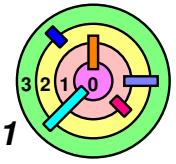
CS 350
PA5: Memory Management And Copy-On-Write

Bill Cheng

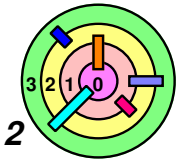
http://merlot.usc.edu/william/usc/

Based on slides created by Kivilcim Cumbul



PA5

- Implement an enhanced process details viewer
  - changes in "`proc.c`"
- Implement copy on write (COW)
  - changes in "`vm.c`" and "`trap.c`"
- Test your implementation



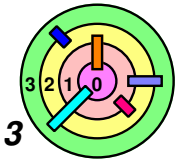
Preparation

- Read Ch 2 of the xv6 book regarding page tables
- Download xv6 for PA5
  - open a *terminal* and type the following

```
cd ~/cs350
mkdir pa5
cd pa5
wget --user=USERNAME --password=PASSWORD \
  http://merlot.usc.edu/cs350-m25/programming/pa5/xv6-pa5-src.tar.gz
tar xvf xv6-pa5-src.tar.gz
cd xv6-pa5-src
```

  - make sure you choose 1 CPU in your VM

```
CPUS := 1
```



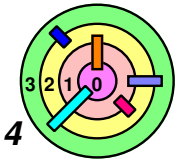
Submission

- Which files do you need to modify?
  - open a terminal and type the following:

```
pwd
cd ~/cs350/pa5/xv6-pa5-src
make -n pa5-submit
```

  - you should see:

```
tar cvzf pa5-submit.tar.gz \
        Makefile \
        pa5-README.txt \
        proc.c proc.h \
        trap.c \
        syscall.c syscall.h \
        sysproc.c \
        defs.h \
        user.h \
        usys.S \
        mmu.h \
        vm.c
```



Part 1: Enhanced Process Details Viewer

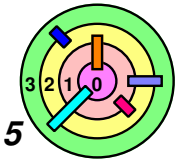
- Add a `procdump()` system call to print:

```
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 ...
1 -> 300, y
200 -> 500, n
2 sleep sh 80104907 80100966 80101d9e ...
1 -> 306, y
200 -> 500, n
```

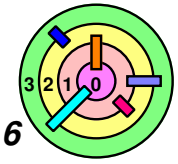
    - process 1 has 2 pages mapped
      - its vpn 1 is mapped *(writable)* to ppn 300
      - its vpn 200 is mapped *(read-only)* to ppn 500
    - the hex numbers after a process name are *return addresses* in the process's call stack (this part is already in `procdump()` in "`proc.c`")



