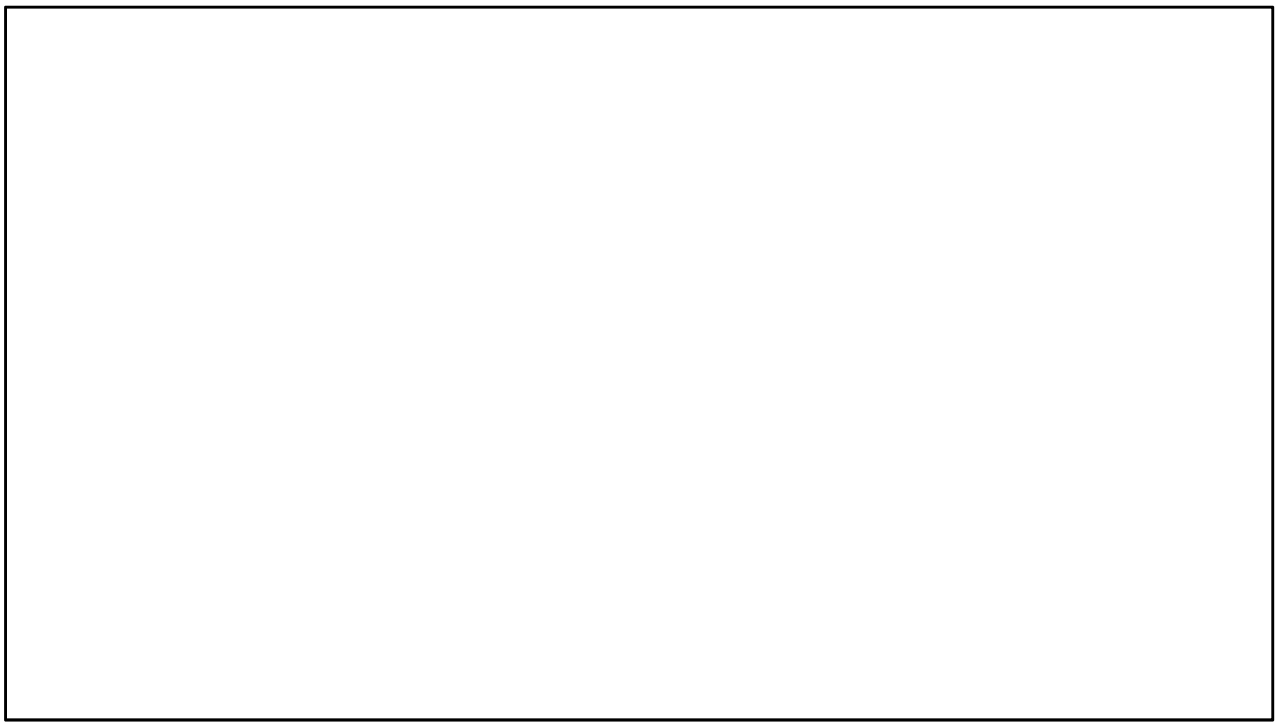


Midterm

- You can bring any notes, books, etc. to the examination hall
- Electronic devices are not allowed
- There could be questions from the must submit assignments

Grading

- To get A+ grade, you have to do the following
  - Conduct help sessions for students who need help
  - Help sessions need to be open for all
  - Please don’t share your code with others, but help them understand the assignments or topics that have been discussed in the class
  - Help others to learn C
- In the end, we will take recommendation from all students
  - Each student can recommend at most one student who was most helpful
  - Those who will conduct help sessions impartially (which are also effective) and score A or A- will be awarded A+

Homework-4

- Q1: cond_wait and cond_signal

```
struct condition
{
   struct list *wait_list;
};
```

```
void cond_wait(struct condition *cond, struct lock *lock)
{
    push_list(cond->wait_list, cur_thread);
    release(lock);
    schedule();
    acquire(lock);
}
```

```
void cond_signal(struct condition *cond, struct lock *lock)
{
     struct thread *t = pop_list(cond->wait_list);
     if (t != NULL)
         push_list(ready_list, t);
}
```

Homework-4

- Q2: release implementation?

```
void release(struct lock *l) {
    unsigned status = interrupt_disable();
    struct thread *t = list_pop(l->wait_list);
    if (t)
        list_push(ready_list, t);
    set_interrupt_status(status);
    l->value = 1;
}
```

Homework-4

- Q2: release implementation?

