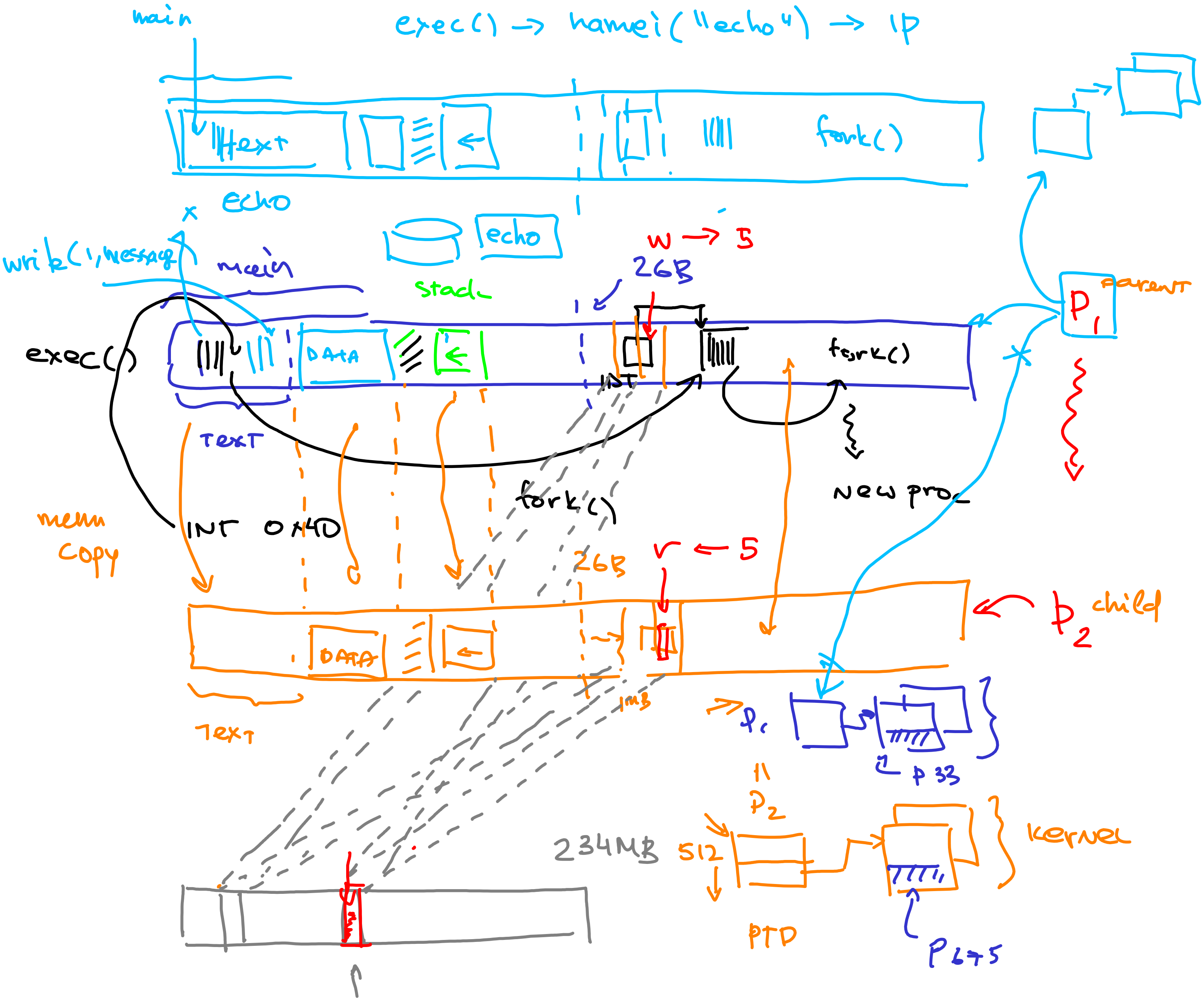
CS 238P
Fall 2018
Midterm
11/15/2018
Time Limit: 3:30pm - 4:50pm

Name (Print):

- **Don't forget to write your name on this exam.**
- **This is an open book, open notes exam. But no online or in-class chatting.**
- **Ask us if something is confusing in the questions.**
- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive full credit**. A correct answer, unsupported by explanation will receive no credit; an incorrect answer supported by substantially correct explanations might still receive partial credit.
- If you need more space, use the back of the pages; clearly indicate when you have done this.

