Assignment-3

- Implement shell
  - Wait for user-input (command)
  - Interpret the command as the original shell will do
    - Parse the input string
  - Execute the command
    - Implement I/O redirection
    - Pipe

SPEC benchmarks

- Running SPEC benchmarks using the gcc and clang compilers
- SPECrate Integer
  - perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, xz
- SPECrate Floating Point
  - cactuBSSN, namd, parest, povray, lbm, wrf, blender, cam4, imagick, nab
- Size of input
  - ref

SPEC

- The SPEC benchmarks are useful when you want to compare the performance of two different architectures or two different implementations of the same architecture
  - e.g., ARM vs. X86, AMD vs. Intel
- The SPEC benchmarks are useful to compare the performance of two compilers
  - e.g., gcc vs. clang
- throughput is not the only metric to judge a given CPU
  - power consumption, cost, etc. are the other factors as well
  - Two different implementations of the same ISA might have different performance
    - Intel Atom vs. Intel Core-i7

Virtual machine

- The hardware support for virtualization enables OSes to run an OS inside a virtual machine
- A virtual machine (VM) is the same as a user-process
  - Not allowed to access the system resources directly
  - the OS inside the VM is called guest OS
  - the OS on which the VM is running is called the host OS
- You have to run the Linux as guest OS

Linux kernel compile

- When you install Ubuntu (or other Linux distribution), they come up with pre-built Linux kernel

- However, Linux is open source
  - You can make your custom Linux kernel and use it

- You are to build the Linux kernel from the source inside the guest OS and boot the guest OS with the newly compiled kernel

Midterm

- You can submit for regrading if there are errors in the evaluation
- If you think that your solution is correct, but you were not given marks then you have to write the difference between your solution and the solution discussed in the class on the last page of your answer sheet before submitting them to TAs for regrading
- If your justification is incorrect two marks will be deducted
  - This is necessary to reduce the number of spurious requests

Q1

```
int foo() {
    int a, b, c;
    a = 0, b = 1;
    c = bar(&a, b);
    return c;
}
```

a (on stack)
b (%edx)
c (%eax)

```
foo:
  sub $4, %esp
  mov $0, (%esp)
  mov $1, %edx
  sub $8, %esp
  lea 8(%esp), %eax
  mov %eax, (%esp)
  mov %edx, 4(%esp)
  call bar
  add $12, %esp
  ret
```

Q1

```
int bar(int *a, int *b) {
    return baz(&a, &b);
}
```

lea 4(%esp), %eax
lea 8(%esp), %ecx
push %ecx
push %eax
call baz
add $8, %esp
Ret

Q1

```
int baz(int **a, int *b) {
    int *x = *a;
    int y = *x;
    int z = *b;
    return y + z;
}
```

x (%esi)
y (%edi)
z (%eax)

```
push %esi
push %edi
mov 12(%esp), %eax
mov (%eax), %esi
mov (%esi), %edi
mov 16(%esp), %eax
mov (%eax), %eax
add %edi, %eax
pop %edi
pop %esi
Ret
```

Q2

- Disadvantage of frame pointers
  - If backtrace is not required, the compiler reserves the %ebp for accessing the local variables and parameters. However, the compiler can calculate the addresses of local variables and parameters by its knowledge of current stack depth. The downside of using the frame pointer is %ebp can not available of allocating a local variable or storing a temporary computation. You have to give an example that asserts the above fact.

Q3

context_switch:
~~push %ebx~~
~~push %esi~~
~~push %edi~~
~~push %ebp~~
mov 4 ~~20~~(%esp), %eax
mov 8 ~~24~~(%esp), %ecx
mov %esp, (%eax)
mov (%ecx), %esp
~~pop %ebp~~
~~pop %edi~~
~~pop %esi~~
~~pop %ebx~~
ret

all registers are caller-saved

Q3

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

```
interrupt_handler:
push %eax
push %edx
push %ecx
push %edi
push %esi
push %ebp
push %ebx
call schedule1
pop %ebx
pop %ebp
pop %esi
pop %edi
pop %ecx
pop %edx
pop %eax
iret
```

Q4

```
int foo() {
    int a = 100;
    return a;
}

void bar() {
    write(1, “hello”, 5);
}
```

bar is the current thread.
interrupted in the write system call handler
call_stack at the start of context_switch.
call_stack at the end of context_switch.

schedule
schedule1
write
system_call
bar

schedule
schedule1
interrupt_handler
foo

Q5

What is wrong with directly jumping to the kernel code?