Bits In A Page Table

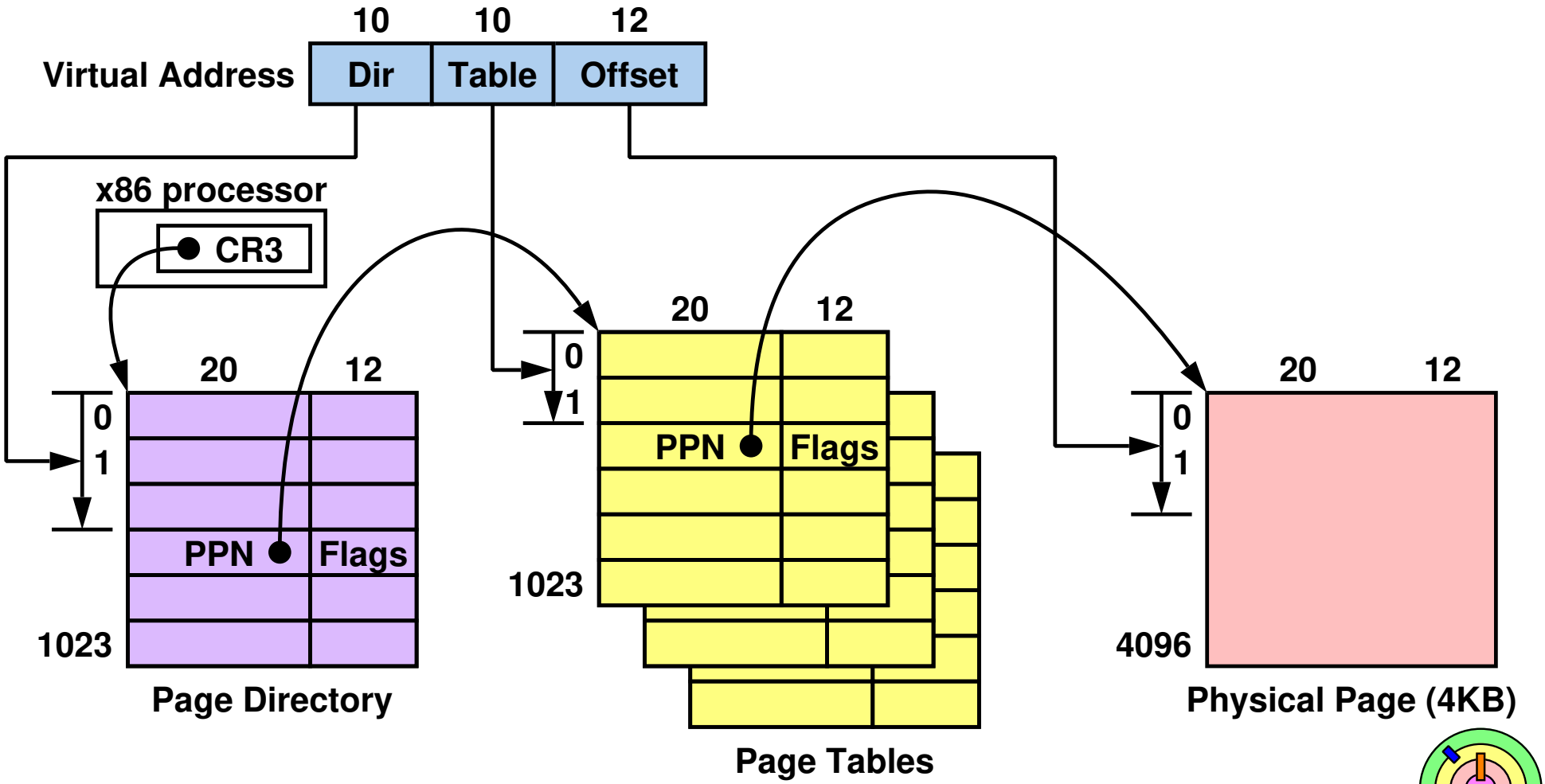
In "`mmu.h`", you can use constants and flags for page table

```
// 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, 1 for 4MB pages

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



XV6 Uses Two Levels Of Page Tables

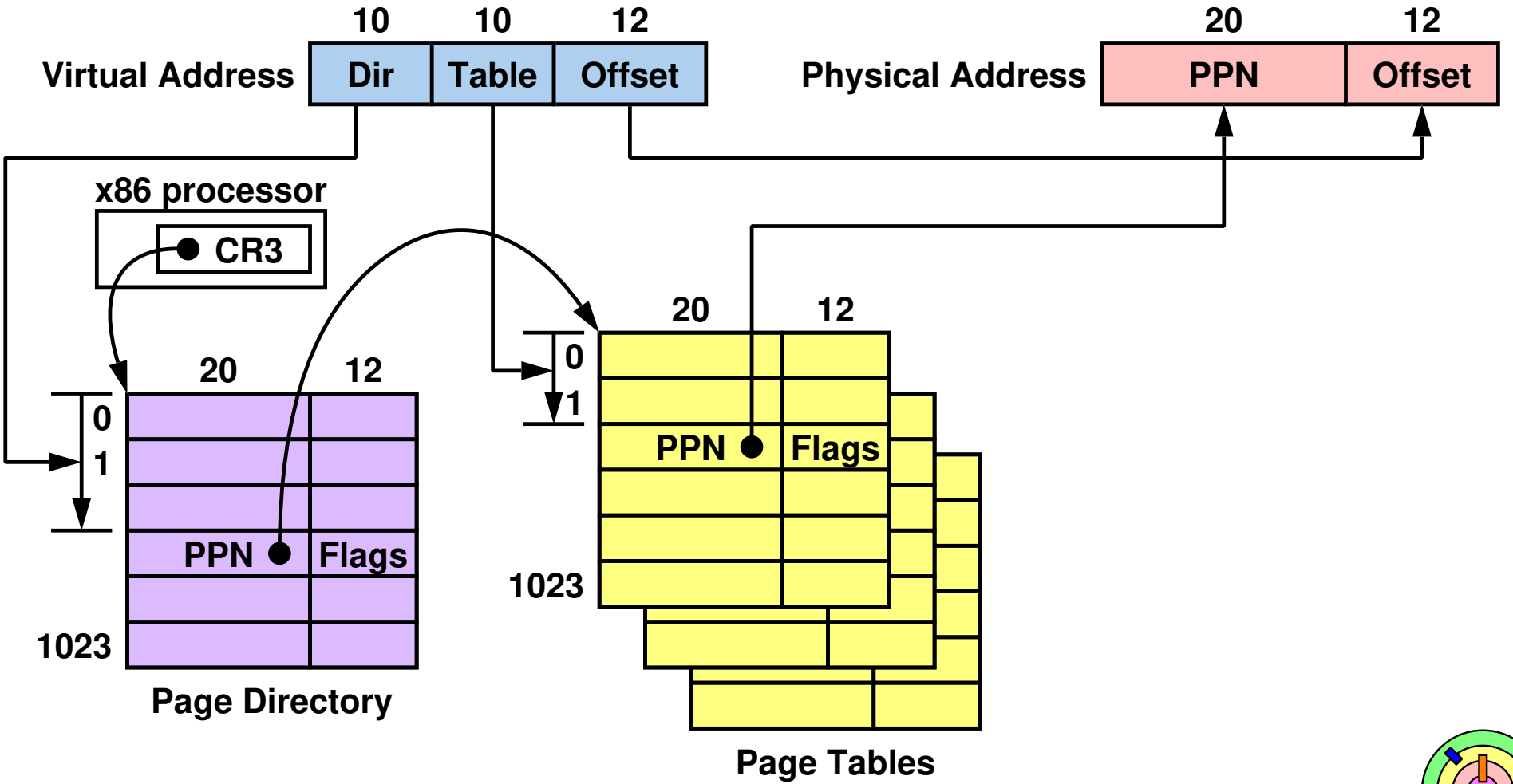
- First level page table is called a *page directory table*
  - entry in page directory table is called a *page directory entry (pde)*
  - entry in page table is called a *page table entry (pte)*