Problem	Points	Score
1	15	
2	5	
3	10	
4	14	
5	10	
6	5	
7	14	
8	3	
Total:	76	



process_ID, page, kstack . . .

EAX = ID of a child

EAX = Φ

1. OS Interfaces

   (a) (5 points) Heres a program that uses the UNIX system call API, as described in Chapter 0 of the xv6 book:

```
#include "param.h"
#include "types.h"
#include "user.h"
#include "syscall.h"

int main() {
  char * message = "aaa\n";
  int pid = fork();

  if(pid != 0){

    char *echoargv[] = { "echo", "Hello\n", 0 };

    message = "bbb\n";
    exec("echo", echoargv);
    write(1, message, 4);

  }

  write(1, message, 4);
  exit();
}
```

Assume that fork() succeeds, that file descriptor 1 is connected to the terminal when the program starts, and echo program exists. What possible outputs this program can produce (explain your answer)?

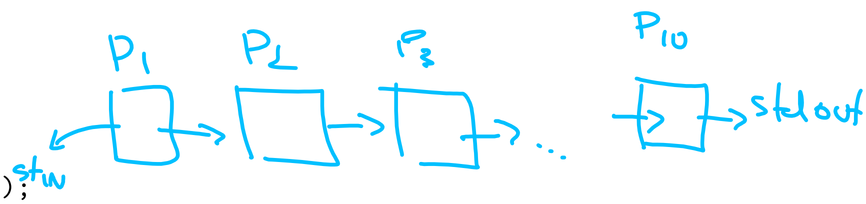
Solution:

```
aaa
Hello
or
Hello
aaa
```

There are two processes after fork succeeds. The parent process runs the exec inside the if block, without running anything after that exec statement. The child process executes the write statement outside the if block and exits. The order is non-deterministic.

(b) (10 points) Write a program that uses the UNIX system call API, as described in Chapter 0 of the xv6 book. The program forks and creates a pipeline of 10 stages. I.e., each stage is a separate process. Each two consequtive stages are connected with a pipe, i.e., the standard output of each stage is connected to the standard input of the next stage. Each stage reads a character from its standard input and sends it to the standard output. The last stage outputs the character it reads from the pipe to the standard output.

Solution:




```
int main(){
  int count = 0;
  int fd[2];
  pid_t pid; //typedef short
  pipe(fd);
  create_child_proc(int count);
}
void create_child_proc(int count){
  char * message "i";
  pid = fork();
  if (pid==0){ // child
    close(fd[1]);
    close(0);
    dup(fd[0]);
    close(fd[0]);
    if(count<8){
      count++;
      create_child_proc(count);
    }
  }
  else if (pid!=0){ // parent
    close(fd[0]);
    close(1);
    dup(fd[1]);
    close(fd[1]);
    read(0,message,1);
    write(1,message,1);
  }
}
```

2. Basic page tables.

   (a) (5 points) Alice wants to construct a page table that maps virtual addresses 0x8000000, 0x80001000 and 0x80002000 into physical addresses 0x0, 0x1000, and 0x2000. Assume that the Page Directory Page is at physical address 0x0, and the Page Table Page is at physical address 0x00001000 (which is PPN 0x00001).

   Draw a picture of the page table Alice will construct (or alternatively simply write it down in the format similar to the one below): :

   Page Directory Page:

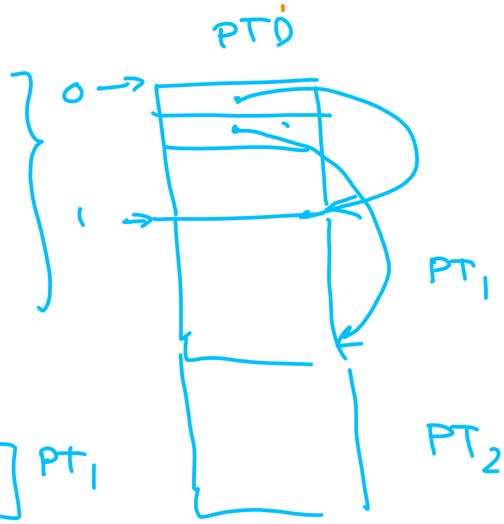
   ```
   PDE 0: PPN=0x1, PTE_P, PTE_U, PTE_W
   PDE 1: PPN=0x2, PTE_P, PTE_U, PTE_W
   ```