The user can overwrite the kernel code.
If you have assumed that the user and kernel are in the same privilege ring, then other answers are also valid.
E.g., jumping directly to schedule may cause some concurrency issues.
The user-program can access private kernel data by inspecting stack, etc.

Q6

```
release(struct lock *l) {
 1.  int status = interrupt_disable();
 2.  struct thread *t = list_pop(l->wait_list);
 3.  set_interrupt_status(status);
 4.  status = interrupt_disable();
 5.  if (t)
 6.      list_push(ready_list, t);
 7.  l->value = 1;
 8.  set_interrupt_status(status);
}
```

```
acquire(struct lock *l) {
  int status = interrupt_disable();
  while (l->value == 0) {
    list_push(l->wait_list, cur_thread);
    schedule();
  }
  l->value = 0;
  set_interrupt_status(status);
}
```

T1: acquired lock
T1: releasing lock
T1: list_pop returns NULL
T1: enables interrupt after line-3
T2: is scheduled
T2: tries to acquire lock acquired by T1
T2: moved to wait_list because l->value == 0
T1: is scheduled
T1: disabled interrupts
T1: releases lock
T2: still in waiting list

Q7

```
void *alloc(int x) {
  char *val = (char*)mymalloc(x-8);
  int i;
  if (val == NULL)
     return NULL;
  for (i = 0; i < x; i++)
     val[i] = 0;
  return val;
}
```

x = 16, the program behaves correctly because even though the application tries to allocate 8 bytes, 16 bytes will be allocated

x = 24, the program will not behave correctly, because the application tries to allocate 12 bytes, 16 bytes will be allocated, but the application writes 24 bytes.

Q8

```c
int counter = 0;

void foo(void *ptr) {
    1. thread_yield();
    2. int val = counter;
    3. counter = val;
    4. thread_exit();
}

int main() {
    create_thread(foo, NULL);
    create_thread(bar, NULL);
    create_thread(baz, NULL);
    wait_for_all();
    return 0;
}

void bar(void *ptr) {
    1. int val = counter;
    2. thread_yield();
    3. counter = val;
    4. thread_exit();
}

void baz(void *ptr) {
    1. int val = counter;
    2. counter = val;
    3. thread_yield();
    4. thread_exit();
}
```

foo : 1
bar : 1
bar : 2
baz : 1
baz : 2
baz : 3
foo : 2
foo : 3
foo : 4
bar : 3
bar : 4
baz : 4

Q9

multi-segments heap
smalloc:

```
struct list {
    int value; int idx;
    struct list *next;
};

int readval(struct list *node, int idx) {
    return node->value;
}

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