- every table and page is 4 KB in size!

XV6 Uses Two Levels Of Page Tables

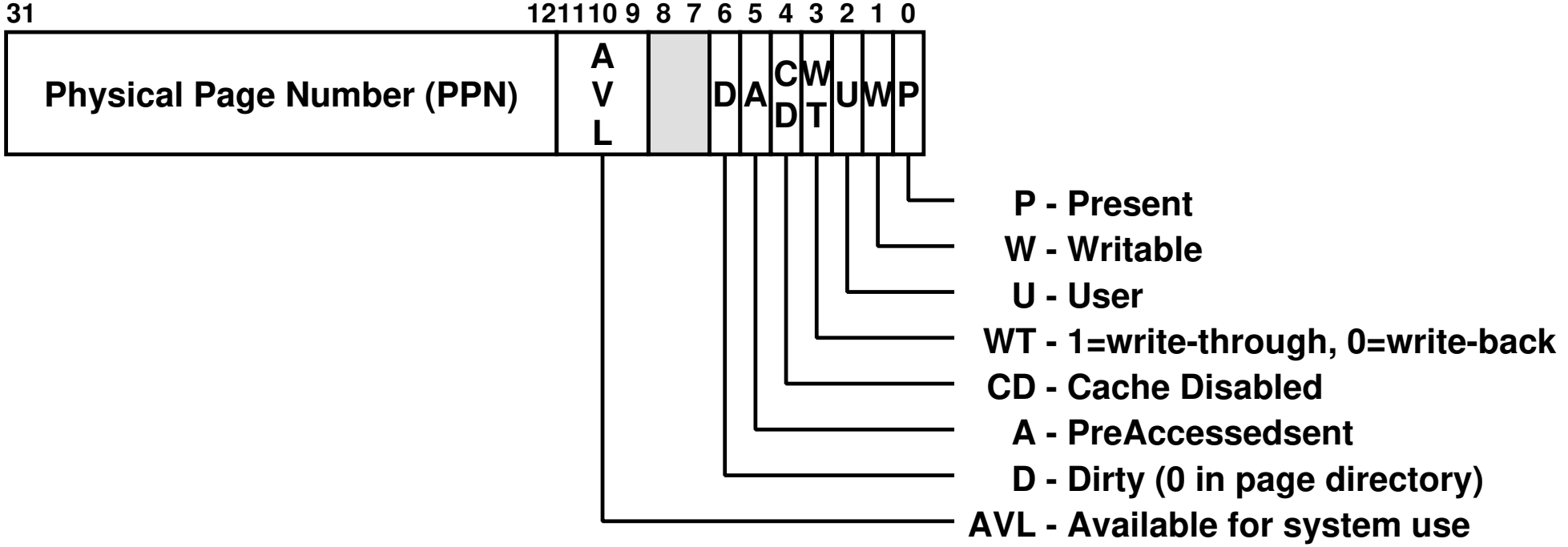
- First level page table is called a *page directory table*
  - entry in page directory table is called a *page directory entry (pde)*
  - entry in page table is called a *page table entry (pte)*



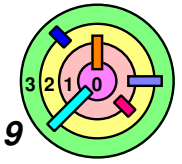
- everything is 4 KB in size!

Page Directory/Table Entry

Page directory entries and page table entries are identical except for the D bit (XV6 doesn't use all the bits)