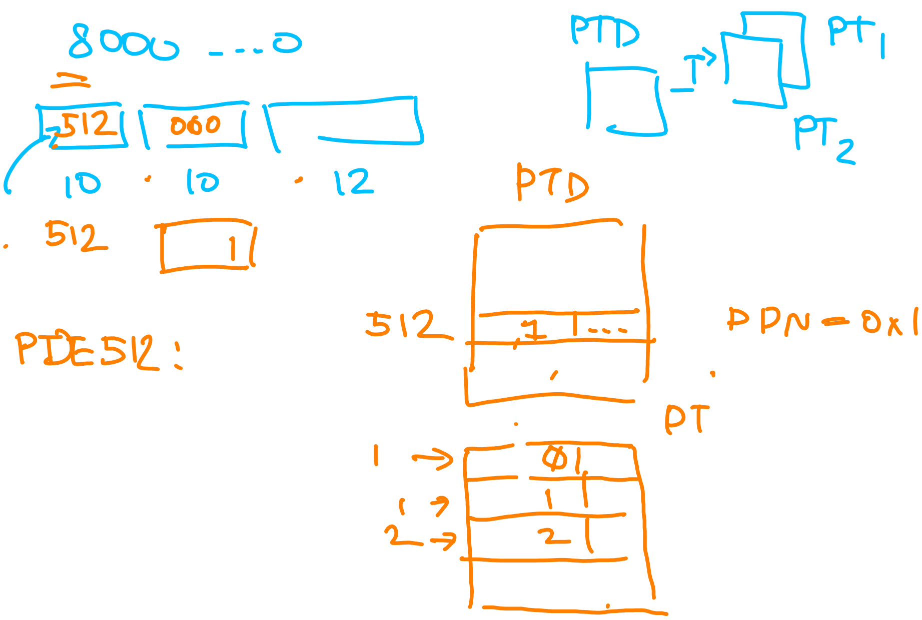
   ... all other PDEs are zero

   The Page Table Page:

   ```
   PTE 0: PPN=0x3, PTE_P, PTE_U, PTE_W
   PTE 1: PPN=0x4, PTE_P, PTE_U, PTE_W
   ```

   ... all other PTEs are zero

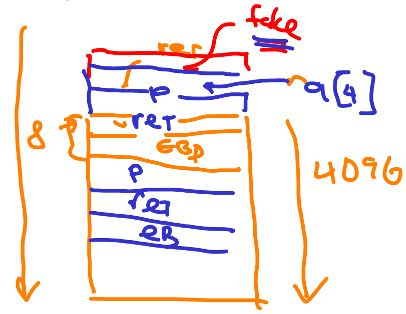




3. Stack and calling conventions.

Alice developed an xv6 program that has a function `foo()` that is called directly from `main()`:

```
int foo(char *p) {
  write(1, "hello\n", 6);
  foo(p);
  return 0;
}

int main() {
  char  a[4];
  foo(a);
  exit();
}
```



(a) (5 points) How many times will she see “hello“ on her screen? Justify your answer.

Solution: Different answers are acceptable for this question as long as the justification is right and account for how data is pushed in the stack. Here are some samples.

Solution 1 (does not account all factors but can receive full points): foo is called recursively, and will be called until the stack is full. Every invocation of foo will insert a return address in the stack. Each return address is of size 4 bytes. A stack page for each process is 4 kilo bytes. Hence there would be approximately 4*1024/4 = 1024 hello prints.

Solution 2 (adds on the above): Based on how your compiler is designed, your compiler may also maintain frames. Hence we might be pushing an extra 4 bytes for EBP on to the stack. Hence there would be approximately 4*1024/8 = 512 hello prints.

Solution 3 (further adding to the above, but all the above are acceptable): Also account for the data pushed for the main function into the stack, this will reduce the number of prints by 2 (one reduction for main’s fake return address (4 bytes), and one for the char local var (4 bytes))

(b) (5 points) Now Alice changes her `main()` function like this

```
static char b[8192];

int foo(char *p) {
  write(1, "hello\n", 6);
  foo(p);
  return 0;
}

int main() {
  char  a[8192]
  foo(a);
  write(1, b, 10);
  exit();
}
```

She runs it again. How many times will she see "hello" on the screen (justify your answer).