```
readval:
 mov 4(%esp), %eax
 mov 8(%esp), %ecx
 mov %cx, %fs
 mov %fs:(%eax), %eax
 ret
```

Segmentation

- What is the problem with multi-segments heap?

Segmentation

- What is the problem with multi-segments heap?
  - The user has to carry the [segment, virtual address] pairs throughout the program
  - The programming model is not easy

Segmentation

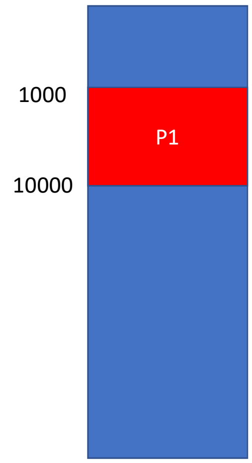
- Why do we need multi-segments heap?

Segmentation

- Why do we need multi-segments heap?
  - Applications may need more memory at runtime
  - A segment is a contiguous area of memory
  - Consecutive memory may not be available at runtime
  - need to relocate the entire segment at runtime

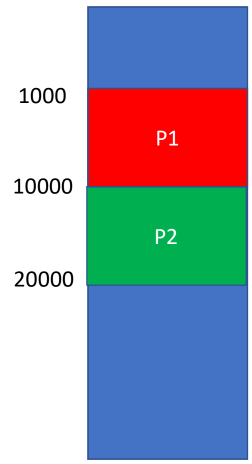
Segmentation

- P1 is loaded in RAM



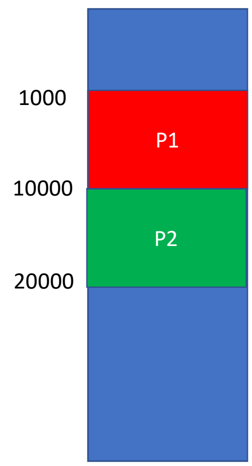
Segmentation

- P1 is loaded in RAM
- P2 is loaded in RAM



Segmentation

- P1 is loaded in RAM
- P2 is loaded in RAM
- P1 needs 1000 more bytes
    - contiguous memory is not available

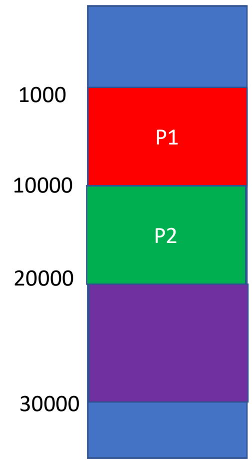


Segmentation

- P1 is loaded in RAM

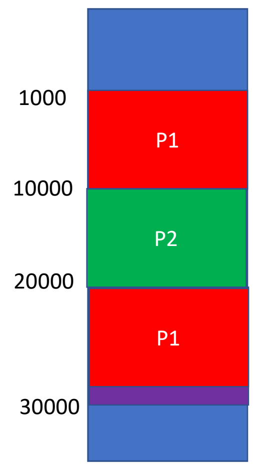
- P2 is loaded in RAM

- P1 needs 1000 more bytes
  - contiguous memory is not available
  - new area [20000-30000] is allocated for P1



Segmentation

- P1 is loaded in RAM
- P2 is loaded in RAM
- P1 needs 1000 more bytes
  - contiguous memory is not available
  - new area [20000-30000] is allocated for P1
  - P1 is copied to the new area
    - how many bytes are copied? 9000

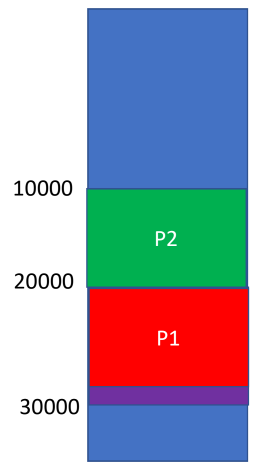


Segmentation

- P1 is loaded in RAM

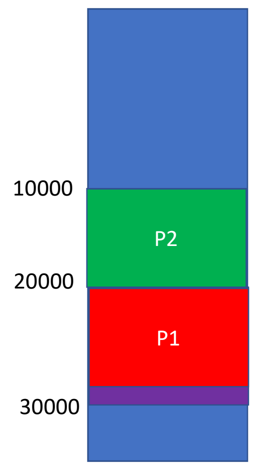
- P2 is loaded in RAM