```
// 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, 1 for 4MB pages

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

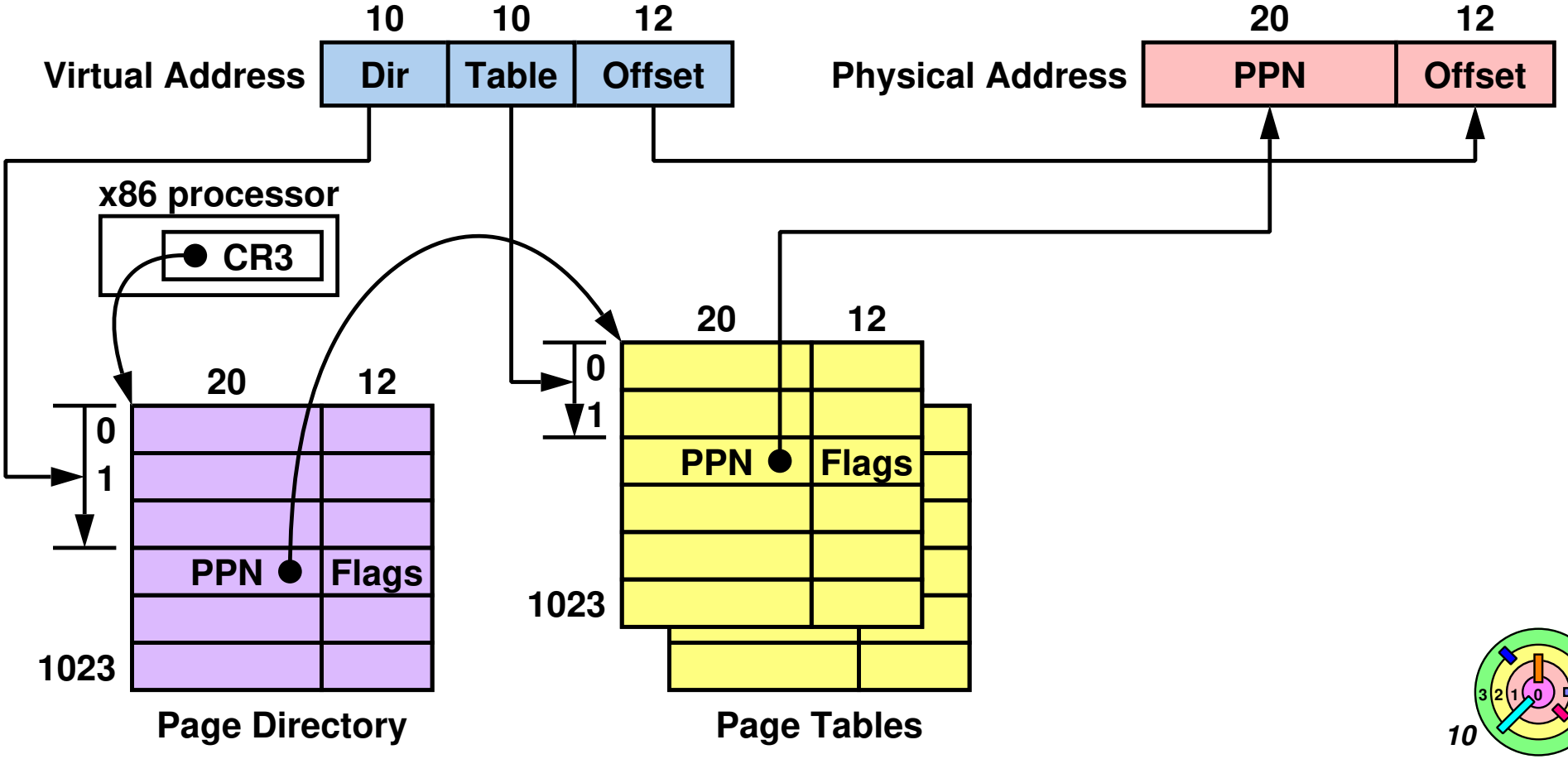


XV6 Uses Two Levels Of Page Tables

- Page directory in XV6 is referred to as the "page table"

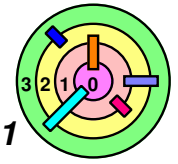
```
pde_t *pt = p->pgdir
```

- Leading 20 bits of a PDE gives the physical frame number of the physical page containing the 2nd level page table associated with that PDE



memlayout.h

```
// 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
```



Accessing Page Table

- 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
  - check if it is *present* and *user*

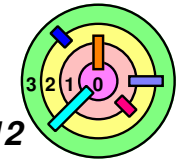
```
(pte & PTE_P) && (pte & PTE_U)
```

    - these are *bit masks:*

```
// 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, 1 for 4MB pages
```

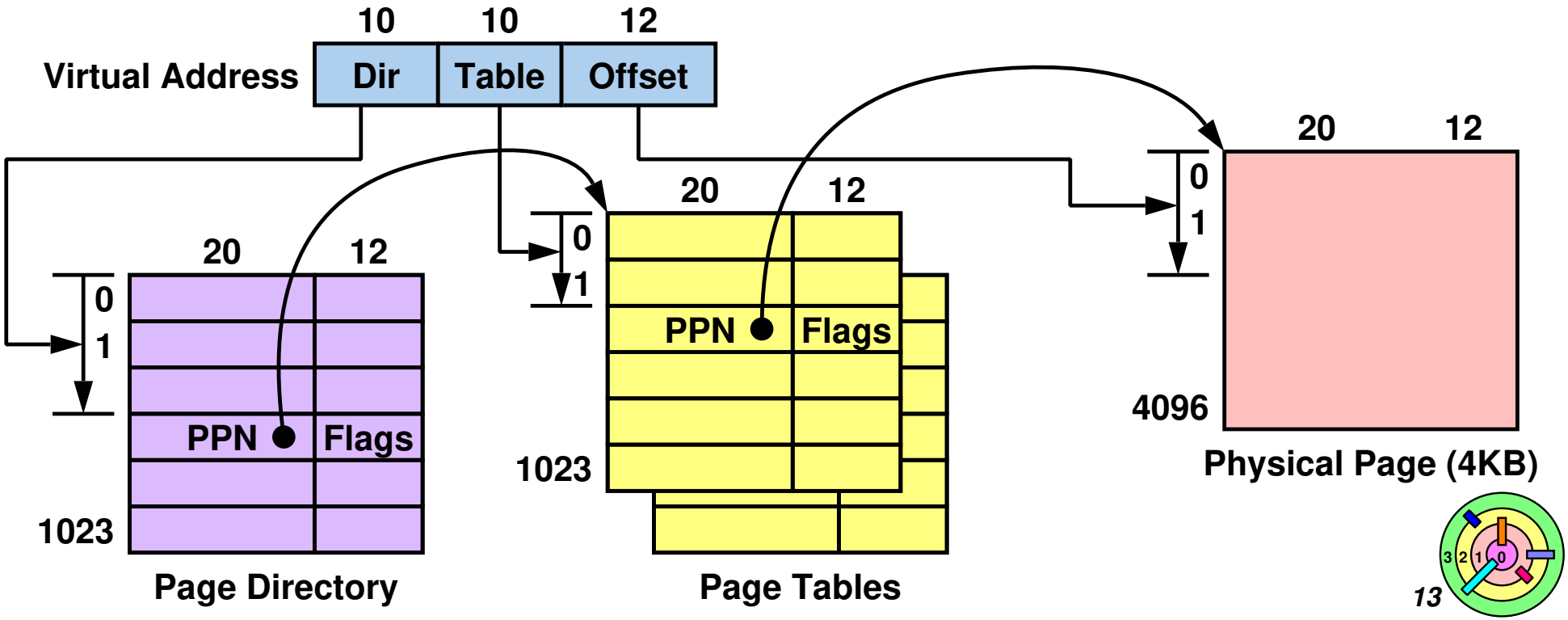
    - if present and user: find physical address (using bit masking and/or shifting)
  - check if writable and print "y" or "n"

```
(pte & PTE_W)
```



