CSCI 350 Project 4

Part 2

Kivilcim Cumbul

Task 2 cont’d: Page Fault method

Page Fault method in vm.c

1- Get cr2 address

2- Check if the address found by rcr2() method is not 0.

3- Find page table entry (PTE) –> hint: use walkpgdir method

4-If pte is not shared –> give panic error → It must be shared to get a pagefault as described in the project 4 document

Use flags we discussed in part 1 to check shared and present

5-If pte is not present –> give panic error → If it is not present, it is invalid.

6-Find physical address(pa) and 20-bit physical page number (ppn)

7- Check if the table is shared or not → hint: you have to check it using counter in this stage

8 - If shared –> look next slides

9- If not shared –> make table make table writable and not shared using flags

10- Reinstall TLB Entries → `lcr3(V2P(myproc()->pgdir));`

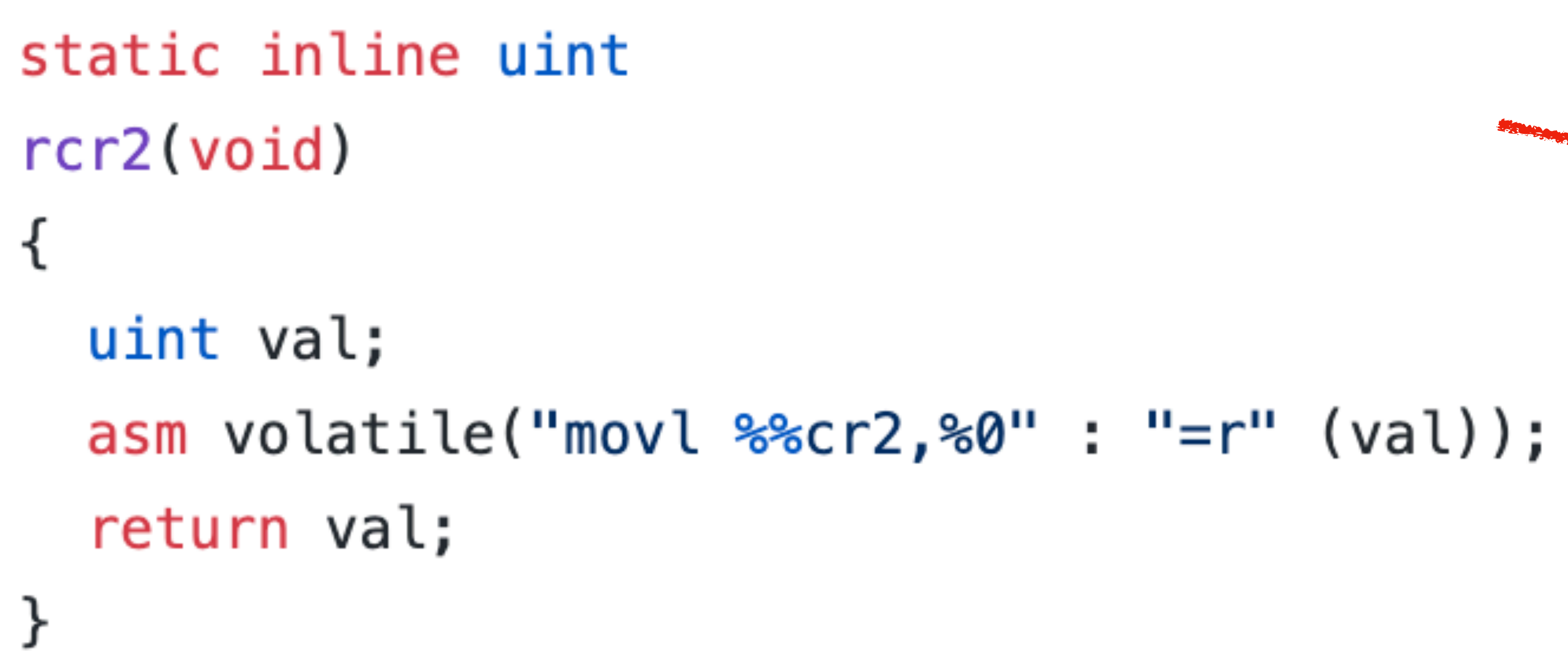
Step by step explanation

1- Get cr2 address

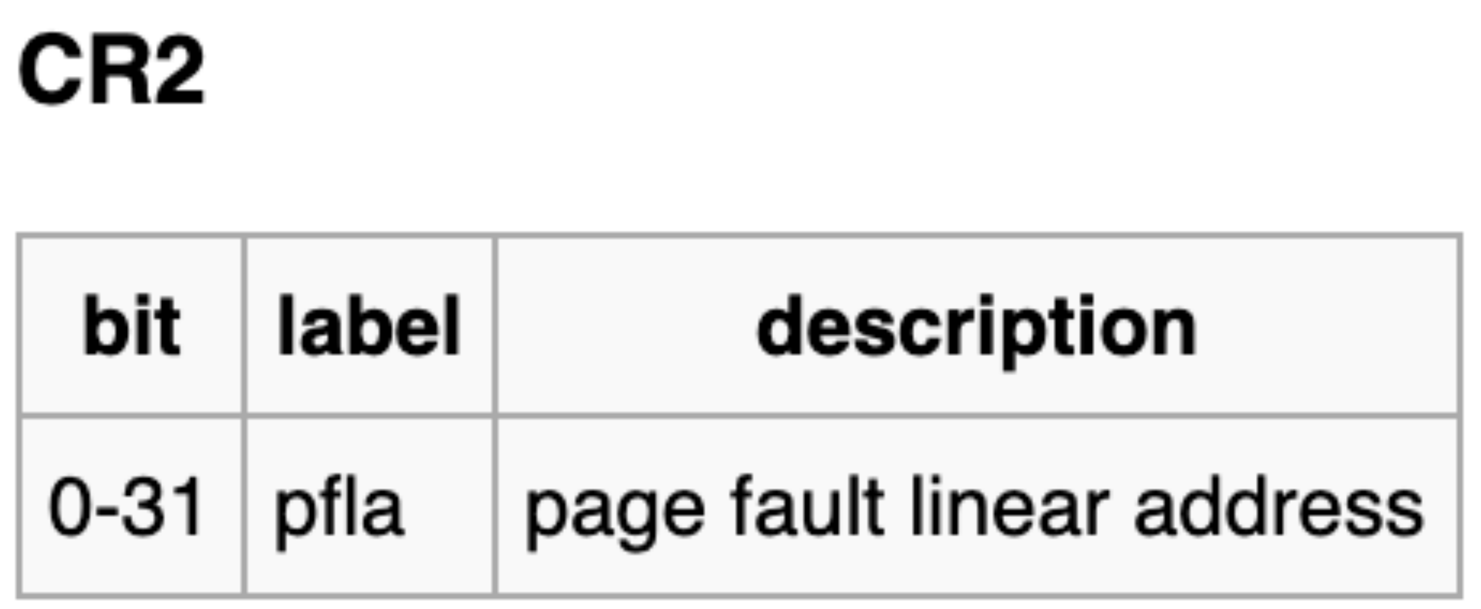
There are 4 control registers:%cr0, %cr2, %cr3, and %cr4

We will get the address from cr2 register —> using rcr2() method

The method is defined in x86.h



```
static inline uint
rcr2(void)
{
  uint val;
  asm volatile("movl %%cr2,%0" : "=r" (val));
  return val;
}
```



CR2

bit	label	description
0-31	pfla	page fault linear address

In xv6, cr2 register gives the virtual address that causes the page fault.

Check this for more info: https://wiki.osdev.org/CPU_Registers_x86#CR2

2- Check if the address found by rcr2() method is not null.

