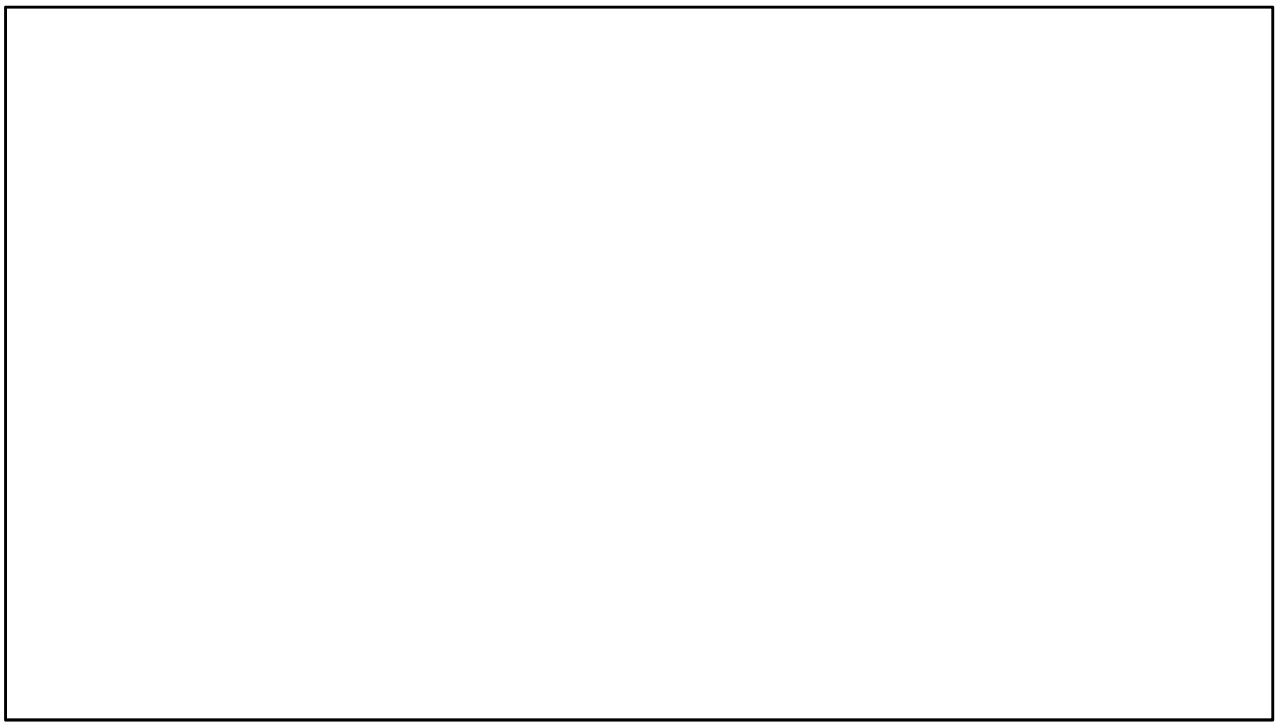


Reference

- Chapter-3 from Intel Manual Vol-3

Example

- Firefox invokes receive(buf) system call to receive a packet in the buf
  - receive(buf) waits until a packet is received in buf

A user application does the receive system call to receive a packet in buf.

receive

```
#define RX_STATUS_PORT   0x402
#define RX_ADDR_PORT      0x403
struct list *wait_list;

void receive (char *buf) {
  /* configure device to copy packet
   * to buf when it is received
   */
  outl (RX_ADDR_PORT, buf);
  outl (RX_STATUS_PORT, 0);
  thread_block(wait_list);    // put the current thread to wait_list and schedule a new thread
}
```

The receive system call handler is a device driver. A device driver is a piece of code that talks to a particular device. In this case, the device driver configures the network device to copy the packet into buf (using DMA), when a packet is received. The current thread is blocked, and a new thread is scheduled.

Interrupt

- rx_intr is invoked after the device has copied an incoming packet to buf

```
void rx_intr () {
  thread_unblock(wait_list); // remove a thread from wait_list
                             // and put it to the ready list
}
```

After the device has copied an incoming packet to buf, it sends an interrupt to the CPU. After receiving the interrupt, the CPU jumps to the interrupt handler of the network device. In the interrupt handler, the thread which is waiting for receiving a packet is moved to the ready list.

Polling

- rx_poll is invoked after every timer interrupt

```
#define RX_STATUS_PORT      0x402
void rx_poll() {
  if (inl(RX_STATUS_PORT) == 1) // if the device has copied a packet to buf
  {
     thread_unblock(wait_list); // remove a thread from wait_list
                                // and put it to the ready list
  }
}
```

Alternatively, the CPU can disable interrupts from the network device and check the status of the device (using port I/O or MMIO). In this example, the device sets the status to 1, after copying the packet. If CPU finds that the device status is 1, it moves the thread that was blocked in the receive system call to the ready list.

Receive livelock

- if the packets are received at a very fast rate
  - CPU spends most of its time in processing the interrupt handler
  - The application (consumer of the packets) won't get a chance to run
  - The throughput of the system will drop to zero
  - this situation is called receive livelock

Eliminating livelock

- Use interrupts to initiate polling
- Disable interrupts (from the network device) after receiving an interrupt
- Re-enabling interrupts after no work is pending

Extra reading (optional)

- Eliminating Receive Livelock in an Interrupt-driven Kernel

Disk driver

- Read disk driver section from chapter 3 of xv6-book for a real device driver
  - Will discuss later in this course

Memory isolation

- Processes have their own reserved quota of RAM
- Decided by OS during fork/exec
- One process can't access memory of the other processes
- OS can access the memory of all processes

11

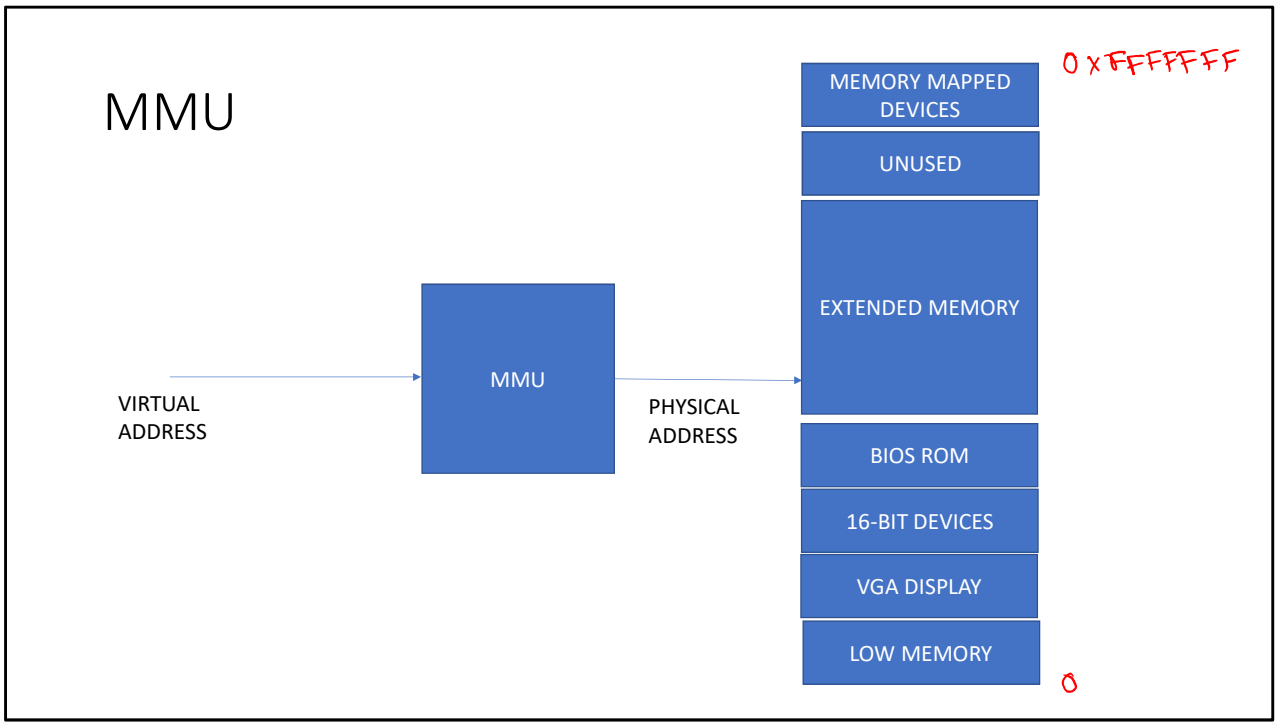
Let's talk about memory isolation. One of the goals of the OS is to ensure that a process can't read or write the data of other processes.