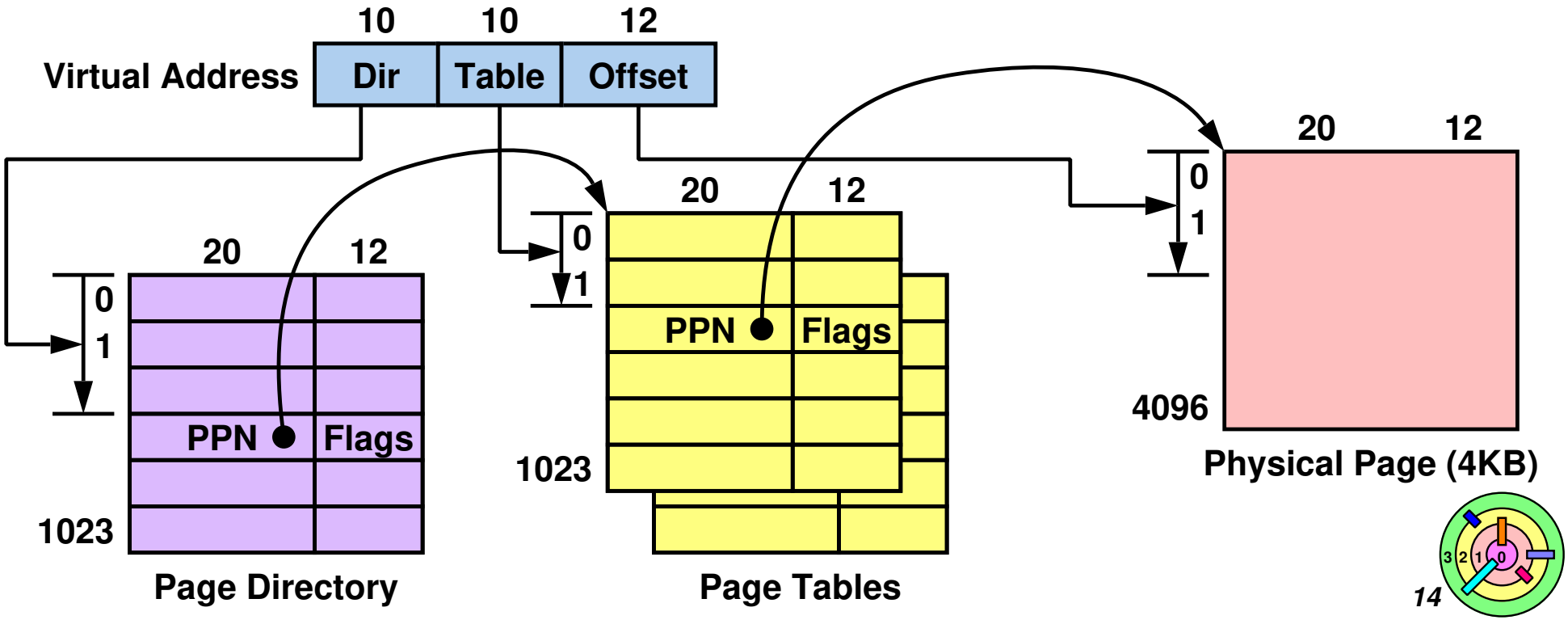
Part 2: Copy-On-Write (COW)

- Implement copy-on-write (COW) in `fork()`
  - changes in "`vm.c`"
- XV6, by default, would copy memory of all pages when `fork()` is called and we need to change it to use *copy-on-write*
  - on `fork()`, just copy page table so that child would *share* all the memory pages with the parent



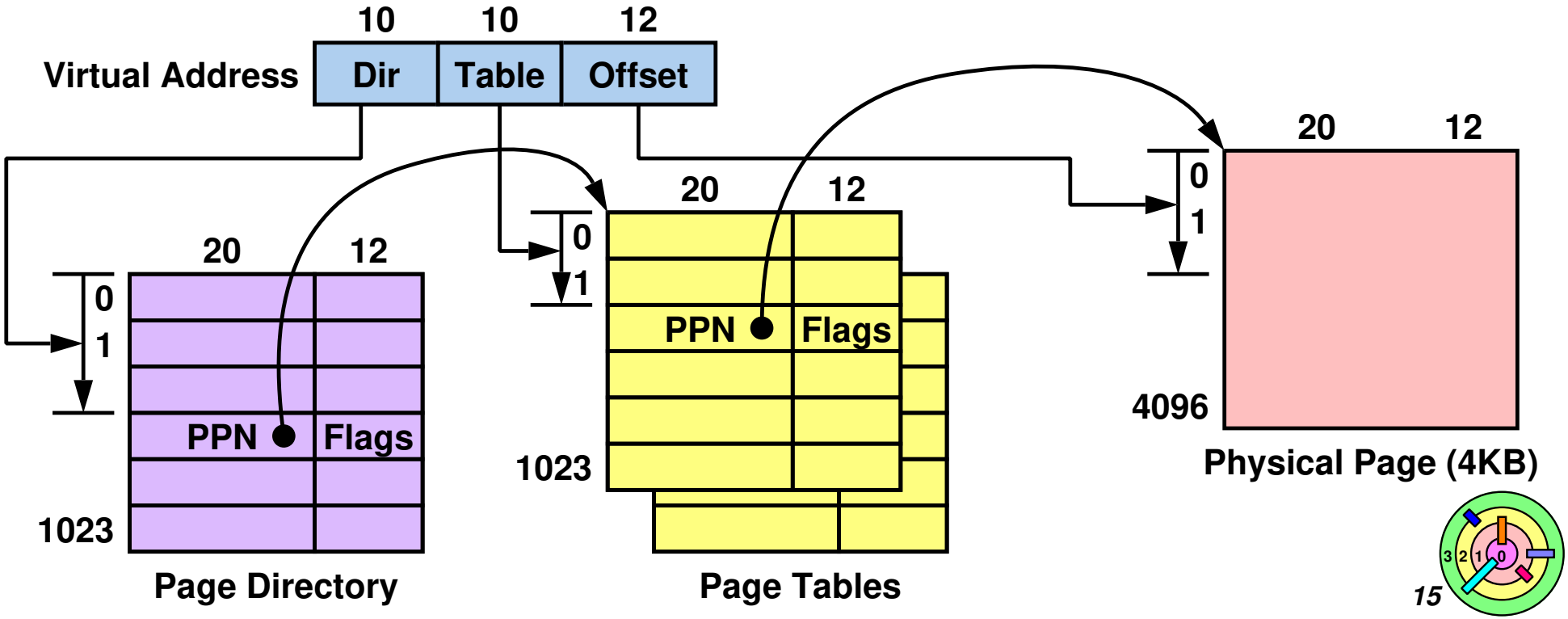
Part 2: Copy-On-Write (COW)