Check if the address is not 0.

This is important because we have to make sure the address we found is not null.

If it is null we have to throw null pointer exception or indicate it is null pointer using panic

Terminate the current process, and exit

3- Find page table entry (PTE)

```
// 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)];
}
```

Use walkpgdir method

4-Check if pte is shared or not

If pte is shared, that means we are good to go, because 2 processes or more share the same page table entry which indicates a pagefault can be thrown.

If pte is not shared, the page fault cannot be thrown in the first place, so we should give an error message

To check, use the shared flag you created in the first part of project 4.

Use panic to throw error.

4-Check if pte is present or not

If pte is present, we can continue.

But if pte is not present, the page fault cannot be thrown in the first place, so we should give an error message.

To check, use the PTE_P flag you learned in first part of the project

Use panic to throw error.

4-Find physical address(pa) and 20-bit physical page number (ppn)

Physical address (pa) is an address that the processor chip sends to main memory.

You can use PTE_ADDR() method in mmu.h to find the pa

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

Each PTE contains a 20-bit physical page number (PPN) and some flags. The paging hardware translates a virtual address by using its top 20 bits to index into the page table to find a PTE, and replacing the address's top 20 bits with the PPN in the PTE. The paging hardware copies the low 12 bits unchanged from the virtual to the translated physical address.

You can find ppn using shifting and/or masking

This step can be skipped in some implementations — depending on how you implemented the counter.

7- Check if the table is shared or not

This time, we will use counter we created earlier to check if there are still process that shared this table.

Everyone can implement the counter differently, so there is no specific way to do this part

pa and/or ppn can be used here

8- If shared

8.1 - create a new page table

8.2 - copy contents of page table we have

8.3 - change flags (user, present, shared flags)

8.4 - insert the new physical page num

8.5 - Decrease the counter

8.1 - create a new page table

8.2 - copy contents of page table we have

Hint : you might want to check these methods

kalloc() method in kalloc.c —> create

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

memmove() method in string.c —> copy

```
void*
memmove(void *dst, const void *src, uint n)
{
  const char *s;
  char *d;

  s = src;
  d = dst;
  if(s < d && s + n > d){
    s += n;
    d += n;
    while(n-- > 0)
      *--d = *--s;
  } else
    while(n-- > 0)
      *d++ = *s++;

  return dst;
}
```

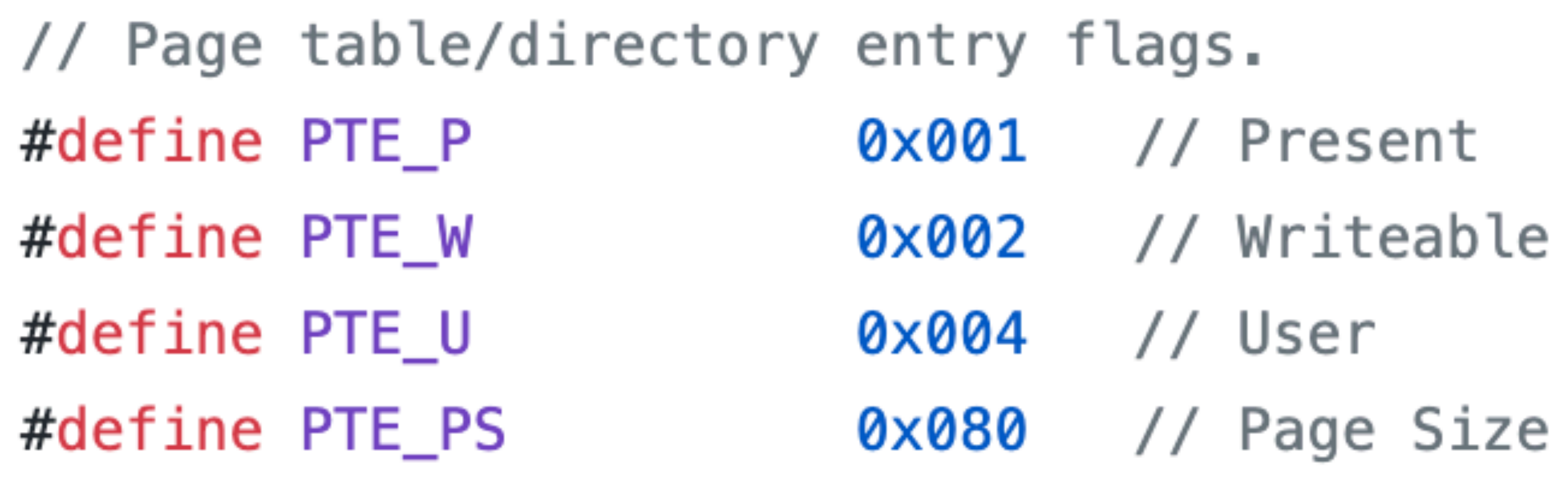
8.3 - change flags (user, writable, shared flags)