- P1 needs 1000 more bytes
  - contiguous memory is not available
  - new area [20000-30000] is allocated for P1
  - P1 is copied to the new area
    - how many bytes are copied? 9000
  - Old memory of P1 is freed



Segmentation

- P1 is loaded in RAM
- P2 is loaded in RAM
- P1 needs 1000 more bytes
  - contiguous memory is not available
  - new area [20000-30000] is allocated for P1
  - P1 is copied to the new area
    - how many bytes are copied? 9000
  - Old memory of P1 is freed
  - base address of user-segment in GDT is updated
    - what was the old base? 1000
    - what is the new base? 20000
  - P1 is resumed



Segmentation

- Do the addresses of existing pointers are the same after relocating the process to a new memory area?

Segmentation

- Do the addresses of existing pointers are the same after relocating the process to a new memory area?
  - Yes, although the base address of the segment is changed the virtual addresses in the process are same
  - The pointers in the process' address space are the virtual addresses, which remain the same
  - The program works perfectly after relocation

Segmentation

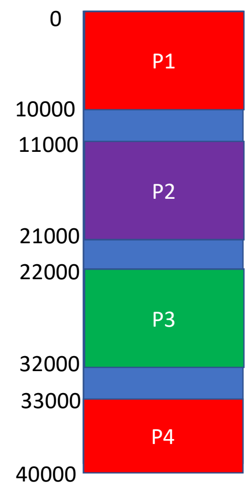
- Why is relocation bad?

Segmentation

- Why is relocation bad?
  - Need to copy the entire process' memory
  - Sometimes, we may also need to relocate other processes

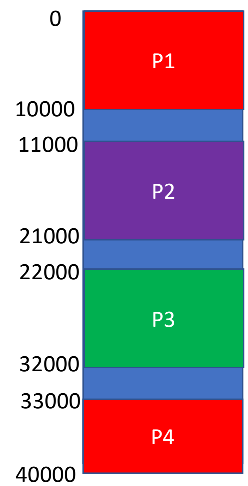
Segmentation

- Current snapshot of RAM



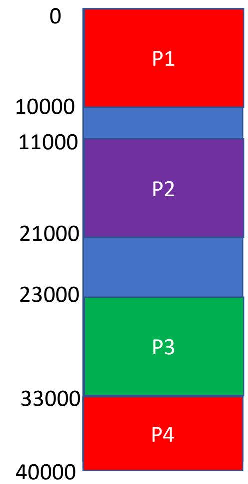
Segmentation

- Current snapshot of RAM
- P1 needs extra 3000 bytes



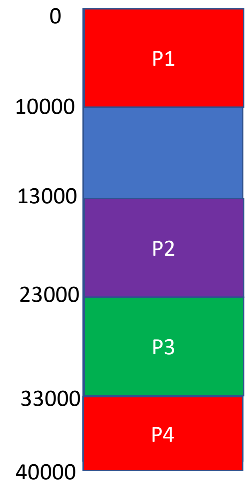
Segmentation

- Current snapshot of RAM
- P1 needs extra 3000 bytes
  - shifting P3