This implementation is not correct. If the scheduler is invoked just after restoring the interrupt status, a thread that was waiting for a lock and has been moved to the ready list by the lock holder (in release) may get scheduled. Because the lock value is still zero, the thread will not get the lock and will be moved to the waiting list again. If no other thread tries to acquire this lock, the thread which is waiting for the lock will never be moved to the ready list.

```
void release(struct lock *l) {
    unsigned status = interrupt_disable();
    struct thread *t = list_pop(l->wait_list);
    if (t)
        list_push(ready_list, t);
    set_interrupt_status(status);
    l->value = 1;
}
```

Homework-4

- Fork
  - straightforward

Homework-4

- Pipes

```
char* const args[] = {"/bin/ls", NULL};
close(1);
dup(fd[1]);
close(fd[0]);
close(fd[1]);
execv (args[0], args);
```

```
char* const args[] = {"/usr/bin/wc", "-l", NULL};
close(0);
dup(fd[0]);
close(fd[0]);
close(fd[1]);
execv (args[0], args);
```

Homework-5

- Q1: interrupts vs. polling
- Interrupts are good when
  - device events are rare
  - low latency is required
- Polling is good when
  - device events are very frequent
    - CPU spends most of its time in interrupt handlers, and the user-applications are not getting scheduled causing the overall throughput to zero

Homework-5

- Q2. Shared linked-list?

Homework-5

- Q2. Shared linked-list?
- OS allocates a memory region that is shared among all the processes
- OS adds an entry corresponding to the shared region in every process' GDT
- The process' sets the GDT index in %fs segment register with the index of the shared segment
  - %fs is reserved for the shared segment
  - e.g., if the GDT index is 1, the user can set %fs to ((1 << 3) | 3)

```
readval:
mov 4(%esp), %eax
mov %fs:(%eax), %eax
ret
```

Homework-5

- Q3: DMA vs. MMIO

Homework-5

- Q3: DMA vs. MMIO
- If the CPU configures the device to use DMA for sending packets, then the CPU can run other threads while the device is copying the packet from RAM to its memory
- If the CPU uses MMIO for copying packet to the device memory, then the CPU has to write packets to the device memory manually, and it can't run other threads at the same time
- DMA is better because the overall system throughput will be better then MMIO

Segmentation overview

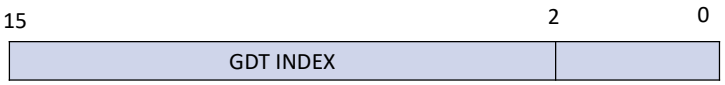
Global descriptor table (GDT)

- The OS creates a GDT in memory
- The GDTR register contains the base address of the GDT
- lgdt instruction is used to load the GDTR
  - lgdt instruction takes 6-byte memory operands
  - 4-byte base address (physical), and 2 bytes size
  - lgdt is a privilege instruction
- The GDT can have at most 2^{13} entries

Segment register

- A segment register contains a 16-bit value
  - The top 13 bits of segment registers contain the index in the GDT
  - The lower 2 bits of the cs segment register contains the current privilege level (CPL)
    - In user-mode, the last two bits of cs register is always 3
    - In kernel-mode, the last two bits of cs register is always 0
  - The hardware identifies the CPL using the lower 2 bits of the cs segment register
  - User applications can't directly modify the CPL
  - CPL is changed during entry to the kernel

Segment selector



Higher 13-bits contains the index in the GDT table.

Default segment register

- cs is the default segment register for instruction pointer (EIP)
- ss is default segment register of for stack pointer (ESP) and frame pointer (EBP)
- ds is default segment register for memory operands (except those which contain esp and ebp)
- es is default segment register in string instructions
- fs and gs are optional can be explicitly used in memory operands

schedule

```
struct thread *ready_list;
struct thread *cur_thread;

void schedule() {
  struct thread *prev = cur_thread;
  struct thread *next =  pop_front(ready_list);
  cur_thread = next;
  update_mmu_mappings(next);
  context_switch(prev, next);
}
```

The OS loads the GDT corresponding to the target process, or update the single GDT with the entries of the target process during scheduling.

update_mmu_mapping

- How to implement update_mmu_mapping, when processes want to use multiple entries in the GDT
  - updating all entries in the GDT may be expensive
- Create per-process GDT
  - load the GDT corresponding to next process in update_mmu_mapping
    - using lgdt

Example

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

The user application wants to write 4-byte zero to the physical address 8192. What would be the assembly code?