Memory isolation

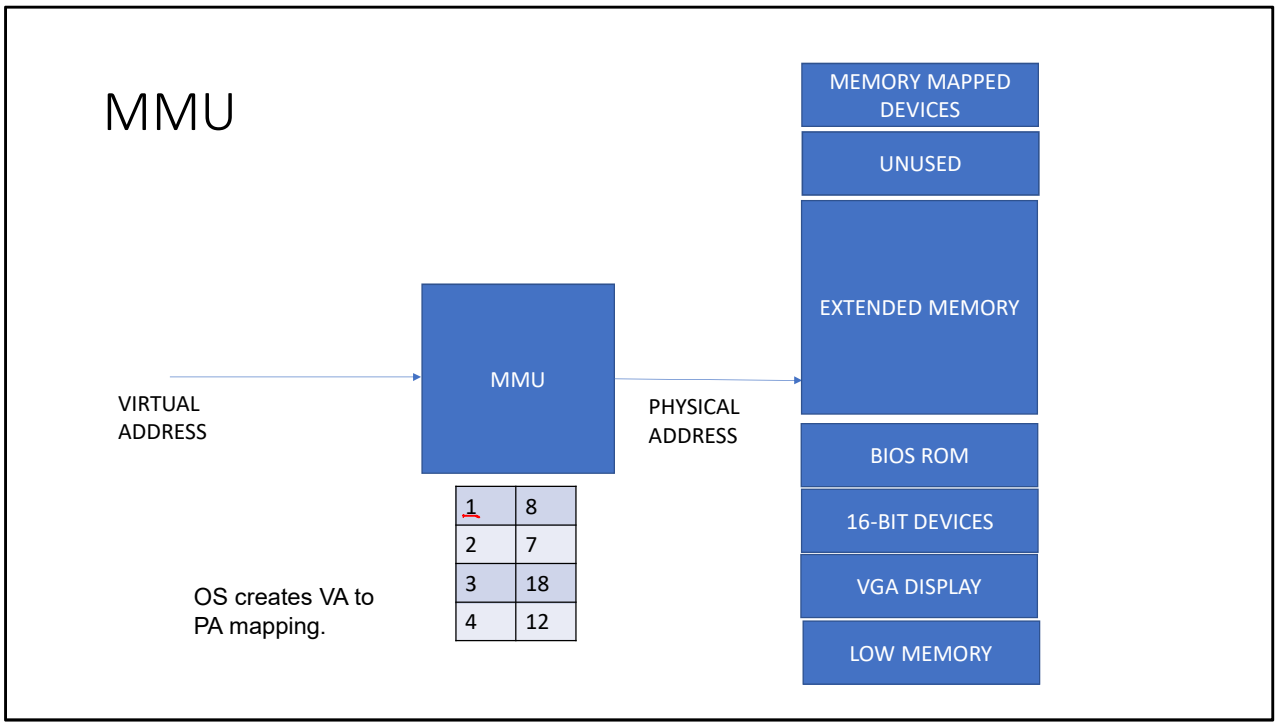
- Memory isolation is done using additional hardware called memory management unit (MMU)
- OS expose a different abstraction to the processes for accessing memory called virtual address space
  - processes can not access the physical address space directly
- MMU translates a virtual address to the corresponding physical address

Memory isolation

- The OS creates the virtual address to physical address mapping
- When the MMU is active memory can only be accessed using virtual addresses

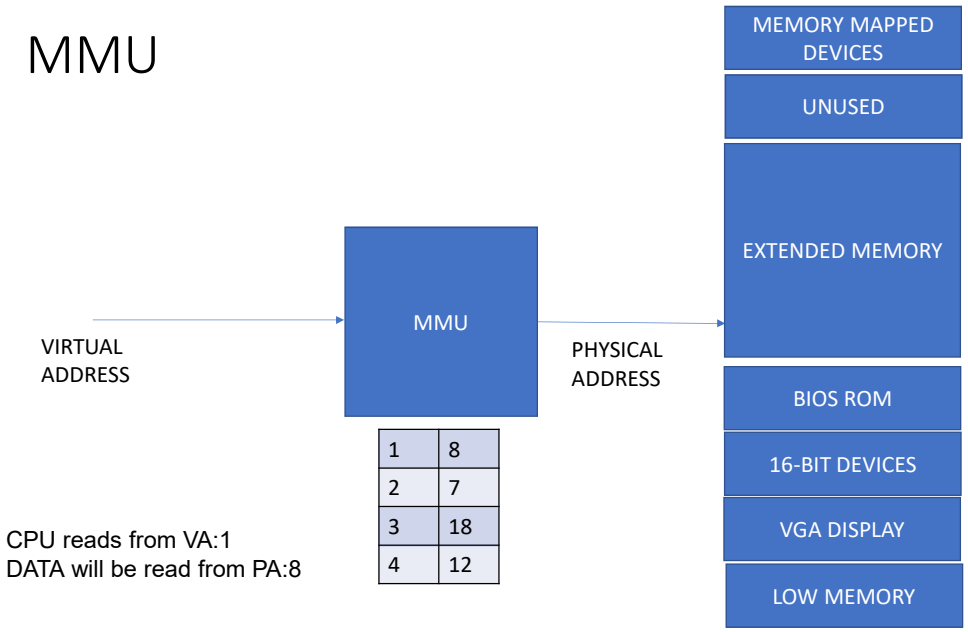


When the MMU is active, all memory accesses are done using virtual addresses. A virtual address is converted to the physical address by the MMU hardware.