- on `fork()`, just copy page table so that child would *share* all the memory pages with the parent
- set `PTE_W` bit for all PTEs to 0 for both parent and child processes
- parent and child can read shared memory locations all they want



Part 2: Copy-On-Write (COW)

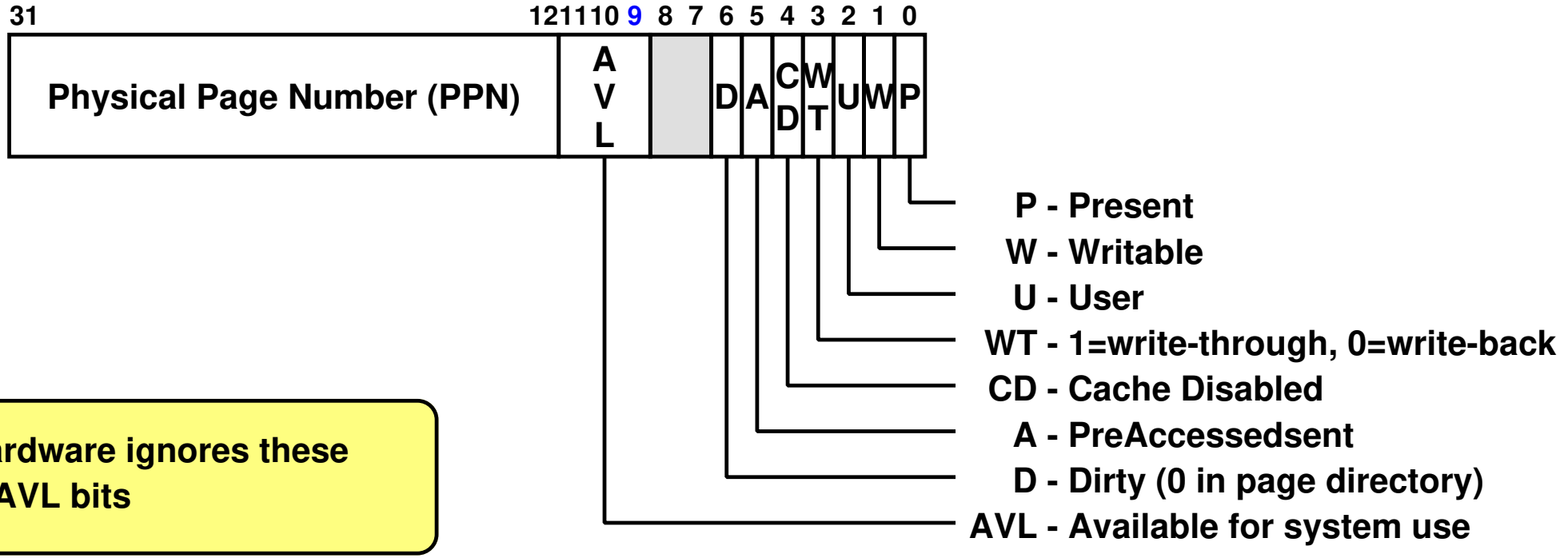
- when either parent or child wants to modify a shared memory location for the first time, a *page fault* will occur
  - page fault handler must make a copy of the shared page and update PTE (update PPN and set `PTE_W` to 1)
  - return from page fault handler to retry the operation that caused the page fault
  - writing to this page will no longer cause a page fault



Copy-On-Write (COW)

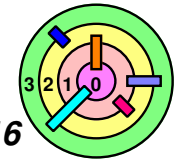
- Need to distinguish between a *shared read-only page (i.e., for the purpose of copy-on-write)* and a regular read-only page
  - add a bit in "`mmu.h`" to indicate if a page is shared or not (in every PTE)

```
#define PTE_SH          0x200   // 1 for shared
```



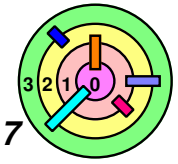
- hardware ignores these 3 AVL bits

  - `(pte & PTE_SH) && ((pte & PTE_W) != PTE_W)` means you need to copy-on-write



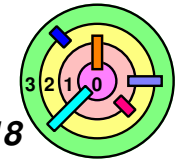
In "vm.c"

- If a page frame is shared, when can you "free" the page frame?
  - *reference counting:* needs to keep track of the number of processes that are sharing a page frame
    - since we can have at most 64 processes in XV6, an *8-bit counter per physical page* would work
  - in `wait()` system call, when a child process is freed, don't just free all the page frames it used blindly
    - do not deallocate shared pages (use the algorithm on the next slide)
- Hints
  - need a spinlock to access the counters
  - one counter per page frame: check PHYSTOP and PGSIZE to determine how many counters you need
  - initialize counters in `inituvm()` and `allocuvm()`