Fragmentation in segmentation

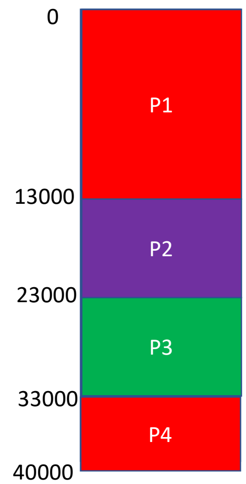
- Current snapshot of RAM
- P1 needs extra 3000 bytes
  - shifting P3
  - shifting P2
- Now 3000 contiguous bytes are available



Fragmentation in segmentation

- Current snapshot of RAM
- P1 needs extra 3000 bytes
  - shifting P3
  - shifting P2
- Now 3000 contiguous bytes are available
- P1's address space is extended
- P1 can resume
- Do we need to change the base of P1?

No



Segmentation

- Any other problem with segmentation?

Segmentation

- Any other problem with segmentation?
  - The virtual address space of the process is limited to the total RAM size
  - What if the process needs more memory than the actual RAM?

Paging

- To mitigate the problem with the segmentation, a new MMU hardware is introduced called the paging hardware

- The basic idea is to divide the process address space into fixed-size memory regions (called pages)

- The MMU maintains a table that converts a VA to the PA

MMU

- There is no way to disable segmentation

- However, the segmentation can be effectively disabled using a simple trick?

MMU

- There is no way to disable segmentation
- However, the segmentation can be effectively disabled using a simple trick?
- Add only one entry to the GDT whose base and limit is set to 0 and 0xFFFFFFFF respectively that can be accessed by both user and kernel

MMU

- When paging hardware is active, the segmentation hardware translates a virtual address to the linear address
  - by adding the base of the segment to the virtual address
- The linear address is converted into the physical address by the paging hardware