Solution (again in this question we'll be looking at your justification - in particular how you would go about calculating the number of prints, rather than the number of prints itself. There is no one acceptable answer): The first invocation would jump over the guard page of the stack, and start writing into the text and data section, and the recursion will continue from there. Now we need to reason about how big is the text + data section. After getting the size, divide this size by any of the numbers as reasoned in the part (a) answers of this question. This would give you the number of "hello" prints, since the text and data sections are executable.

Reasoning about the size of text+data section. Since programs are of different sizes, xv6 dynamically allocates pages for programs. For a small program xv6 can assign 1 page, so for this program we can assume to use 1 page for the text section. Now for the variable, a, we would need an extra 2 pages. So we would need at least 2 pages for the data section. So at least 3 pages would be used for the text+data section.

Refer https://uci.yuja.com/V/Video?v=258979&node=1217208&a=95671235&autoplay=1 (time 15:20)

4. Xv6 process organization.

In xv6, in the address space of the process what does the following virtual addresses contain?

(a) (3 points) Virtual address 0x0

Solution: Text section

(b) (3 points) Virtual address 0x80100000

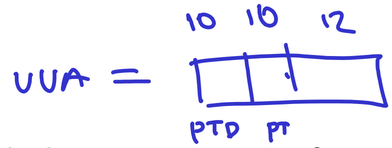
Solution: Kernel section

(c) (3 points) What physical address is mapped at virtual address 0x80000000

Solution:0x0

(d) (5 points) Bob looks at the implemenation of the `clearpteu()` function in the xv6 kernel (see below). He is confused about the role of the `walkpgdir()` function.

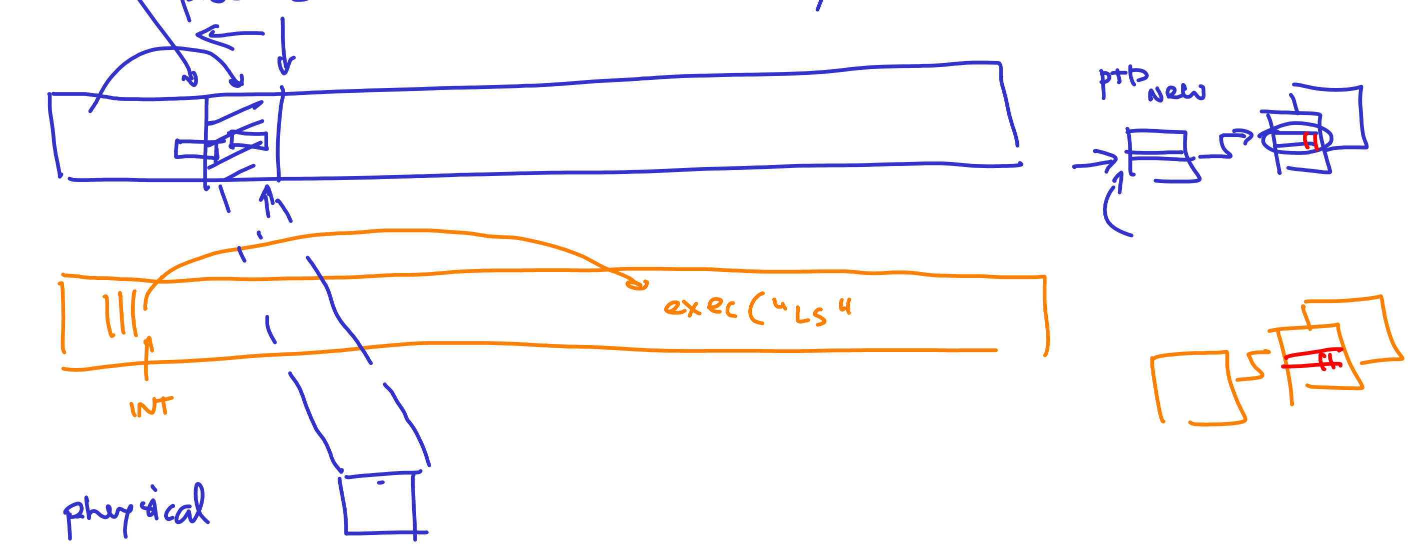
```
void
clearpteu(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if(pte == 0)
    panic("clearpteu");
  *pte &= ~PTE_U;
}
```



Can you explain Bob why `walkpgdir()` is needed here and what purpose it serves?

Solution:

The 3 args of walkpgdir: pgdir - page table directory, the table to be walked, uva - virtual address, 0 - flag to indicate whether to allocate additional entries. The function splits the virtual address in 3 parts (10 bits, 10 bits, 12 bits) and walks the page table pointed by pgdir, and returns the page table entry that maps this address.