In "vm.c"

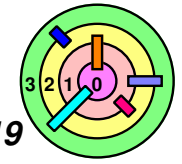
- In `deallocuvm()`:
  - if counter is 0: free 2nd-level page table (which is the original code)
  - if not: decrement counter and check if counter is 0 again
  - if counter is now 0, update shared and writable flag
    - only the parent process is using this page, so the page should be writable and not shared in the parent's page table



cow()

Code of `cow()` should be similar to the code of `copyuvm()`

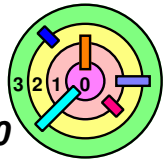
```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t* copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  char *mem;
  if((d = setupkvm()) == 0) return 0;
  for(uint i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) panic();
    if(!(*pte & PTE_P)) panic();
    uint pa = PTE_ADDR(*pte);
    uint 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()

`setupkvm()` creates a page directory table and set up the 2nd level page tables that the kernel uses

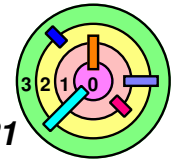
```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  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;
}
```



COW()

- `mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)` sets up mappings in `pgdir` starting with `PGROUNDDOWN(va)` to `pa` with permissions `perm`

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  pte_t *pte;
  char *a = (char*)PGROUNDDOWN((uint)va);
  char *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;
}
```

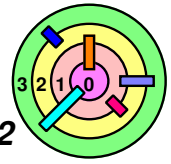


COW()

- `walkpgdir(pde_t *pgdir, void *va, int alloc)` returns the address of the PTE in *2nd-level page table* that corresponds to `va`
  - if cannot find and `alloc` is not 0, create a 2nd-level page table

```
// 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)
{
  pte_t *pgtab;
  pdt_t *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)];
}
```

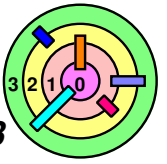
- `PDX(va)` is the first 10 bits of `va`, i.e., the *page directory index*



cow()

- `pde_t* cow(pde_t *pgdir, uint sz)` creates a copy of `pgdir` and returns it
  - calls `setupkvm()` to create and return a page directory table and set up the kernel portion of it
  - for every page, use `walkpgdir()` to locate the parent's PTE
    - set up parent's PTE for copy-on-write (i.e., shared and R/O)
    - map child to parent's page using `mappages()`
    - increment the refcount on the physical page (with the spinlock locked)
  - need to reinstall the parent's TLB entries by doing the following just before `cow()` returns:

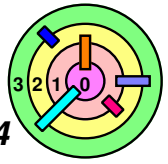
```
lcr3(V2P(pgdir));
```



Handle Page Fault

- Page fault handler in "`vm.c`" (let's call this `pagefault()`)
  1) get CR2 register
  2) check if the address found by `rcr2()` method is not 0
  3) find page table entry (PTE) by calling `walkpgdir()`
  4) if PTE is not shared, call `panic()`
  5) if PTE is not present, call `panic()`
  6) find physical address (`pa`)
  7) check if the page frame is being shared or not
     - 7.1) if shared, perform the "copy" part of copy-on-write
     - 7.2) if not shared, change PTE to be writable and not shared
  8) reinstall TLB entries by calling:

```
lcr3(V2P(myproc()->pgdir));
```



Handle Page Fault

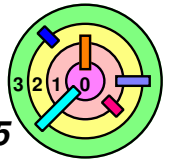
- Step 1: get CR2 register
  - CR2 register gives the virtual address that causes the page fault (https://wiki.osdev.org/CPU_Registers_x86#CR2)

bit	label	description
0-31	pfla	page fault linear address

  - x86 CPU has 4 control registers: `%cr0`, `%cr2`, `%cr3`, and `%cr4`
  - to get the page fault virtual address, do:

```
uint va = rcr2();
```

- Step 2: check if the address found by `rcr2()` method is not 0
  - if `va` is 0, we have a "segmentation fault" and we should terminate the user process
    - for PA5, you can just call `panic()` since this is not supposed to happen in PA5



Handle Page Fault

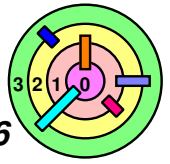
- Step 3: find page table entry (PTE) by calling `walkpgdir()`

```
pte_t *pte = walkpgdir(...);
```

- Step 4: if PTE is not shared, call `panic()`
  - in XV6, a page fault can only happen if 2 processes or more share the same PTE
- Step 5: if PTE is not present, call `panic()`
- Step 6: find physical address (`pa`)
  - `PTE_ADDR()` method in "`mmu.h`" to find the physical page number (`ppn`), which is the leading bits of the `pa`

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

  - copy the lowest 12 bits from the `va`
    - offset within a page stays the same

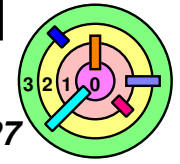


Handle Page Fault

- Step 7: check if the page frame is being shared or not
  - 7.1) if shared, perform the "copy" part of copy-on-write
    - allocate a new page by calling `kalloc()`
    - copy contents of page by calling `memmove()`
    - update pte so that `ppn` points to the newly allocated page (can use V2P() to get the `ppn`)

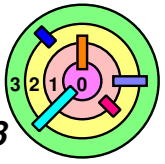
```
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
```

    - update pte so that flags contain the right values:
      - it should no longer be shared
      - it should now be writable
      - it should be present
      - it should be a user space entry
    - decrement the reference count of the original page frame since a pointer to that page frame has just been removed
  - 7.2) if not shared, change PTE to be writable and not shared