8192 is the base of GDT entry at index 2. The application code can first set the index field in a segment register with 2, and then access a VA corresponding to PA 8192. The last three bits in all the segment register is set to three in user-mode, and zero in kernel-mode. The segment register value corresponding to GDT entry 2 is (2 << 3) | 3 in the user-mode. After setting the value in a segment register with (2 << 3) | 3, the application can write 4-byte zero to virtual address 0.

Example

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

```
mov $((1<< 3) | 3), %ax
mov %ax, %fs
movl $0, %fs:120
```

The user application wants to write 4-byte zero to the physical address 3120. What would be the assembly code?

Example

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

```
mov $3, %ax
mov %ax, %fs
movl $0, %fs:128
```

The user application wants to write 4-byte zero to the physical address 1128. What would be the assembly code?

Example

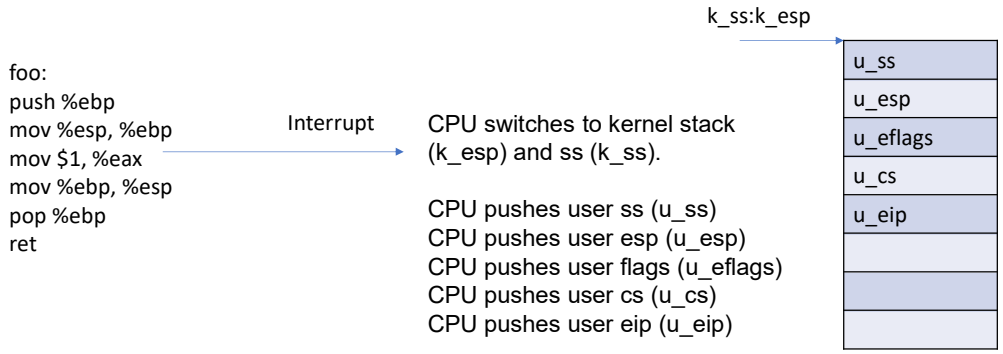
Not possible

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

The user application wants to write 4-byte zero to the physical address 3128. What would be the assembly code?

It is not possible to access PA 3128, because it is outside the limit of index 1.

Interrupts in user-mode



Interrupts in kernel-mode

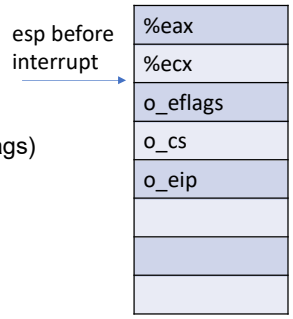
```
interrupt_handler:
push %eax
push %ecx
push %edx
call schedule1
pop %edx
pop %ecx
pop %eax
ret
```

Interrupt →

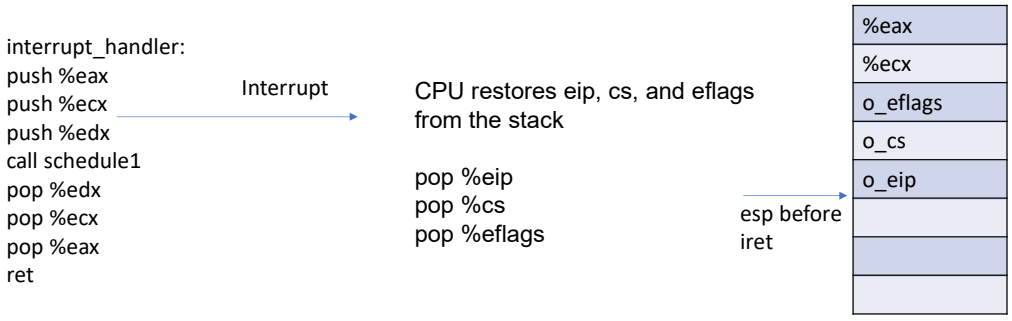
CPU doesn't switch stack.