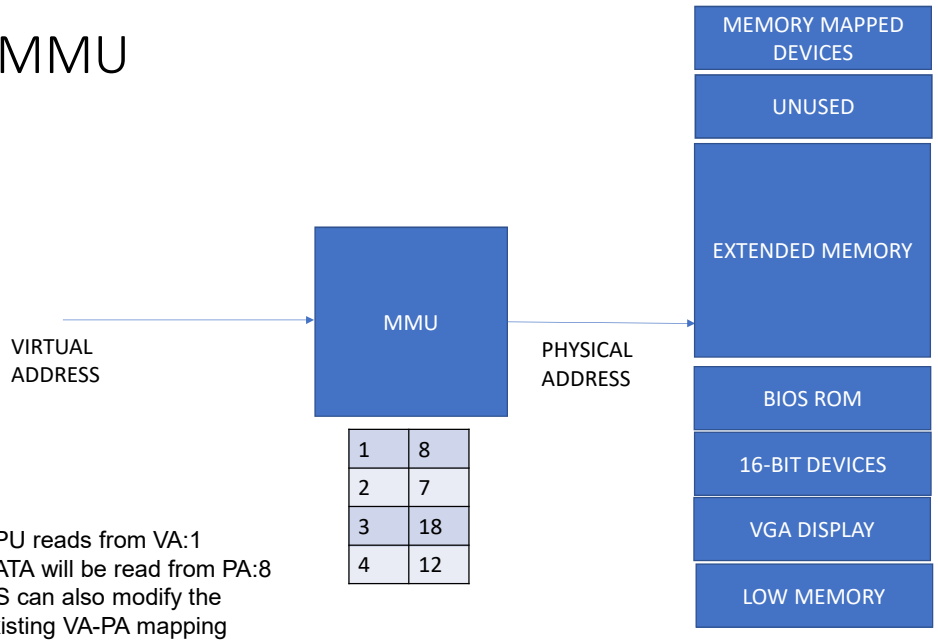
There are several ways to implement the virtual address (VA) to physical address (PA) translation. A simple strategy would be that MMU hardware uses a table called MMU table that contains VA to PA mapping. Whenever memory is accessed, the corresponding address is translated into PA using the MMU table. OS is allowed to add or modify entries to this table.

MMU



CPU reads from VA:1
DATA will be read from PA:8

MMU



CPU reads from VA:1
DATA will be read from PA:8
OS can also modify the existing VA-PA mapping

Memory isolation

- How to implement memory isolation using MMU

Memory isolation

- How to implement memory isolation using MMU
  - Create different VA to PA mappings for different processes

The OS creates different VA-PA mappings for each process.