You have to do this stage because after creating a new page table and copying contents, the process will have a new and its own page table, this means:

-It is not shared

-It is writable

-It is user and present



```
// 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
```

8.4 - insert the new physical page num

We need to set first 20 bits of the pte to the physical address of the new page table

This means we have to use either one of these methods: P2V or V2P. Choose the proper one.

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

8.5 - Decrease the counter

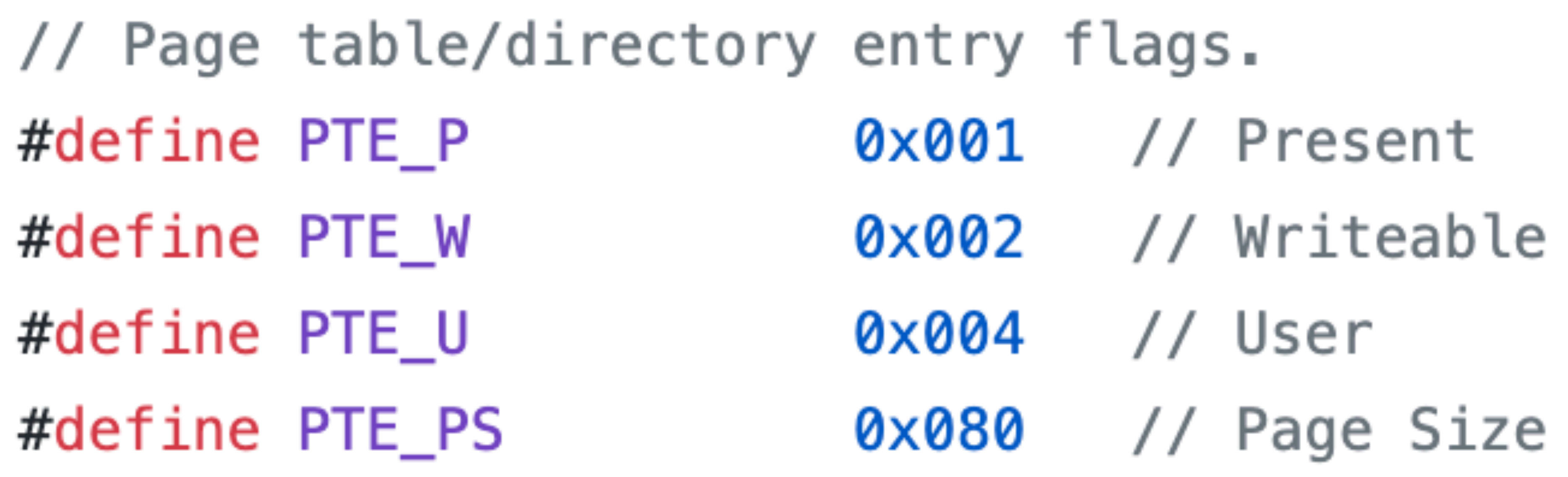
Now, because we throw a pagefault interrupt, the tables of the child and parent got separated:

This means, we have to decrease the counter which we implemented earlier to store how many processes are sharing the parent's page table.