CPU pushes old flags (o_eflags)
CPU pushes old cs (o_cs)
CPU pushes old eip (o_eip)

esp before interrupt →



iret to kernel-mode



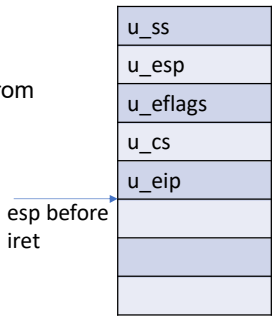
iret to user-mode

```
foo:
push %ebp
mov %esp, %ebp
mov $1, %eax
mov %ebp, %esp
pop %ebp
ret
```

Interrupt

CPU restores five values from the stack:

```
pop %eip
pop %cs
pop %eflags
pop %esp
pop %ss
```



Segmentation

- What happens if a user-application accesses a virtual address that is outside the limit of the segment?

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

movl $100, %ds:(%eax)

what is the PA, where
%eax = 1000
%ds = (1 << 3) | 3;

Segmentation

- What happens if a user-application accesses a virtual address that is outside the limit of the segment?
  - What is the problem with crashing the system?

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

movl $100, %ds:(%eax)

what is the PA, where
%eax = 1000
%ds = (1 << 3) | 3;

If we bring down the whole system when an application accesses an invalid address, then any process can bring the entire system down.

Segmentation

- What happens if a user-application accesses a virtual address that is outside the limit of the segment
  - The hardware generates an exception called general protection fault

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

movl $100, %ds:(%eax)

what is the PA, where
%eax = 1000
%ds = (1 << 3) | 3;

Exceptions

- First 32 values in IDT are reserved for exceptions
  - Read Section 6.15 from Intel Manual - 3
- There are multiple kinds of exceptions
  - e.g., general protection fault, divide by zero, illegal instruction, etc.
  - The exceptions are predefined by the x86 hardware
  - OS can not add a new type of exception
- Each exception has a unique vector number in the IDT
  - e.g., general protection fault vector (13), divide by zero (0), etc.

Exceptions are similar to interrupts, except they are deterministic and can only occur if the program reaches a particular state. The number of exceptions is fixed, and exceptions are predefined by the x86 hardware. The first 32 entries in the IDT are reserved for exceptions.

Exceptions

- On some exceptions, the CPU additionally pushes the error code in addition to three or five values (as discussed before)
  - This is just extra information
  - You can ignore it for now

Interrupts and exceptions

- Interrupts are non-deterministic
  - can come at any program point
- Exceptions are deterministic
  - they are triggered when a particular state is reached

Precise interrupt

- If an interrupt occurs during the partial execution of an instruction, the CPU delays the delivery of the interrupt until the execution of the instruction is completed

Precise interrupt

```
call *0x100   // 0x100 is within limit
```

SEMANTICS:
Save return address on the stack.

Jump to a function whose address is stored in 0x100.

Precise interrupt

```
call *0x100   // 0x100 is within limit

push %eip  // successful
```

Precise interrupt

```
call *0x100   // 0x100 is within limit

push %eip  // successful
interrupt
```

Precise interrupt

```
call *0x100   // 0x100 is within limit

push %eip  // successful
interrupt
interrupt queued
```

Precise interrupt

```
call *0x100   // 0x100 is within limit

push %eip  // successful
interrupt
interrupt queued
mov 0x100, %eip     //successful
```

Precise interrupt

```
call *0x100   // 0x100 is within limit

push %eip  // successful
interrupt
interrupt queued
mov 0x100, %eip     //successful
injecting interrupt
```

Precise interrupt

```
call *0x100   // 0x100 is within limit

push %eip  // successful
interrupt
interrupt queued
mov 0x100, %eip     //successful
injecting interrupt
        switch to k_esp, k_ss
        push u_ss
        …
```

Precise exception

- Can we delay the delivery of an exception until the execution of the current instruction is completed?