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);
}
```

During scheduling, the MMU table is overwritten by the VA-PA mappings corresponding to the target process.

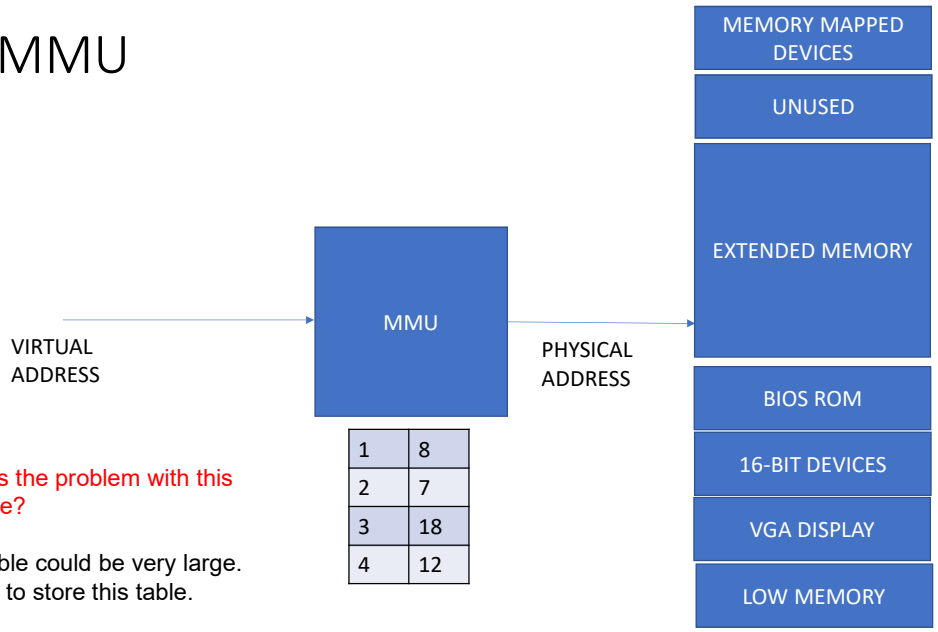
MMU

- Can two processes have the same virtual addresses?

MMU

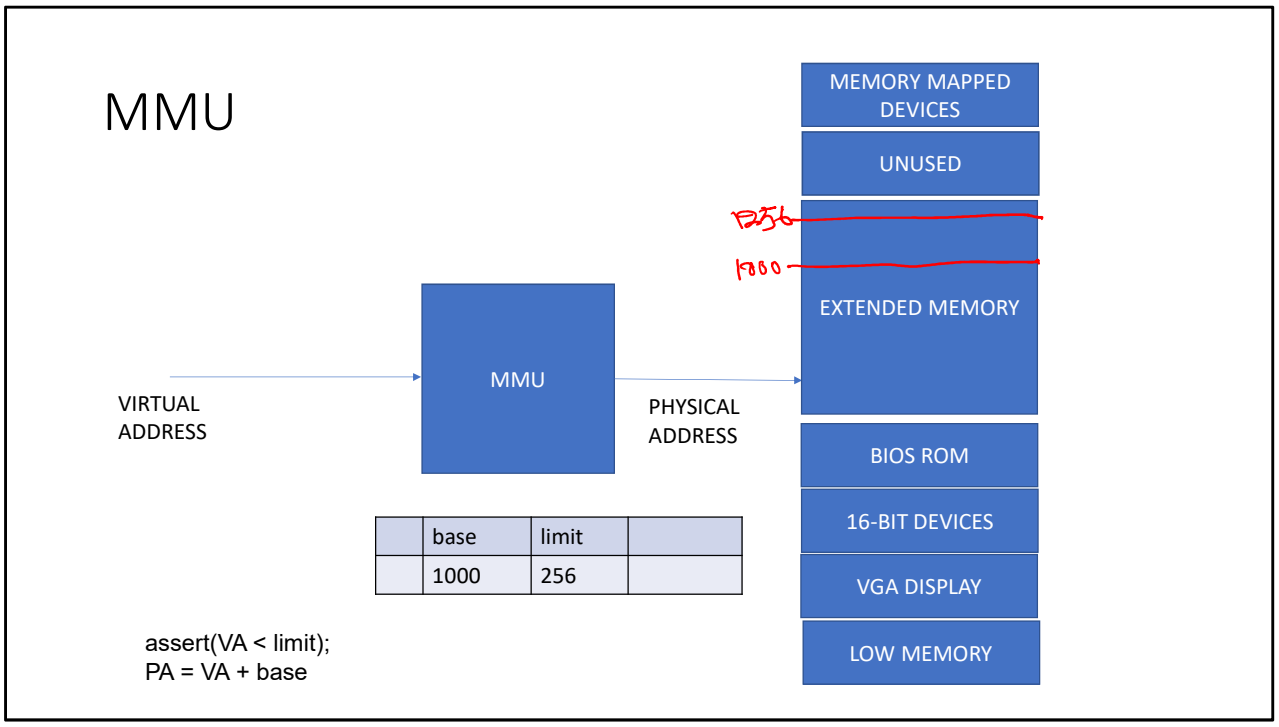
- Can two processes have the same virtual addresses?
  - Yes, as long as the VA-PA mapping is different for both processes, and they are reflected in the MMU table during scheduling

MMU



What is the problem with this scheme?

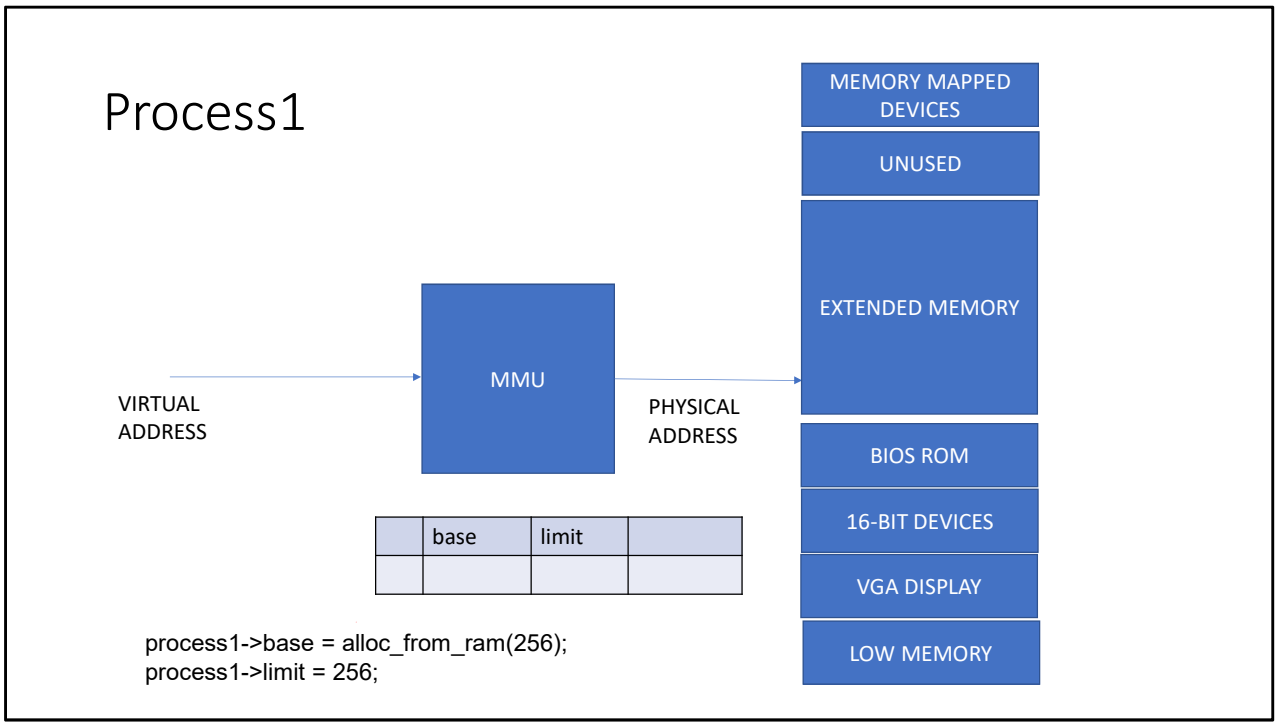
The table could be very large.
Where to store this table.



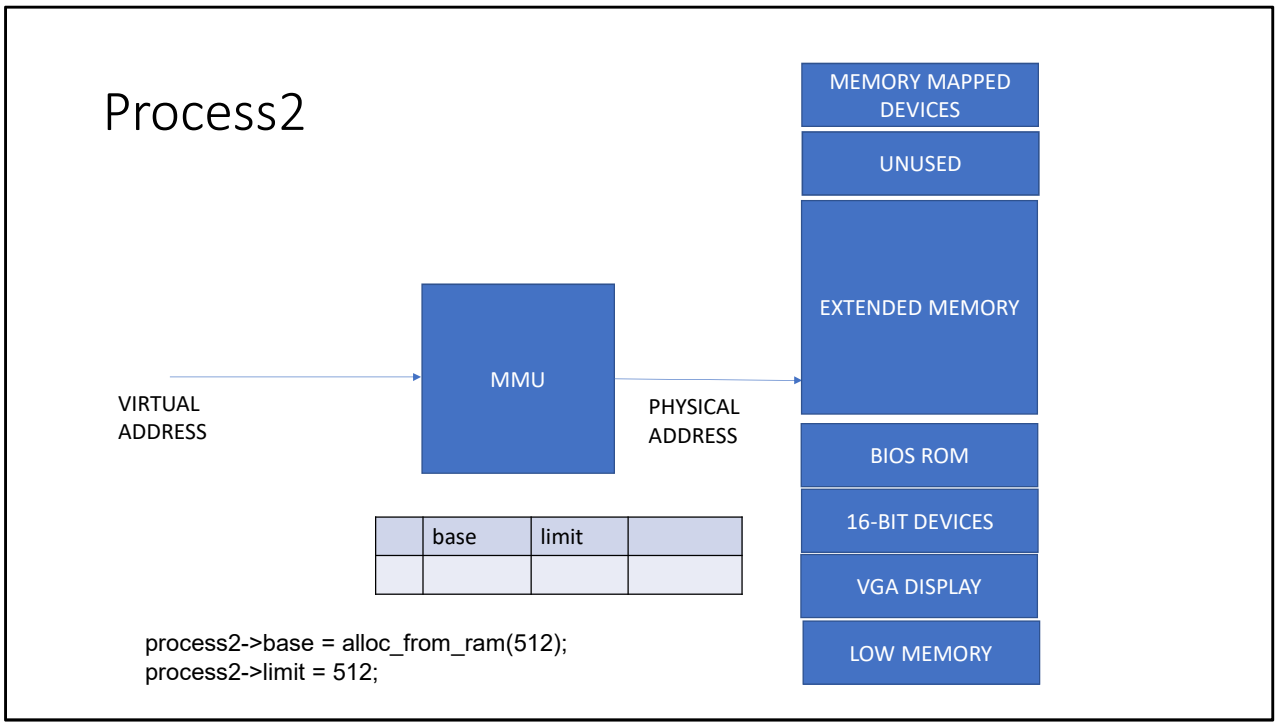
An alternative solution to allocate contiguous memory region for a process in the physical address space. The OS decides the size during the process creation and allocates a contiguous chunk in the RAM for the process. This contiguous space is also called a segment. A virtual address is an offset in the segment. In this scheme, a virtual address can never be greater than the size of the process. MMU table now contains only one entry with two fields: base and limit. The base is the starting address of a process, and the limit is the size of the process. The VA to PA translation is done by merely adding the base to the VA.

MMU

- The OS decides the size of a process
  - OS allocates a memory region of the given size for the process from RAM
  - Each process has a base and limit
    - base is the starting address of the region
    - limit is the size of the process
  - The virtual address space of the process is [0 - limit]
  - PA = VA + base, where VA < limit



MMU table's base and limit fields are set to the base and limit of the target process during scheduling. The OS maintains a metadata corresponding to each process. The process metadata contains the base and limit of the process. In this example, the OS sets the base and limit in process1's metadata with the starting address of process in RAM and 256.



OS creates 512 bytes for process2 and sets the metadata accordingly.

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);
}
```

update_mmu_mappings

```
// next process is process1
mmu->base = process1->base;
mmu->limit = process1->limit;
```