The counter implementation can be different for every student, so there is no specific way to decrement it.

9- If not shared:

Use flags to make the table not shared and writable



```
// 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
```

And the flag you created

This is the end of the pagefault method discussion,

but don’t forget to reinstall TLB Entries!

Task 2 cont’d: changes in proc.c

Now, instead of copyuvm() method, you have to call cow() method while **forking.**

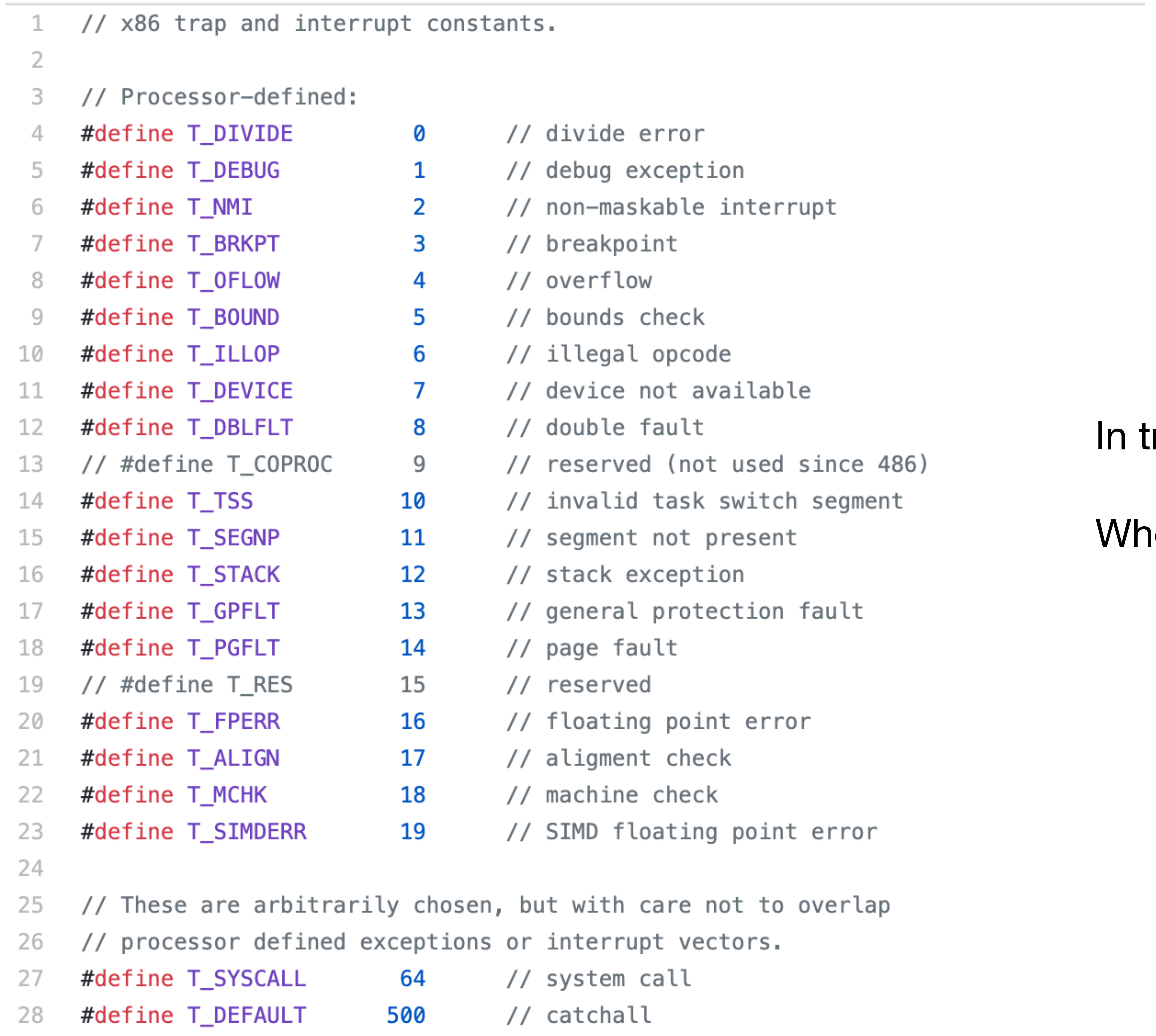
Task 2 cont’d: changes in defs.h

Add definitions for cow() and pagefault() methods we created in vm.c

to defs.h

Task 2 cont’d: changes in trap.c

Add pagefault as a new type of trap in trap.c —> define new case

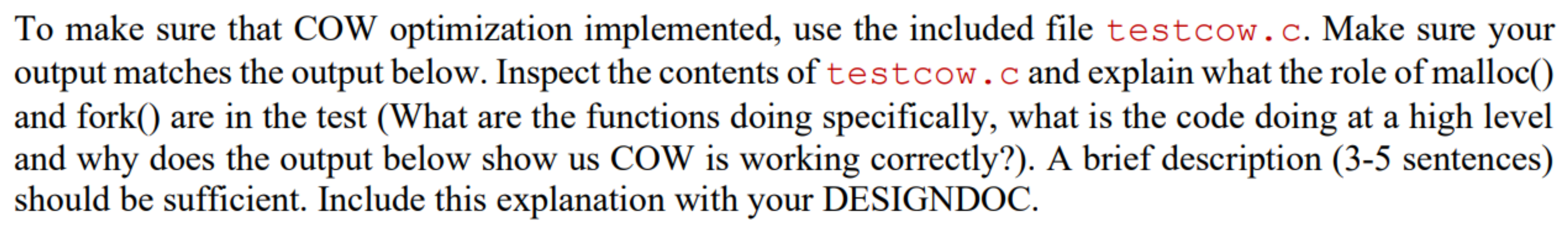


```c
// x86 trap and interrupt constants.

// Processor-defined:
#define T_DIVIDE         0      // divide error
#define T_DEBUG          1      // debug exception
#define T_NMI            2      // non-maskable interrupt
#define T_BRKPT          3      // breakpoint
#define T_OFLOW          4      // overflow
#define T_BOUND          5      // bounds check
#define T_ILLOP          6      // illegal opcode
#define T_DEVICE         7      // device not available
#define T_DBLFLT         8      // double fault
// #define T_COPROC      9      // reserved (not used since 486)
#define T_TSS           10      // invalid task switch segment
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
// #define T_RES        15      // reserved
#define T_FPERR         16      // floating point error
#define T_ALIGN         17      // aligment check
#define T_MCHK          18      // machine check
#define T_SIMDERR       19      // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64      // system call
#define T_DEFAULT      500      // catchall
```

In traps.h —> T_PGFLT(pagefault) flag is already defined:

When this case occurs pagefault method will be raised

Task 3: testcow.c



To make sure that COW optimization implemented, use the included file testcow.c. Make sure your output matches the output below. Inspect the contents of testcow.c and explain what the role of malloc() and fork() are in the test (What are the functions doing specifically, what is the code doing at a high level and why does the output below show us COW is working correctly?). A brief description (3-5 sentences) should be sufficient. Include this explanation with your DESIGNDOC.

The test can be different for each student but make sure to have these:

create values in different pages using malloc

Print parent info using procdump()

Fork to create new child processes

procdump() to print info of child

Update **something** in child process $\longrightarrow$ Find it out!

procdump() to print info of child

That’s all for project 4!

Don’t forget to add explanations about testcow.c to your DESIGNDOC.

Stay safe !!!