- When paging is not active, the linear addresses are equal to the physical addresses

Example

idx	base	limit	DPL
0	0	0xffffffff	3

OS and applications both use index 0.

What is the linear address corresponding to VA 1000? 1000

What is the linear address corresponding to VA 3000? 3000

What is the physical address corresponding to VA 1000?

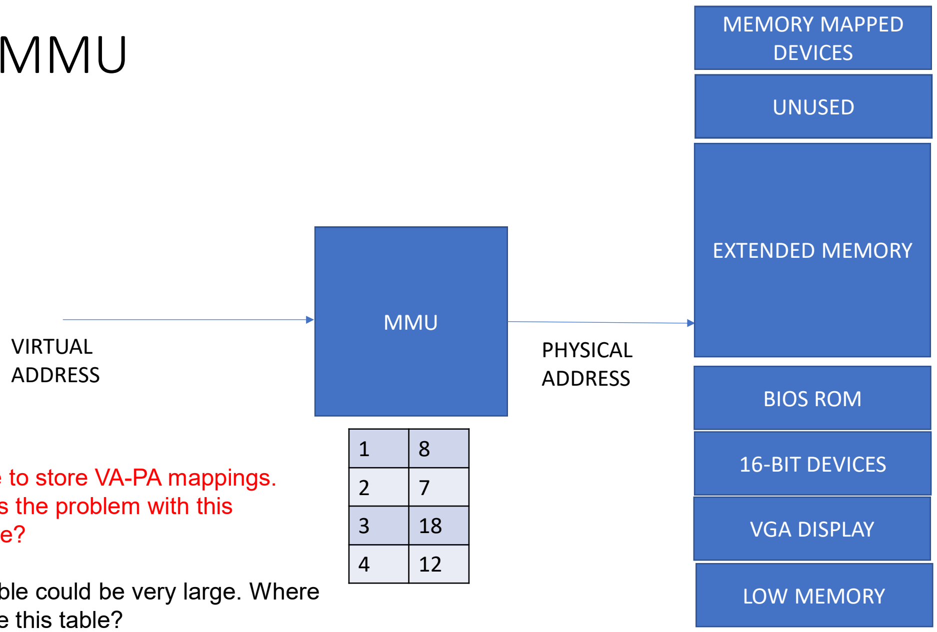
if paging is disabled 1000

If paging is enabled, the physical address will be calculated by the paging hardware.

MMU

- We will use virtual address instead of linear address in our page table discussion because in most OSes they are the same (using the trick discussed earlier)

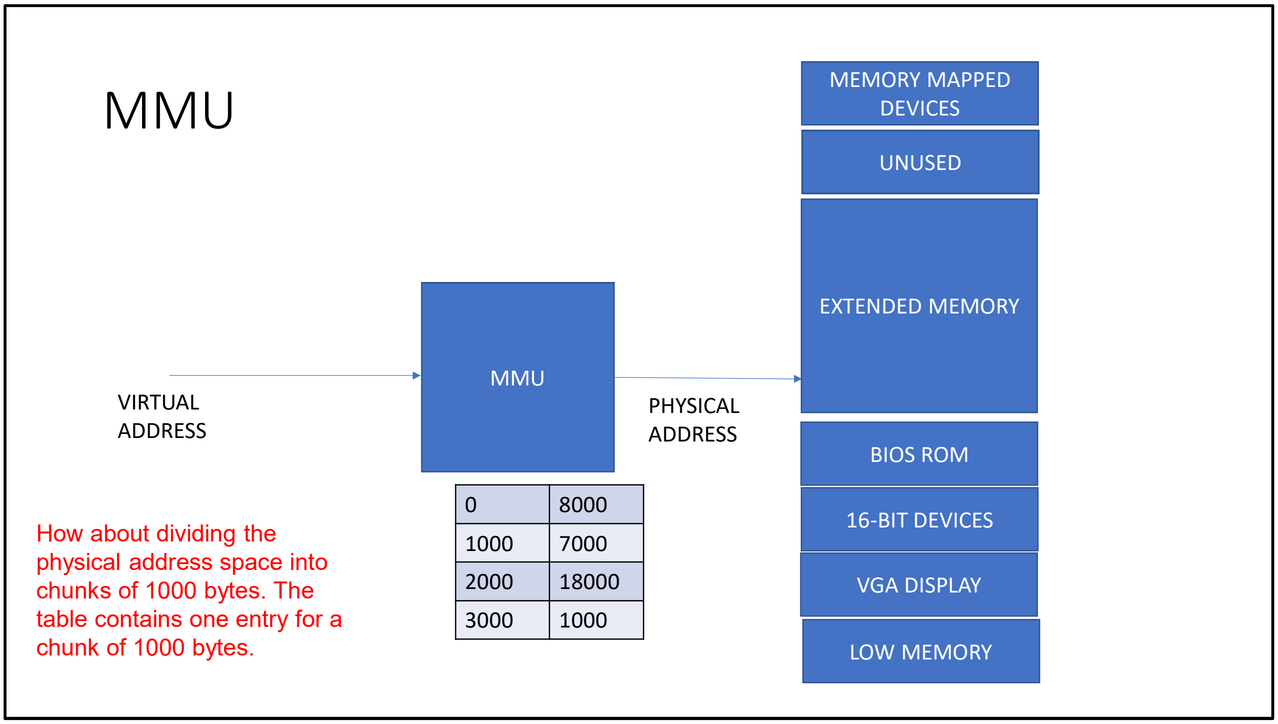
MMU



1	8
2	7
3	18
4	12

A table to store VA-PA mappings. What is the problem with this scheme?

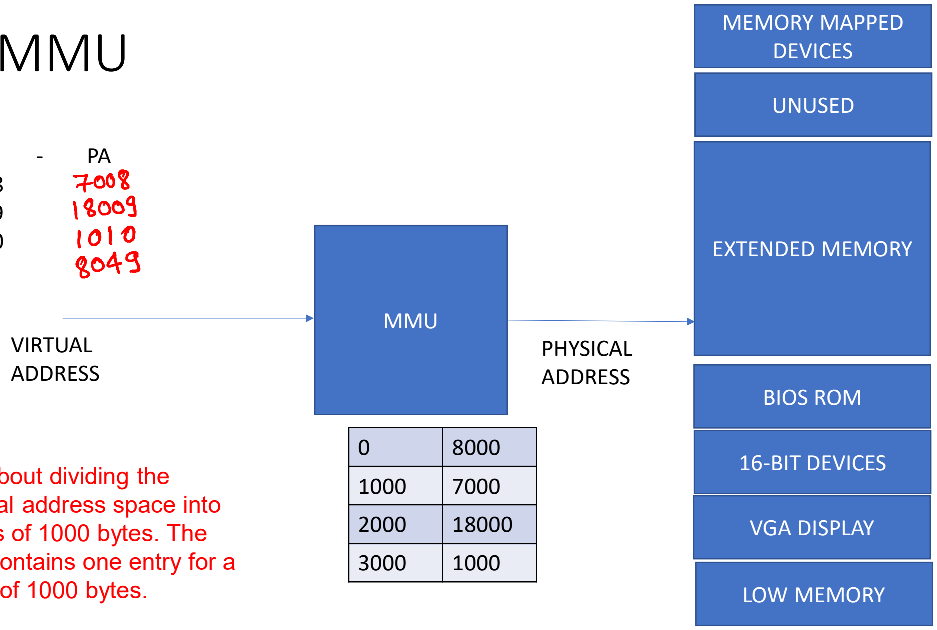
The table could be very large. Where to store this table?



The physical address space and virtual address space both are 4 GB long [0 – 0xFFFFFFFF]. Let us divide both of them into the chunks of 1000 bytes. The MMU keeps a mapping from a virtual chunk address to the physical chunk address. The offsets in the virtual and physical chunks are the same.

MMU

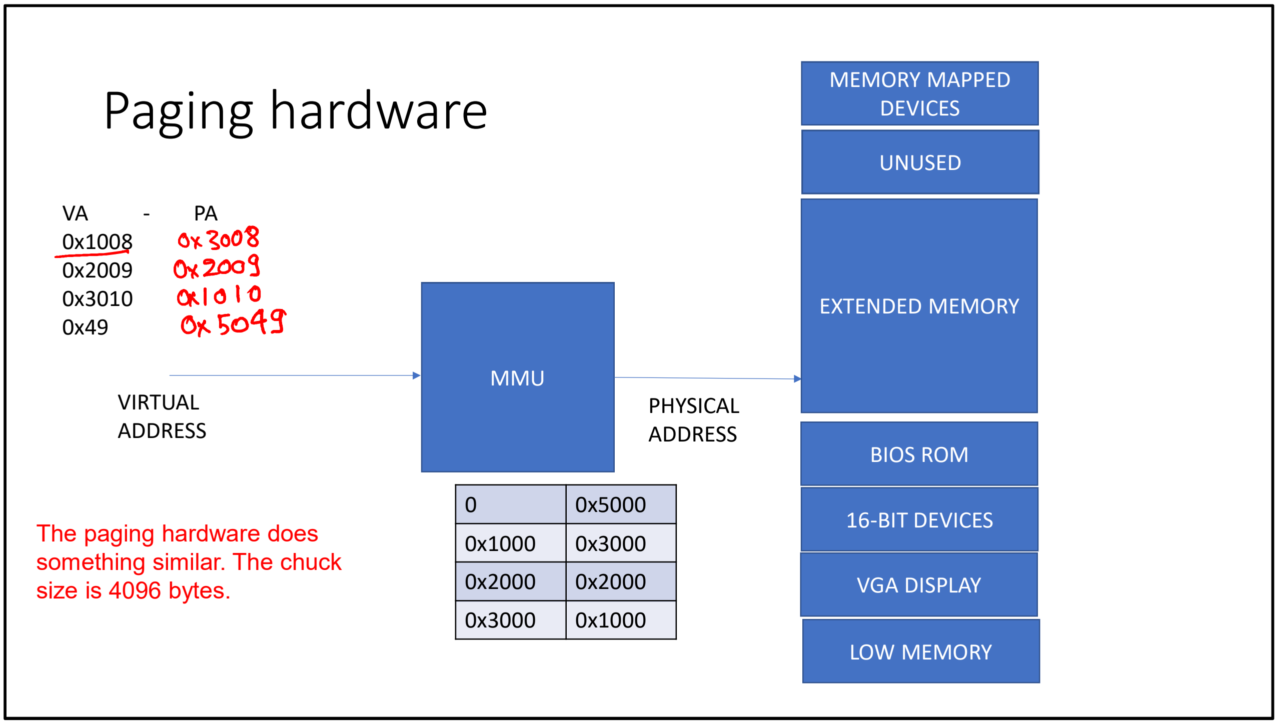
VA	-	PA
1008		7008
2009		18009
3010		1010
49		8049



0	8000
1000	7000
2000	18000
3000	1000

How about dividing the physical address space into chunks of 1000 bytes. The table contains one entry for a chunk of 1000 bytes.

VA 1000 is chunk 1. The physical address of 1008 would be 7008 because the chunk offsets in virtual and physical addresses are the same.