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

movl $100, %ds:(%eax)

what is the PA, where
%eax = 1000
%ds = (1 << 3) | 3;

Precise exception

- Can we delay the delivery of an exception until the execution of the current instruction is completed?
  - No, because the reason we have exception is that the hardware doesn't know how it is supposed to execute the excepting instruction

Precise exception

- During exception, CPU rollbacks all the changes made by the excepting instruction before jumping to the exception handler

Precise exception

```
call *0xffff0000   // 0xffff0000 is outside limit
```

Precise exception

```
call *0xffff0000   // 0xffff0000 is outside limit

push %eip   // successful
```

Precise exception

```
call *0xffff0000   // 0xffff0000 is outside limit

push %eip   // successful
mov (0xffff0000), %eip  // exception
```

Precise exception

```
call *0xffff0000   // 0xffff0000 is outside limit

push %eip   // successful
mov (0xffff0000), %eip  // exception
pop %eip // rollbacking change
```

Precise exception

```
call *0xffff0000   // 0xffff0000 is outside limit

push %eip   // successful
mov (0xffff0000), %eip  // exception
pop %eip // rollbacking change
restore the old value on the stack before push %eip
injecting exception
        switch to k_esp, k_ss
        push u_ss
        …
```

Interrupt handler

In addition to caller-saved registers, the OS also saves and restores user's segment registers during kernel entry and exit.
After saving the user's %ds, %es, %fs, and %gs, the interrupt handler sets the segment registers to the values corresponding to kernel segment registers.
Because the kernel needs only one entry in the GDT to access the entire physical address space, all the segment registers can be set to the same value during the kernel execution.

```
push %eax
push %ecx
push %edx
push %ds
push %es
push %fs
push %gs
mov $kernel_seg, %ax
mov %ax, %ds
mov %ax, %es
mov %ax, %fs
mov %ax, %gs
call schedule1
pop %gs
pop %fs
pop %es
pop %ds
pop %edx
pop %ecx
pop %eax
iret
```

No need to save and restore ss and cs registers. CPU automatically saves and restores them during entry and exit.

The kernel uses only one entry in the GDT whose base and limit is set to 0 and 0xFFFFFFFF. Every GDT has an entry corresponding to the kernel. In this code, kernel_seg is the segment register value corresponding to kernel entry. For example, if the kernel is always mapped at index 1 in the GDT, the kernel_seg will be (1 << 3). The last three bits in the segment registers are always zero in kernel-mode. The hardware automatically saves and modifies the %cs and %ss registers on kernel entry. The interrupt_handler manually sets the values of the rest of the segment registers with kernel_seg before calling the other kernel routines. Before modifying the segment registers, the user's segment registers need to be saved on the stack, such that they can be restored before returning to the user-mode.

system call

void receive(char *buf);

How does a user process pass the system call identifier (corresponding to receive) and buf to the kernel?

system call

```
USER:
do_syscall_u(int id, void *buf);
do_syscall_u:
  mov 4(%esp), %eax
  mov 8(%esp), %ecx
  int $128
  ret
```

User-program invokes do_syscall_u to perform the system call.

do_syscall_u takes a system call identifier and a pointer argument and passes them to the kernel.

do_syscall_u copies the identifier to %eax, the pointer to %ecx, and transfers control to the kernel using int instruction. The kernel returns the result of the system call in %eax register.

The kernel is using vector 128 for system call.

system call

USER:

```
do_syscall_u(int id, char *buf);
do_syscall_u:
  mov 4(%esp), %eax
  mov 8(%esp), %ecx
  int $128
  ret
```

KERNEL:

```
do_syscall_k(int id,  void *buf);
system_call:
```

do_syscall_k is a kernel's routine. It takes the system call identifier and user pointer as input.
system_call is the handler corresponding to vector 128. The system_call routine calls do_syscall_k and passes the user's values as arguments.

system call

```
SAVE_ALL:
push %eax
push %ecx
push %edx
push %ds
push %es
push %fs
push %gs
```

```
SET_KERNEL_SEGS:
mov $kernel_seg, %ax
mov %ax, %ds
mov %ax, %es
mov %ax, %gs
mov %ax, %fs
```

```
RESTORE_ALL_EXCEPT_EAX:
pop %gs
pop %fs
pop %es
pop %ds
pop %edx
pop %ecx
add $4, %esp
iret
```