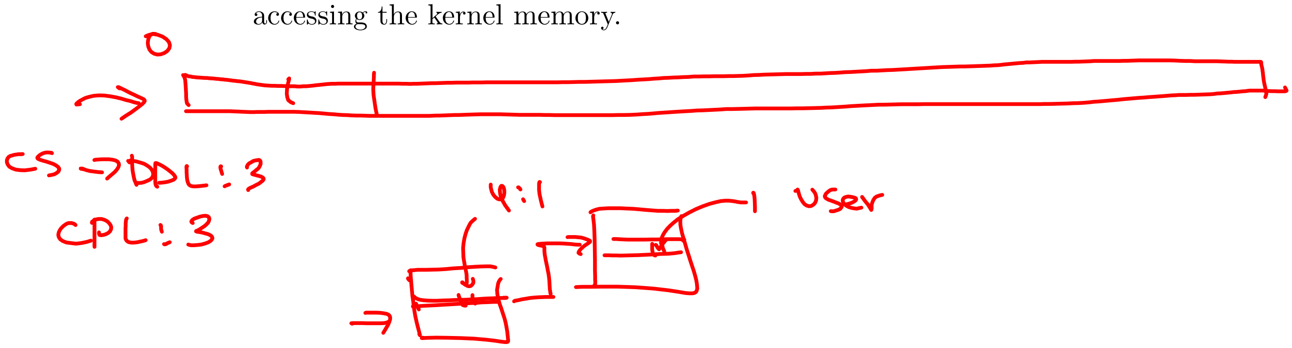


5. Protection and isolation

(a) (5 points) In xv6 all segments are configured to have the base of 0 and limit of 4GBs, which means that segmentation does not prevent user programs from accessing kernel memory. Nevertheless, user programs can't read and write kernel memory. How (through what mechanisms) such isolation is achieved?

Solution:

Segments themselves don't limit your memory. However, segments encode the privilege level. For example, segments for a user level program would indicate that the program runs at privilege level 3 (by activating the user accessible bit). This user-accessible bit would be used in page translation. Then, if the page being mapped is mapped as user accessible, the translation will go through, else the translation is aborted. Conversely, all pages of the kernel are mapped with the user bit unset, which prevents the user from accessing the kernel memory.