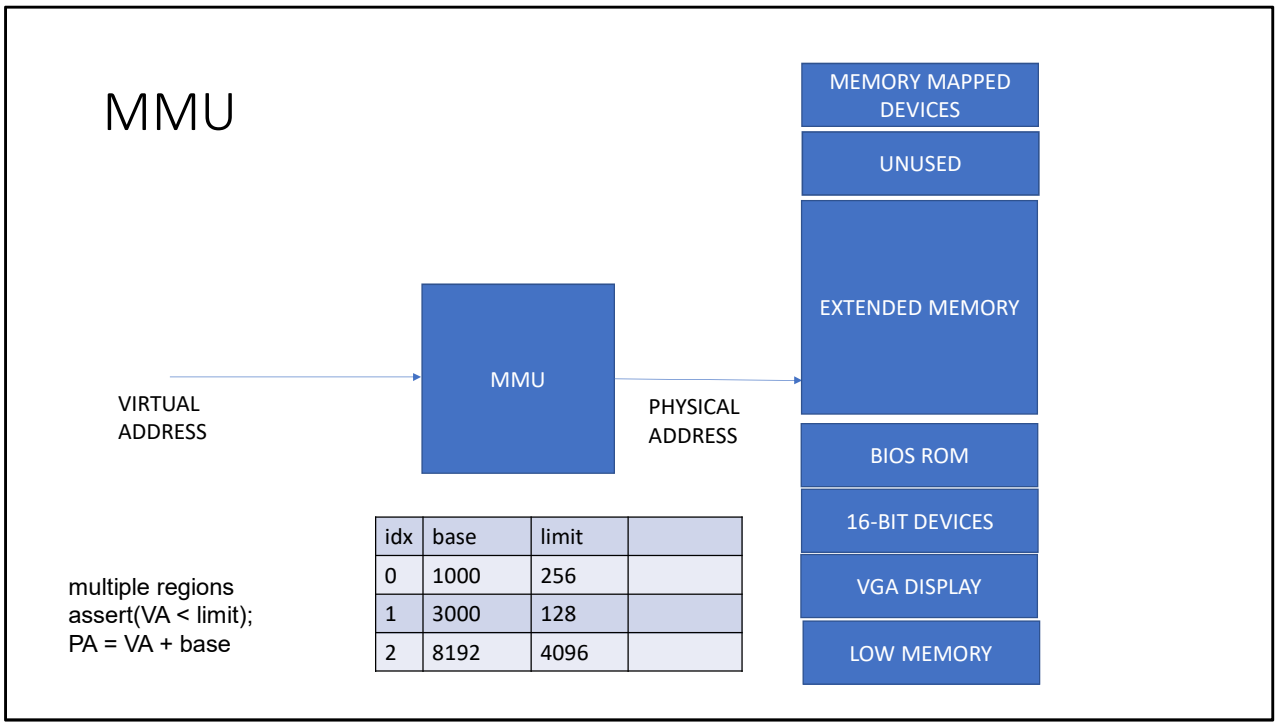
Before scheduling a new thread, OS sets the base and limit in the MMU table from the corresponding process's metadata.

update_mmu_mappings

```
// next process is process2
mmu->base = process2->base;
mmu->limit = process2->limit;
```

MMU

- The disadvantage of using this scheme is that the process can access only one segment (continuous region of memory)



If we want multiple segments, then we need more entries in the MMU table. We also need a mechanism to tell the MMU, which entry to use for the translation.

MMU

- For multiple segments, we need to store the MMU table's index to tell the hardware, which segment to look for the translation
  - To store the index, x86 has six segment registers: %cs, %ds, %es, %ss, %fs, %gs
  - All memory operands in x86 take a segment register
    - default segment register for a memory operand is %ds
  - The compiler generates the virtual addresses (an offset within a segment)
  - MMU converts a virtual address to the physical address by adding the base of the segment to the virtual address

mov $100,%si(,1,eax)

A segment register is used to store the index in the MMU table. Each memory access is associated with a segment register. The translation is done using the index in the segment register.

Example

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

movl $100, %ds:(%eax)

Let us say, %eax contains 1000.
%ds contains index 1.
What is the physical address?

In this example, because the index in %ds segment register is 1, the second entry (with index 1) will be used for the translation. Because the limit of that particular entry is 128, and the virtual address in memory operand (%eax) is 1000 (which is greater than the limit), the hardware will generate some fault.

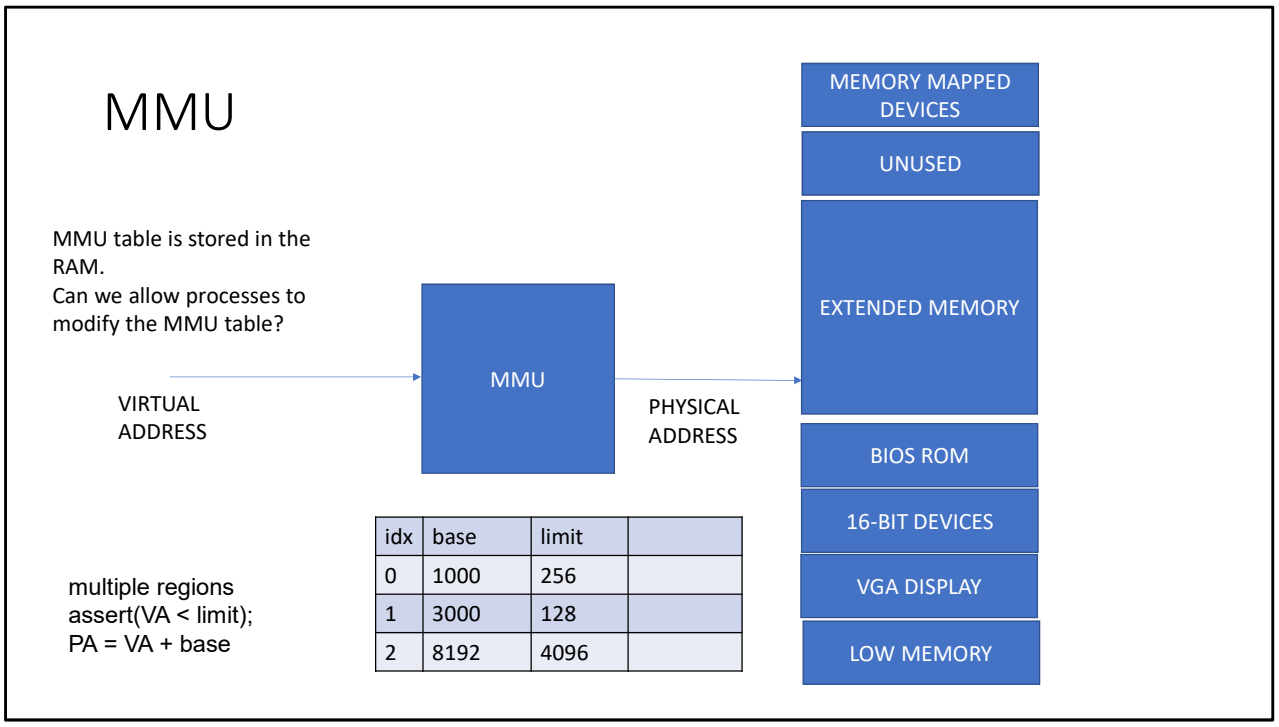
Example

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

movl $100, %ds:(%eax)

Let us say, %eax contains 100.
%ds contains index 1.
What is the physical address?
3100

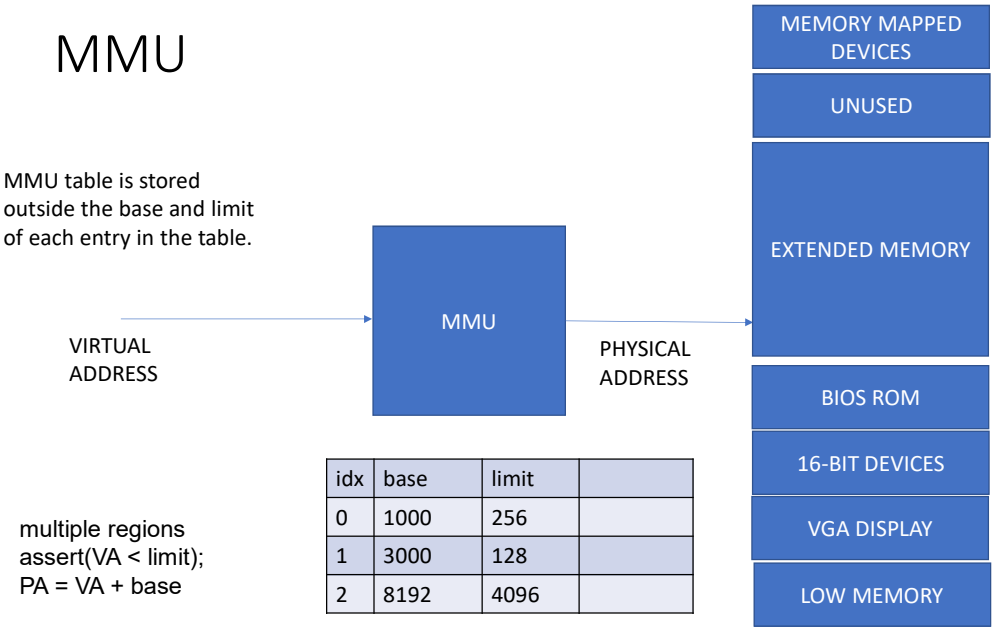
In this example, because the index in %ds segment register is 1, the second entry (with index 1) will be used for the translation. In this case, the virtual address is 100, which is less than the limit (128), the PA is computed by adding the base (3000) to the VA(100). So the physical address is 3100.