We have defined some macros here. We will use them in the system_call routine. Notice that the return value of the system call is stored in the %eax register. Here, RESTORE_ALL_EXCEPT_EAX doesn't restore %eax.

system call

```
USER:
do_syscall_u(int id, char *buf);
do_syscall_u:
  mov 4(%esp), %eax
  mov 8(%esp), %ecx
  int $128
  ret
```

do_syscall_k is a kernel's routine. It takes the system call identifier and user pointer as input.
system_call is the handler corresponding to vector 128. The system_call routine calls do_syscall_k and passes the user's values as arguments.

```
KERNEL:
do_syscall_k(int id,  void *buf);
system_call:
SAVE_ALL
SET_KERNEL_SEGS
push %ecx
push %eax
call do_syscall_k
add $8, %esp
RESTORE_ALL_EXCEPT_EAX
```

The system_call routine is the target of vector 128 in IDT. system_call saves and restores the user's registers (except restoring %eax), and calls do_syscall_k with the arguments passed by the user. Because the user has passed the arguments in %eax and %ecx registers, they are pushed on the stack before calling the do_syscall_k.

do_syscall_k

```
do_syscall_k(int id, void *buf) {
   switch (id) {
       case RECEIVE_ID:  return receive((char*)buf);
       …
   }
}
```

The do_syscall_k routine checks which system call handler to call (using id) and calls the corresponding hander with the user-supplied buf.

receive

```
int receive(char *ubuf) {
  char kbuf[128];
  receive_in(kbuf);   // receive packet in kernel buffer
  if (!copy_to_user(kbuf, ubuf, 128))  // copy to user buffer
      terminate_process();
  return 1;
}
```

The receive routine reads the packet in a kernel buffer and calls copy_to_user to copy the packet in the user-supplied buf.

copy_from_user

- bool copy_to_user(char *kbuf, char *ubuf, unsigned size);

```
char *kaddr = user_to_kernel(ubuf);
unsigned i;

if (kaddr == INVALID_ADDR)  // invalid user address
   return 0;                // terminate process

for (i = 0; i < size; i++)
    kaddr[i] = kbuf[i];

return 1;
```

copy_to_user routine copies size bytes from kbuf to ubuf. The user's and kernel's virtual addresses are not same because the bases of their respective segments may be different. The user VA can also pass an invalid address. The kernel has to check the validity of the virtual address before translating them to the kernel virtual address. user_to_kernel routine does both the jobs.

user_to_kernel

char* user_to_kernel(char *uaddr, unsigned size);

```
assert(uaddr <= 4096);
assert(uaddr + size <= 4096);
assert(uaddr <= uaddr + size);
```

idx	base	limit	DPL
0	0	0xffffffff	0
1	8192	4096	3

GDT

user_to_kernel checks if virtual addresses in the range [uaddr, uaddr+size] belong to the user's address space. For example, in this case, the user_to_kernel checks if the uaddr and uaddr + size are less than 4096, and uaddr is less than or equal uaddr + size. For example, if the uaddr is 0xffffffff and size is 1, then uaddr + size will be a 0 (due to overflow), which is a valid user virtual address. If these checks are successful, user_to_kernel adds the base of the user segment to the virtual address to compute the physical address. If the base of the kernel segment is not zero, then we need to subtract it from the physical address to get the kernel virtual address; otherwise, kernel virtual address is the same as the physical address.

user_to_kernel

```
char* user_to_kernel(char *uaddr, unsigned size);

char *limit = get_user_gdt_limit();
if (uaddr > limit || (uaddr + size) > limit || (uaddr + size) < uaddr)
    return INVALID_ADDR;
char *base = get_user_base();
return base + (unsigned)uaddr;        // if kernel base is 0
```

idx	base	limit	DPL
0	0	0xffffffff	0
1	8192	4096	3

GDT

Here is the pseudo-code corresponding to user_to_kernel.

User pointers in kernel

- Kernel dereferences all user pointers in special functions, e.g.,
  - copy_to_user
  - copy_from_user
- Before accessing the user pointers kernel checks if the pointers are valid and translate them to kernel virtual address before accessing them

Memory management

- After the midterm, we will study other MMU implementations