The actual paging hardware does something similar. It divides the virtual and physical address space into chunks of 4096 (0x1000) bytes. These chucks are also called pages. The virtual chunk is called a virtual page, and the physical chunk is called a physical page. The MMU maintains a mapping from a virtual page to the physical page. The page offsets in a virtual and physical address are the same. The starting address of a virtual or physical page is always aligned to 4096 bytes. The PA corresponding to VA 0x1008 is 0x3008.

Paging hardware

- The virtual address space is the same as physical address space [0 - 2^{32}-1]
- The physical address space is divided into 4096 bytes chunks called physical pages
  - The page address is 4096-byte aligned
- The virtual address space is divided into 4096 bytes chunks called the virtual pages
- The OS keeps a mapping from a virtual page to the physical page
  - The offsets within the virtual and physical pages are the same

Offsets

- Last 12-bits in a virtual address is the offset in the virtual page
- Last 12-bits in a physical address is the offset in the physical page
- Because the offsets within the virtual and physical pages are the same, the last 12 bits of physical and virtual addresses are the same
    - We don’t need to store them in the MMU table

Page table

- The MMU maintains the mapping from virtual page number (VPN) to physical page number (PPN) in the page table

- How many bits are required to store a VPN-PPN mapping?
  - 40 bits, 20 bits for VPN + 20 bits for PPN

Page table

- the page table can be implemented using an array
- VPN is the index in the page table
- PPN is stored in the page table
  - let us assume PPN is stored in 32-bits