No, processes are not allowed to modify the MMU table; otherwise, they can access any physical address by changing the base and limit in the MMU table.

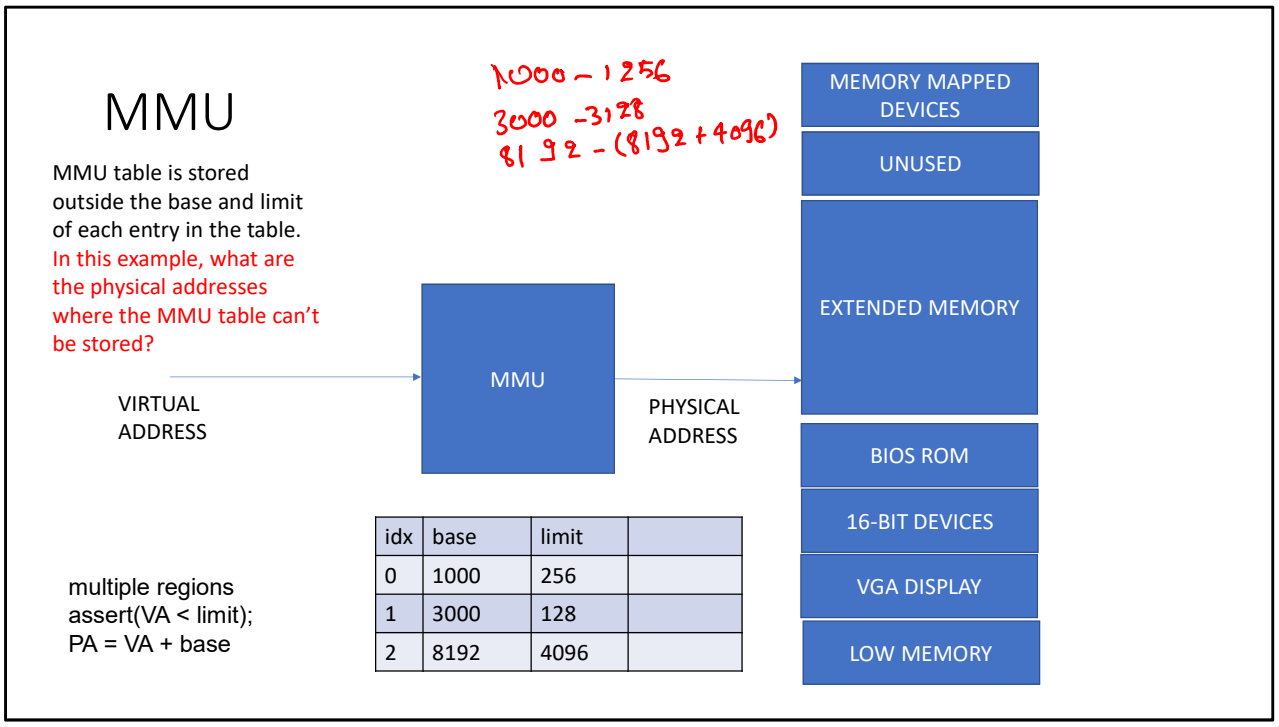
MMU

MMU table is stored outside the base and limit of each entry in the table.



multiple regions
assert(VA < limit);
PA = VA + base

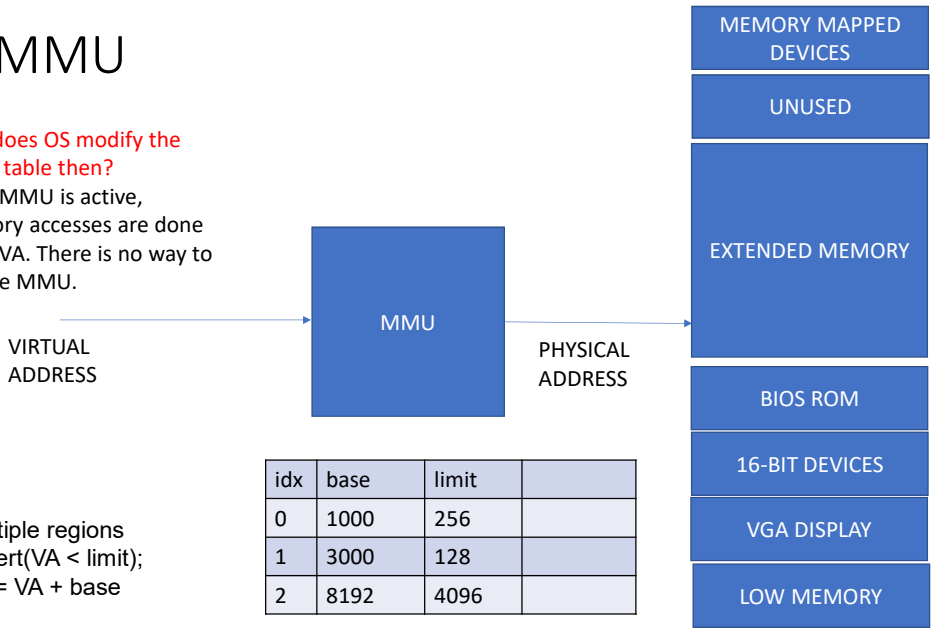
idx	base	limit	
0	1000	256	
1	3000	128	
2	8192	4096	



MMU table can't be stored in physical addresses that belong to the ranges: [1000-1256], [3000-3128], and [8192-8192+4096].

