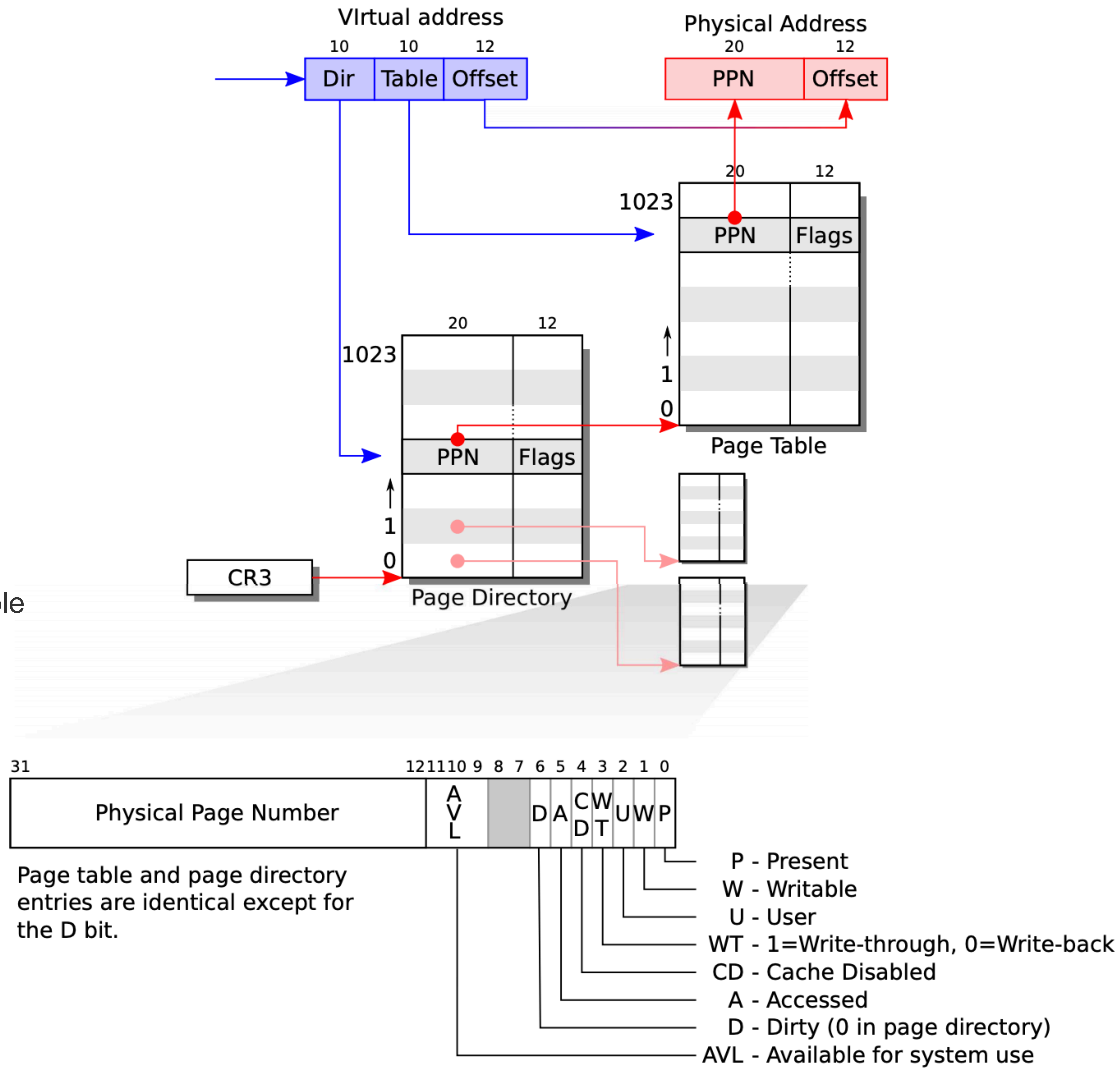
Get first table which is page directory:

```
pde_t *pt = p->pgdir;
```



xv6 uses 2-level page tables.