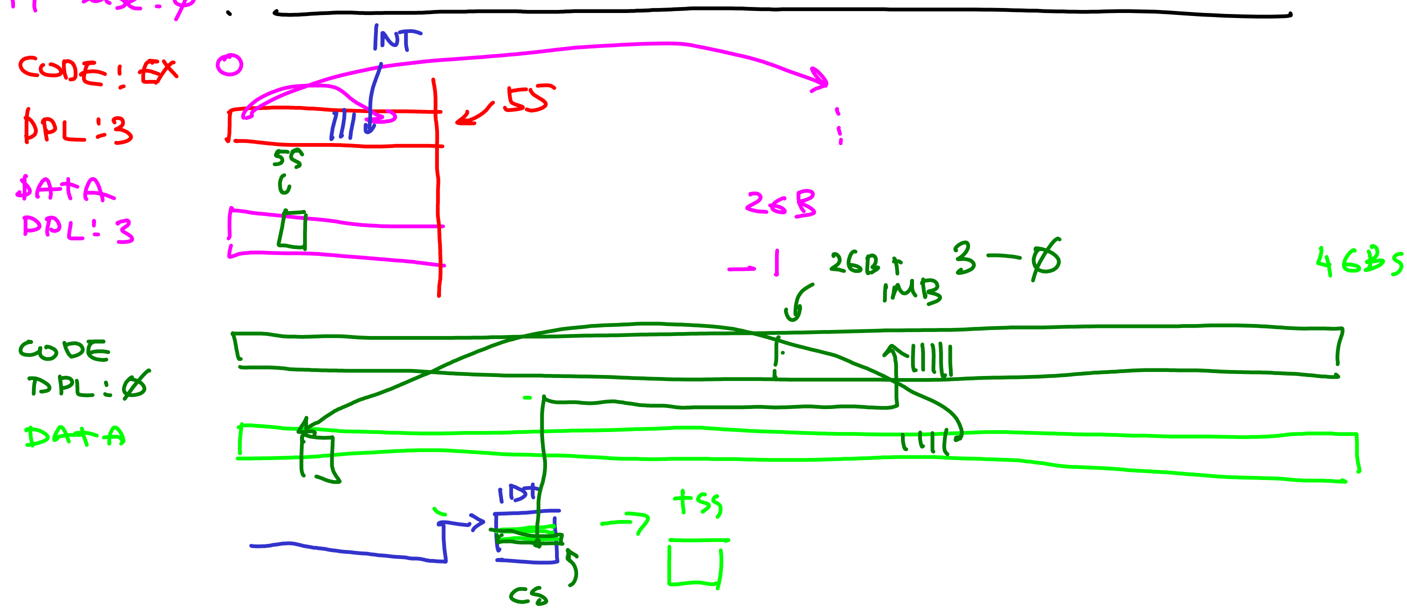


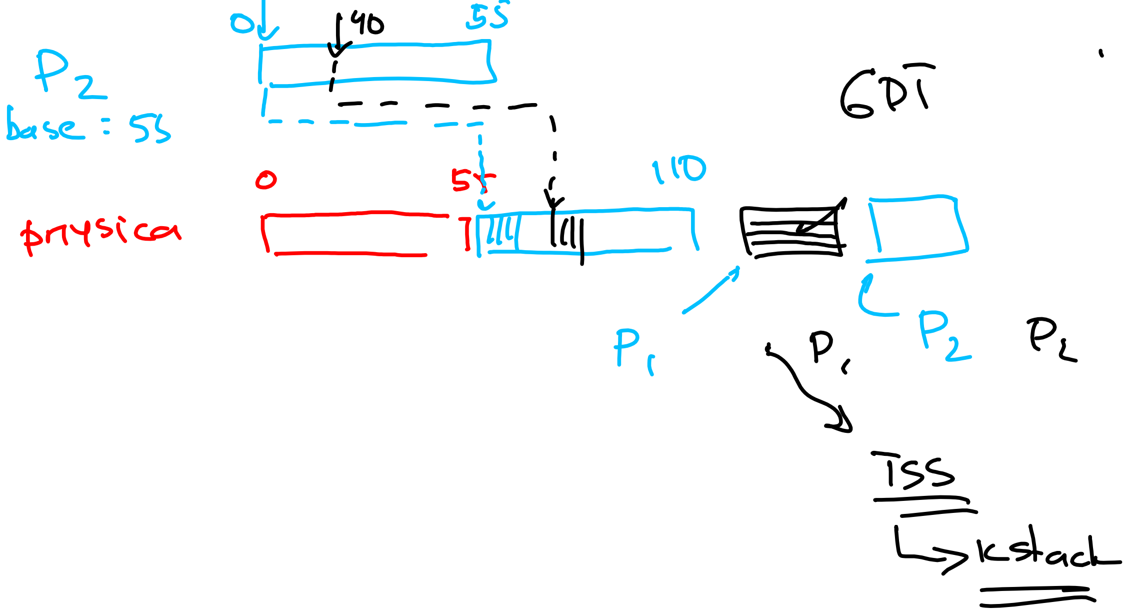
(b) (5 points) Imagine you plan to run xv6 on the hardware that is identical to x86, but does not provide support for paging. What changes you have to make to the xv6 kernel to make sure that the isolation and protection across the processes and between the process and the kernel is in place.

Solution:

Use segmentation. At any given time, there would be two pairs of segments (user and kernel) in the GDT. Every time you switch between processes, you have to reload those entries or switch to another GDT. The kernel entries would stay the same and won't need reloading. The user segments, however, would change.

- *Also elaborate on how context switch would happen, how fork and exec would work here.*





$40 + base_{p2} =$
$= 40 + 55$

6. System calls

   (a) (5 points) What is the purpose of the line 6138 in the listing below (`sys_read()` is the xv6 system call that reads data from a file)?

```
6131 int
6132 sys_read(void)
6133 {
6134   struct file *f;
6135   int n;
6136   char *p;
6137
6138   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6139     return 1;
6140   return fileread(f, p, n);
6141 }
```