MMU

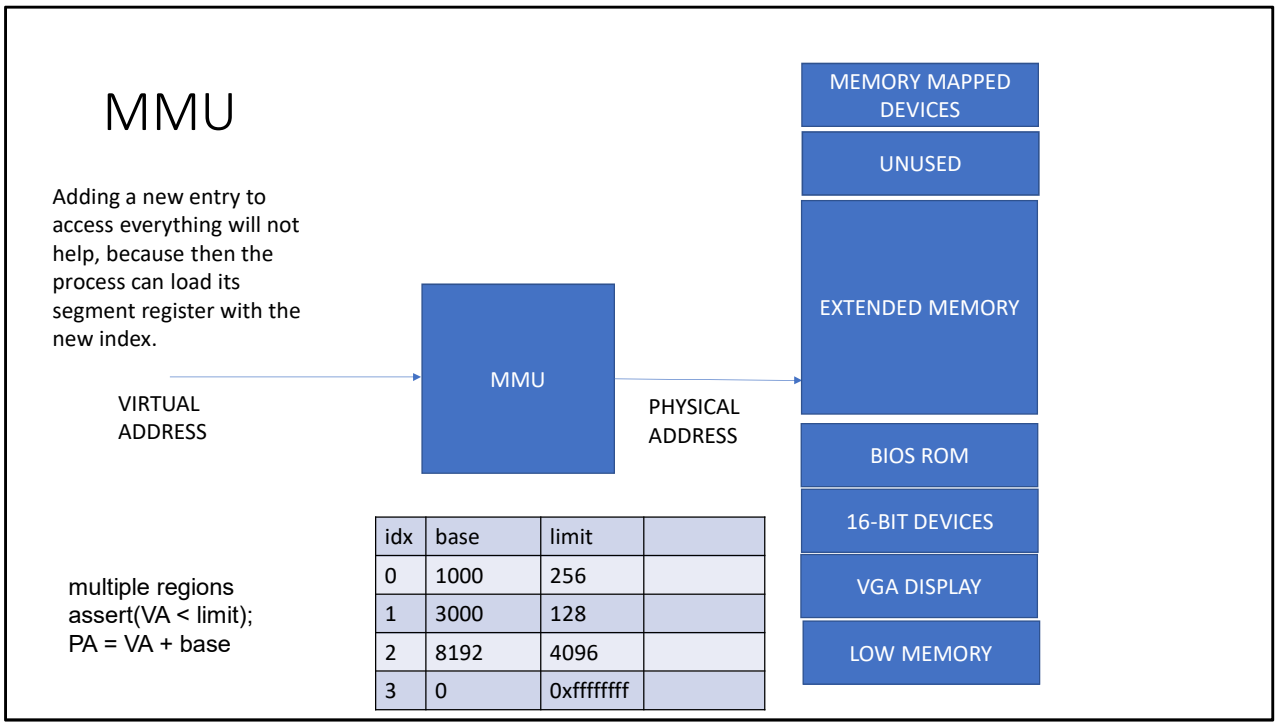
How does OS modify the MMU table then?
If the MMU is active, memory accesses are done using VA. There is no way to disable MMU.



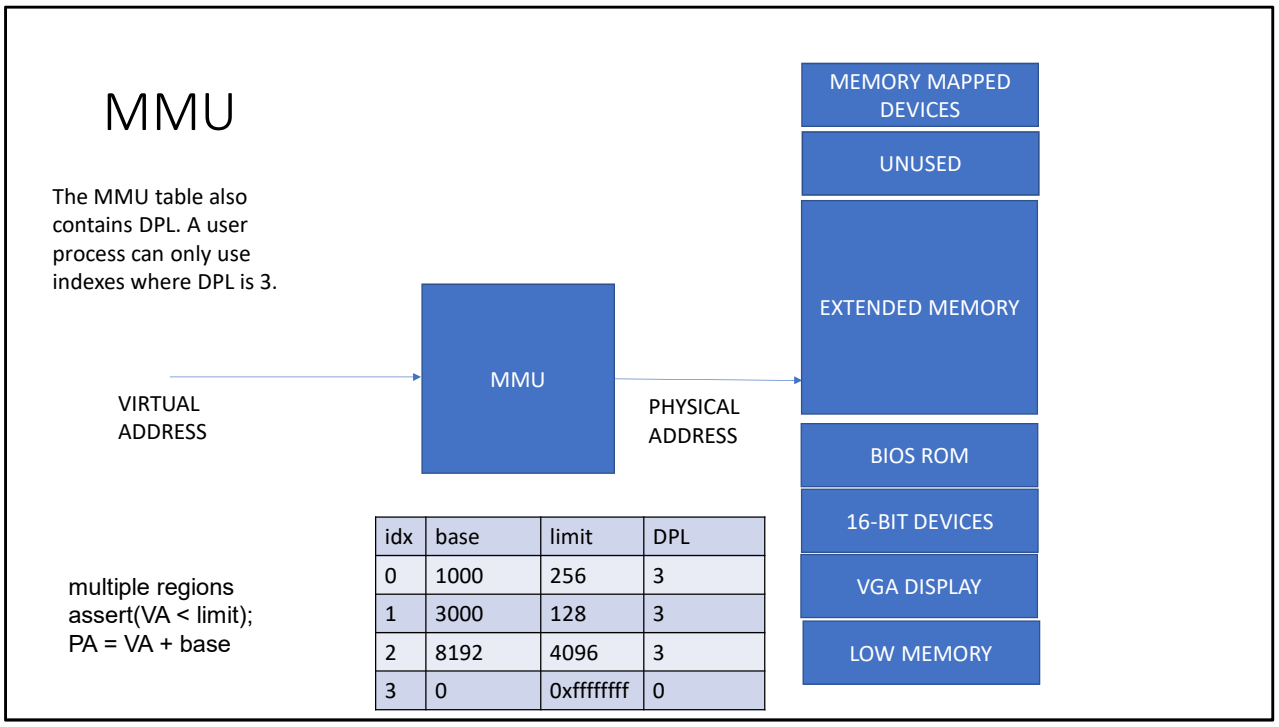
multiple regions
assert(VA < limit);
PA = VA + base

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

OS memory accesses are also virtual because all memory accesses by CPU are done using virtual addresses. There is no way to disable MMU. To allow the modification of the MMU table by OS, we need to do something more.



One way of doing that would be to add a new entry to access any physical address, but that is not a valid solution because the user application can also access everything using that entry.

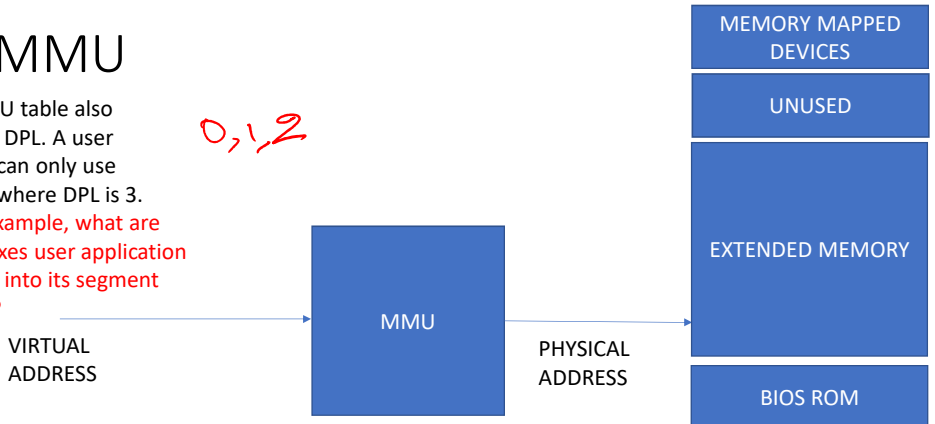


The MMU table contains another field that contains the least privilege level required to access that entry, to prevent a user program from accessing the entry corresponding to OS (which allows access to entire physical address space).

MMU

The MMU table also contains DPL. A user process can only use indexes where DPL is 3.
In this example, what are the indexes user application can load into its segment register?