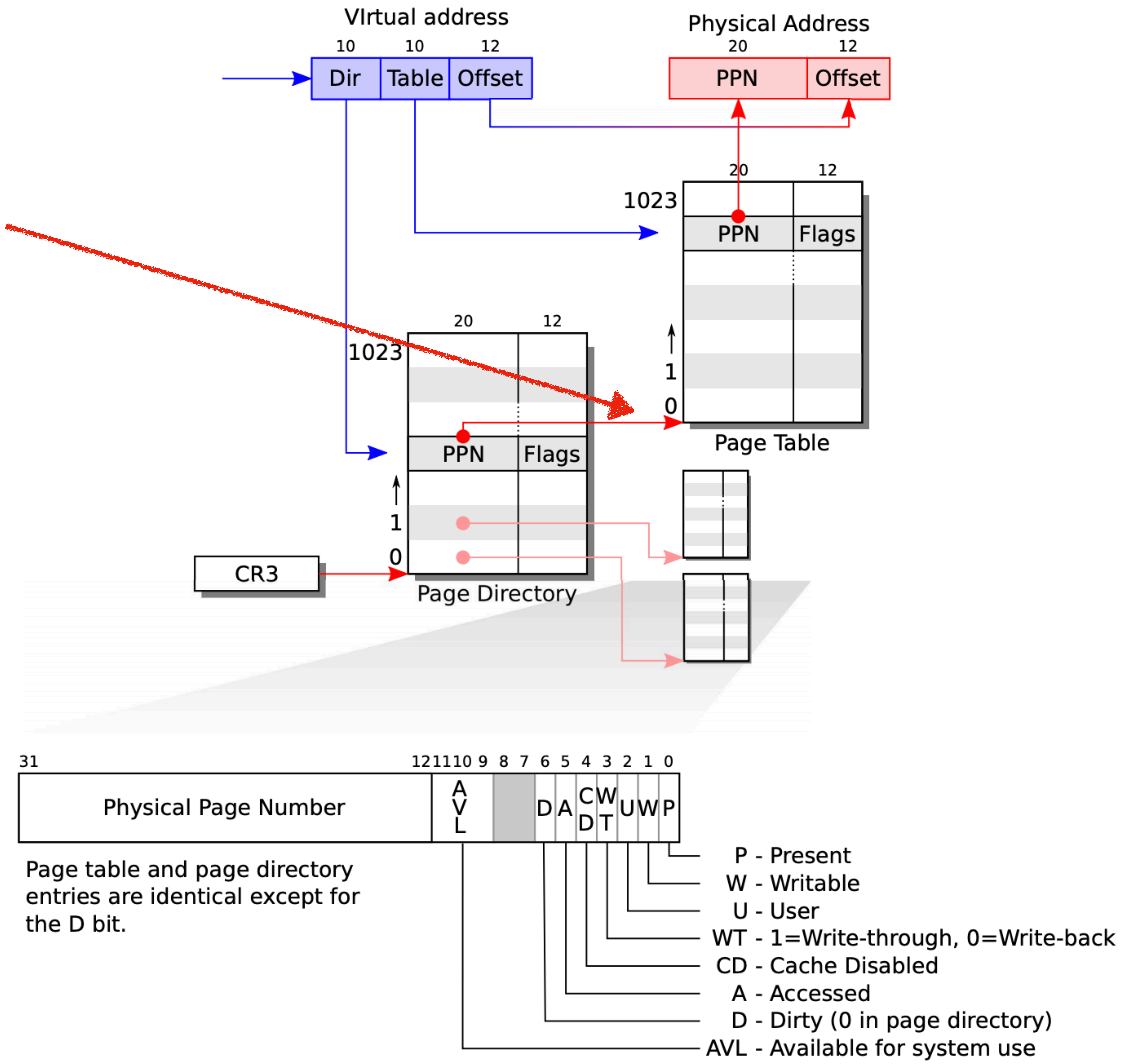
The first 20 bits gives the physical frame number of the physical page containing the second level page table associated with that entry



xv6 uses 2-level page tables.

Convert the physical address of the second level table into virtual address

You have to check memlayout.h for P2V and V2P method usage



memlayout.h

```c
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
```

After finding the virtual address of the page table:

Loop through page table, (NPTENTRIES)

Each process's virtual address space is divided into two parts, the user part and the kernel part