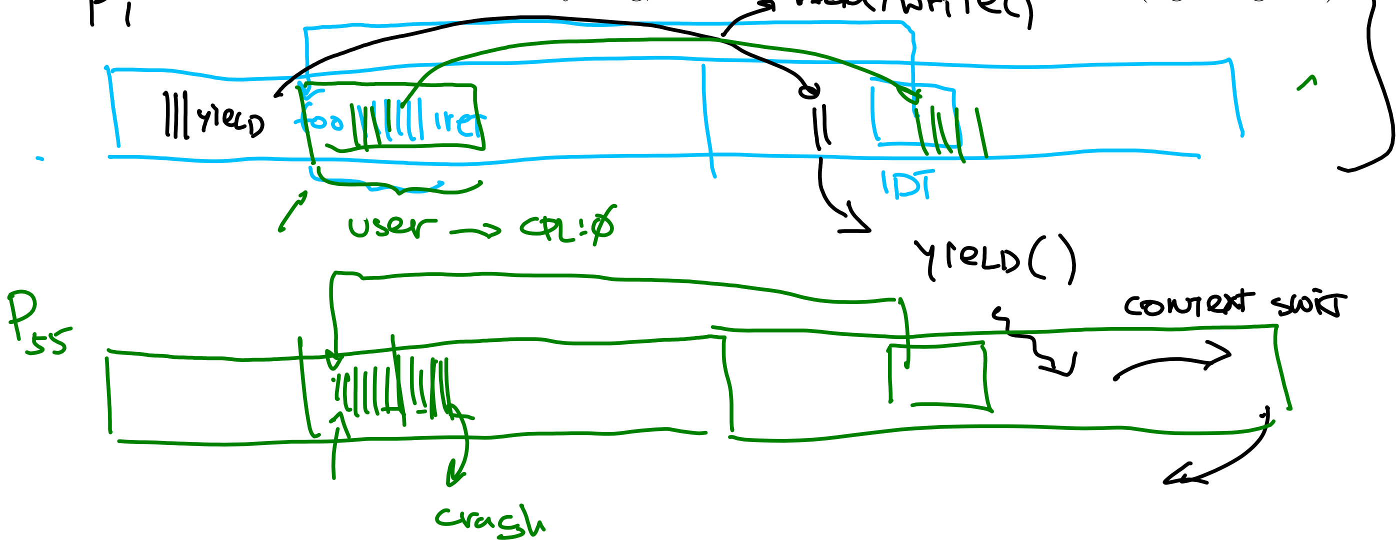
   Solution: The line checks whether the arguments are in the user space, and whether they are well formed.

7. Bob thinks that its ok to let user processes register interrupt handlers. He starts with a timer interrupt, i.e., he adds a new system call that takes a pointer to a function that the kernel adds to the IDT as the handler for the timer interrupt (vector32). The rest of the kernel stays unchanged (same fields in the IDT, same CS selector, same kernel stack in the TSS).

   (a) (7 points) Bob implements his change and it even works! He sees that his timer interrupt handler is executed several times, but then the system crashes in a mysterious way. Explain why the system works initially, but crashes later?

   Solution: The IDT is inside the kernel address space. The IDT is pointed by the hardware register (IDTR). The IDT points to the user level timer interrupt, which is in the user space. Hence we would be running user code, with a privilege level of 0 - this is dangerous, as the user code could do anything, and thus trigger the mysterious crash (e.g. a segfault).

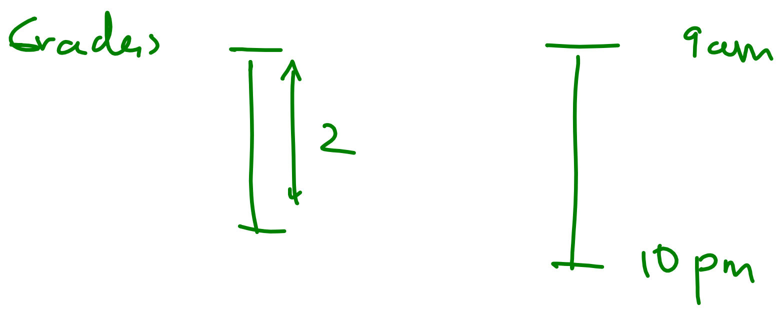
   FINAL



   (b) (7 points) Bob's friend Alice who is a mature OS hacker tells him that his change is ultimately insecure and breaks isolation guarantees of the xv6 kernel? Can you explain what does Alice mean?

   Solution: Following from the above, by being able to execute user code from the kernel, we would be breaking isolation straight away.

1,5 hours → 2 hour



8. 238P organization and teaching

   (a) (3 points) If there is one single most important thing you would like to improve in the CS238P class, what would it be?