0, 1, 2



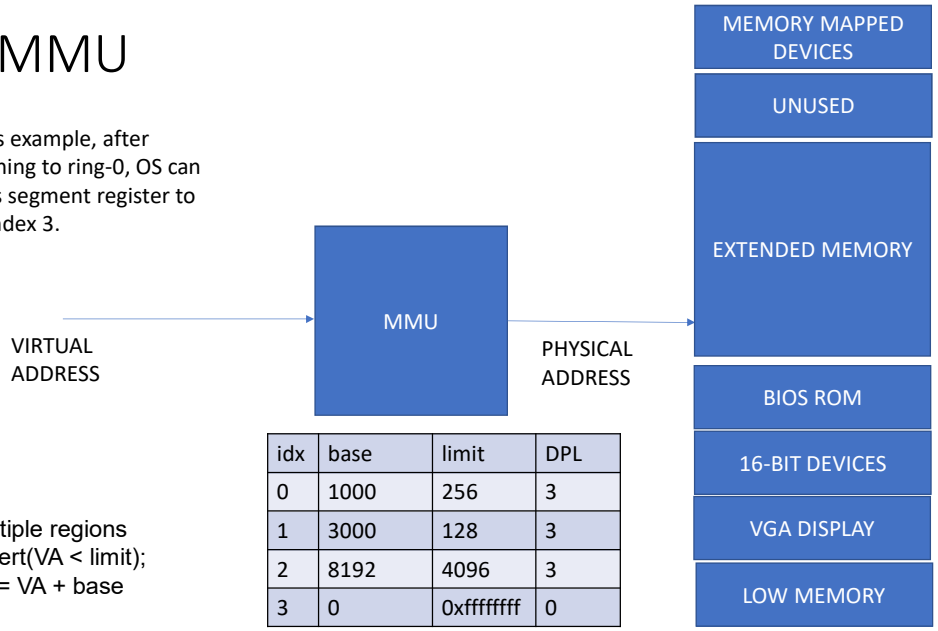
multiple regions
assert(VA < limit);
PA = VA + base

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

User applications can not set its segment registers to use index 3, because only software running in ring-0 can use that index.

MMU

In this example, after switching to ring-0, OS can set its segment register to use index 3.



multiple regions
assert(VA < limit);
PA = VA + base

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

OS entry points

- A user program can enter OS
  - due to a system call
  - due to interrupts and exceptions

MMU

- The MMU table discussed previously is called the global descriptor table (GDT)

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

GDT

- Is it possible that the base address of the GDT is virtual?

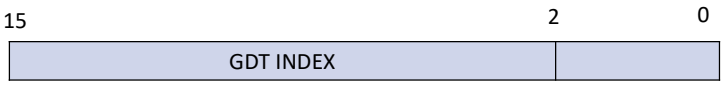
GDT

- Is it possible that the base address of the GDT is virtual?
  - No, GDT is used to translate a virtual address to physical address
  - For translation, GDT need to know the base address of the GDT to fetch the base address at a given index
  - If the base address of GDT is itself virtual then how hardware is supposed to convert it to the physical address
    - it's a chicken and egg problem

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 in a segment register 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

Segmentation in practice

- In practice, it is difficult to generate code using multiple memory regions having the same virtual addresses
- Most compilers use only one region [base, limit]
  - only one entry for process in the GDT
  - all segment registers use the same index in GDT

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);
}
```

update_mmu_mapping

- How to implement update_mmu_mapping
  - Only two entries in the GDT
  - index 0 for the OS
  - index 1 for the process

GDT[1].base = process.base;

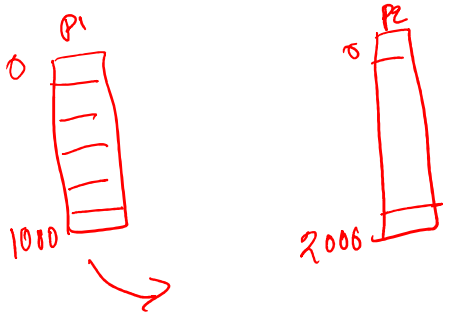
GDT[1].limit = process.size;

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

update_mmu_mapping

execute_lgdt(process->gdt_base, process->gdt_limit);



If the number of GDT entries used by a process is too many, it is better to have a per-process GDT table instead of overwriting a single GDT table during scheduling. In this case, the GDT of the next process is loaded during schedule.

Example

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

mov $100, %ds:(%eax)

what is the PA, where
%eax = 100
%ds = (1 << 3) | 3;

3100

The higher 13-bits in %ds contains 1. The PA is computed using adding base at index 1 (3000) to VA (100).

Example

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

10: mov $100, %ds:(%eax)

Let us say that the VA of mov instruction is 10.
what is the PA of mov, where %cs = (2 << 3) | 3;

8202

%cs is the default segment register for EIP. In this case, the EIP of mov instruction is 10. Because, the higher 13 bits in %cs register contains 2, PA corresponding to EIP can be obtained by adding 8192 to 10.

Compiler

- Does the compiler know the virtual addresses during compile time

Compiler

- Does the compiler know the virtual addresses during compile time
  - Yes

Compiler

- Does the compiler know the virtual addresses during compile time
  - Yes
- Memory operands and instruction pointers are virtual addresses or physical addresses?

Compiler

- Does the compiler know the virtual addresses during compile time
  - Yes
- Memory operands and instruction pointers are virtual addresses or physical addresses?
  - virtual addresses

Compiler

- Does the compiler know the virtual addresses during compile time
  - Yes
- Memory operands and instruction pointers are virtual addresses or physical addresses?
  - virtual addresses
- How they get converted to the physical address

Compiler

- Does the compiler know the virtual addresses during compile time
  - Yes
- Memory operands and instruction pointers are virtual addresses or physical addresses?
  - virtual addresses
- How they get converted to the physical address
  - Each memory operand, EIP, ESP, etc., are associated with a segment register
  - The top 13-bits in segment registers is an index in the GDT table
  - The MMU hardware finds the base from the GDT entry corresponding to the index
  - The base is added to the virtual address to get the physical address

Global variable

```
int a;

foo() {
  a = 0;
}
```

```
4:    // a is allocated here

00000008 <foo>:
  8:  55                      push   %ebp
  9:  89 e5                   mov    %esp,%ebp
  b:  c7 05 04 00 00 00 00    movl   $0x0,0x4
 12:  00 00 00
 15:  5d                      pop    %ebp
 16:  c3                      ret
```

Because the compiler statically knows the virtual address of global variables, it can generate code for directly accessing the global variables. Here the global variable is allocated at virtual address 4. The compiler can generate code to directly write to virtual address 4 that will be automatically get converted to the PA by the segmentation hardware at runtime.

Global variable

```
int a;

foo() {
  a = 0;
}
```

```
4:    // a is allocated here

00000008 <foo>:
  8:  55                    push  %ebp
  9:  89 e5                 mov   %esp,%ebp
  b:  c7 05 04 00 00 00 00  movl  $0x0,0x4
 12:  00 00 00
 15:  5d                    pop   %ebp
 16:  c3                    ret
```

At runtime, the OS sets all the segment registers to:
(2 << 3) | 3
before returning to user mode.
What is the address of a.
Which address movl $0x0, 0x4 will write to?

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

The mov will write to physical address 8196.

call

```
foo() {
  bar();
}
```

```
00000008 <foo>:
  8:  55                    push  %ebp
  9:  89 e5                 mov   %esp,%ebp
  b:  e8 02 00 00 00        call  12 <bar>
  10: 5d                    pop   %ebp
  11: c3                    ret

00000012 <bar>:
  12: c3                    ret
```

At runtime, the OS sets all the segment registers to:
(2 << 3) | 3
before returning to user mode.
What value will be pushed on the stack at call 12

0x10

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

The call instruction pushes the virtual address 0x10 on the stack.

ret

```
foo() {
  bar();
}
```

At runtime, the OS sets all the segment registers to:
(2 << 3) | 3
before returning to user mode.
Which address bar will return to when called from foo

```
00000008 <foo>:
  8:  55                    push   %ebp
  9:  89 e5                 mov    %esp,%ebp
  b:  e8 02 00 00 00        call    12 <bar>
  10: 5d                    pop    %ebp
  11: c3                    ret

00000012 <bar>:
  12: c3                    ret
```

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

The ret instruction jumps to the virtual address on the stack. The actual physical address translation of the return address is done using the %cs segment register. In this case, the physical address of the return address in foo is 8192 + 0x10.