Additional Changes

- In `fork()`, instead of calling `copyuvm()`, call `cow()`
- In "`defs.h`", add definitions of `cow()` and `pagefault()` we created in "`vm.c`"
- In "`trap.c`", add call to `pagefault()` if trap number is `T_PGFLT`

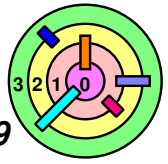


Part 3: Test Your Implementation

- `testcow-parent.c`: parent process doing copy-on-write

```
int main()
{
  int pid, i;
  int SIZE = 4096 + 1;
  char *space = (char *)malloc(SIZE);
  printf(1, "testcow-parent: space=%d\n", (unsigned int)space);
  procdump();
  if ((pid = fork()) == 0) {
    exit();
  } else {
    printf(1, "\nParent process before changing values\n\n");
    procdump();
    for (i = 0; i < SIZE; i++) {
      space[i]++;
    }
    printf(1, "\nParent process after changing values\n\n");
    procdump();
    wait();
  }
  free(space);
  exit();
  return 1;
}
```

  - need to see copy-on-write

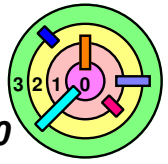


testcow-parent

Ex: running `testcow-parent`

```
testcow-parent: space=40952
1 sleep  init 80103db7 ...
0 -> 57210, y
2 -> 57206, y
2 sleep  sh 80103db7 ...
0 -> 57140, y
1 -> 57138, no
3 -> 57135, y
3 run    testcow-parent
0 -> 57060, y
2 -> 57057, y
3 -> 57276, y
4 -> 57277, y
5 -> 57279, y
6 -> 57280, y
7 -> 57281, y
8 -> 57282, y
9 -> 57283, y
10 -> 57284, y
```

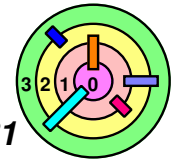
```
Parent process before changing values

1 sleep  init 80103db7 ...
0 -> 57210, y
2 -> 57206, y
2 sleep  sh 80103db7 ...
0 -> 57140, y
1 -> 57138, no
3 -> 57135, y
3 run    testcow-parent
0 -> 57060, no
2 -> 57134, y
3 -> 57276, no
...
10 -> 57284, no
4 zombie testcow-parent
0 -> 57060, no
2 -> 57057, y
3 -> 57276, no
...
10 -> 57284, no
```



testcow-parent

Ex: running testcow-parent

```
Parent process after changing values

1 sleep  init 80103db7 ...
0 -> 57210, y
2 -> 57206, y
2 sleep  sh 80103db7 ...
0 -> 57140, y
1 -> 57138, no
3 -> 57135, y
3 run    testcow-parent
0 -> 57060, no
2 -> 57134, y
3 -> 57276, no
...
8 -> 57282, no
9 -> 57056, y
10 -> 57055, y
4 zombie testcow-parent
0 -> 57060, no
2 -> 57057, y
3 -> 57276, no
...
10 -> 57284, no
```

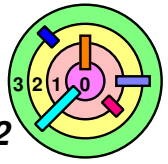


testcow-child

testcow-child.c: child process doing copy-on-write

```
int main()
{
  int pid, i;
  int SIZE = 4096 + 1;
  char *space = (char *)malloc(SIZE);
  printf(1, "testcow-child: space=%d\n", (unsigned int)space);
  procdump();
  if ((pid = fork()) == 0) {
    printf(1, "\nChild process before changing values\n\n");
    procdump();
    for (i = 0; i < SIZE; i++) {
      space[i]++;
    }
    printf(1, "\nChild process after changing values\n\n");
    procdump();
    wait();
  } else {
    exit();
  }
  free(space);
  exit();
  return 1;
}
```

- need to see copy-on-write

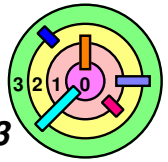


testcow-child

Ex: running `testcow-child`

```
testcow-child: space=40952
1 sleep  init 80103db7 ...
0 -> 57210, y
2 -> 57206, y
2 sleep  sh 80103db7 ...
0 -> 57140, y
1 -> 57138, no
3 -> 57135, y
3 run    testcow-child
0 -> 57060, y
2 -> 57057, y
3 -> 57276, y
4 -> 57277, y
5 -> 57279, y
6 -> 57280, y
7 -> 57281, y
8 -> 57282, y
9 -> 57283, y
10 -> 57284, y
```

```
Child process before changing values

1 sleep  init 80103db7 ...
0 -> 57210, y
2 -> 57206, y
2 sleep  sh 80103db7 ...
0 -> 57140, y
1 -> 57138, no
3 -> 57135, y
3 sleep  testcow-child ...
0 -> 57060, no
2 -> 57134, y
3 -> 57276, no
...
10 -> 57284, no
4 run    testcow-child
0 -> 57060, no
2 -> 57057, y
3 -> 57276, no
...
10 -> 57284, no
```



testcow-child

Ex: running `testcow-child`

```
Child process after changing values

1 sleep  init 80103db7 ...
0 -> 57210, y
2 -> 57206, y
2 sleep  sh 80103db7 ...
0 -> 57140, y
1 -> 57138, no
3 -> 57135, y
3 sleep  testcow-child 80103db7 ...
0 -> 57060, no
2 -> 57134, y
3 -> 57276, no
...
10 -> 57284, no
4 run    testcow-child
0 -> 57060, no
2 -> 57057, y
3 -> 57276, no
...
8 -> 57282, no
9 -> 57056, y
10 -> 57055, y
```