Find if it is present and user

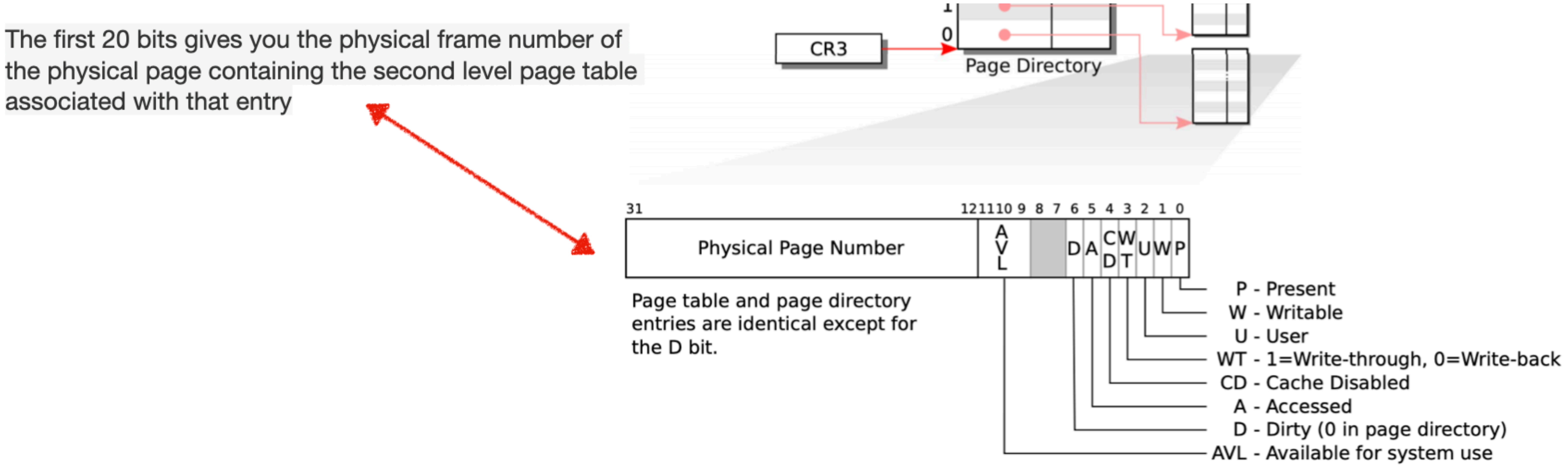
```
// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page

#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size
```

Use these flags

(Address & PTE_P && Address&PTE_U)

Cont'd

If present and user: find physical address again which is first 20 bits



You need to use bit masking and/or shifting!!

Cont’d

Check if the entry is writableusing these flags
and print y or n

```
// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page

#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size
```

Use these flags

If (Address & PTE_W) -> “y”

Task 2: Copy on Write (COW)

Task 2

Most changes will be in vm.c

Upon execution of a fork command, a process is duplicated together with its memory

In this task you will implement the COW optimization in xv6. To accomplish this task, change the behavior of the fork system call so that it will not copy memory pages, and the virtual memory of the parent and child processes will point to the same physical pages

If parent and child share the same table it has to be read only so that both parent and child cannot change the table. If someone tries to make a change a page fault will be raised and then the page should be copied to a new place

In order to be able to distinguish between a shared read-only page and a non-shared read-only page, add another flag to each virtual page to mark it as a shared page.

Let's start from the last sentence

In order to be able to distinguish between a shared read-only page and a non-shared read-only page, add another flag to each virtual page to mark it as a shared page.

We already have some flags in mmu.h so you can add this flag to mmu.h as well

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size
```

Define a new flag here!

In mmu.h

To facilitate such a decision, add a counter for each physical page to keep track of the number of processes sharing it (it is up to you to decide where to keep these counters and how to handle them properly). Since a limit of 64 processes is defined in the system, a counter of 1 byte per page should suffice. Since the wait system call deallocates all of the process's user space memory, it requires additional attention – remember do not deallocate the shared pages.

A parent can fork multiple times so we have to keep track of how many processes share the same table

You need some kind of **synchronization mechanism** for this counter

hint1: check **PHYSTOP** and **PGSIZE** to determine how many counters you need

hint2: Initialize counters in **inituvm** and **allocuvm**

In vm.c

In **dealloc**:

if counter is 0: **free page table** in dealloc

If not: decrement counter and check if counter is 0 again.

If counter is now 0, update shared and writable flag

```
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
}
```

You might use this function to **free page table**

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size
```

You will use these flags and the shared flag you created

In vm.c

Hint: You can check copyuvm() method to understand following slides better

```c
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(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");
    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) {
      kfree(mem);
      goto bad;
    }
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

CoW will be very similar to this

In vm.c

cow() —> Copy on write method

```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

Setup and check it does not fail

using this method

In vm.c

```c
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

Check this method and learn what it does.

You will use this a lot!

Hint: it has something to do with parent's pt

In vm.c

cow() —> Copy on write method

Use setupkvm()

Use walkpgdir()

Make parent’s pte shared and read only after finding it

Map child to parent’s page

Increment counter

Lastly, reinstall TLB Entries

```
lcr3(V2P(pgdir));
```

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

In vm.c

In Part 2, we will cover:

— pagefault in vm.c

— changes in proc.c - to call cow() method

— testcow.c implementation

STAY SAFE !!!