```
unsigned *page_table = malloc(x);

unsigned va_to_pa(unsigned va) {
  unsigned off = (va & 0xfff);
  unsigned vpn = (va >> 12);
  return (page_table[vpn] << 12) | off;
}
```

what is the value of x? = 4 MB

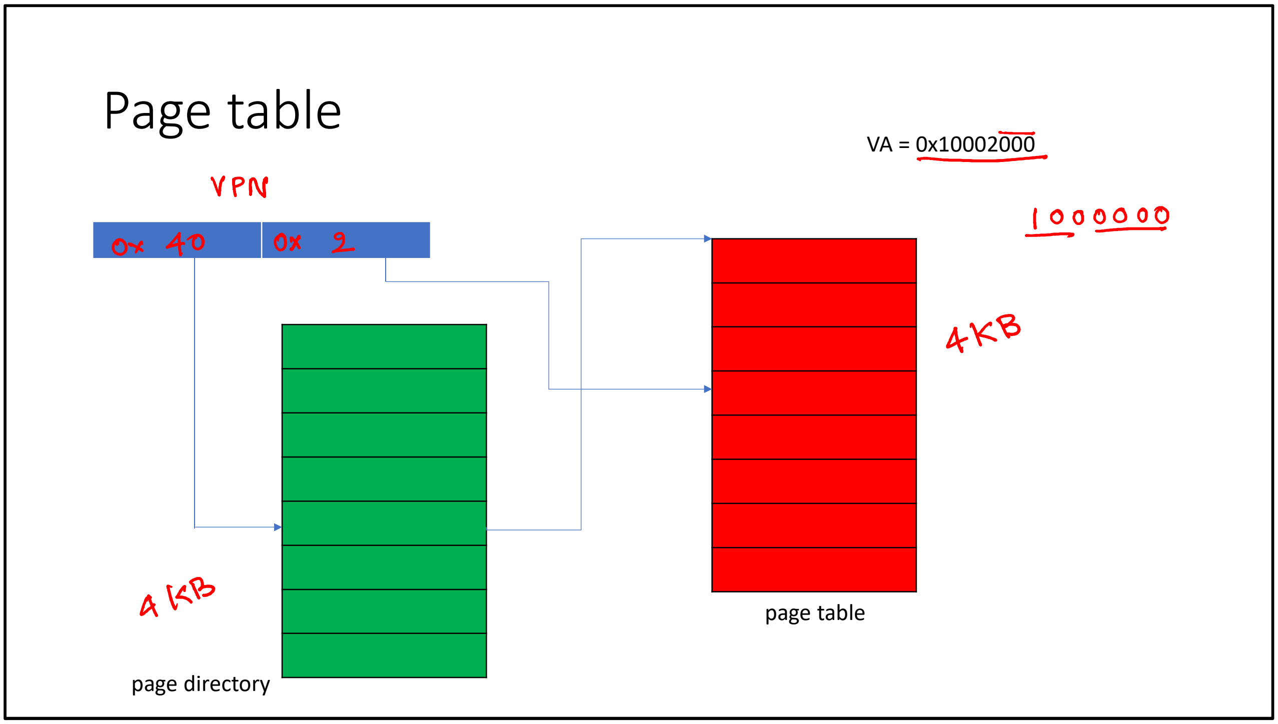
Let us discuss a straightforward implementation of a page table. Here page table is an array of unsigned values that contain the PPNs. The PPN corresponding to a VPN is stored at index whose value is equal to VPN. Because the total number of VPNs is $2^{32}/4096 = 2^{20}$, the total memory required for the page table would be $4 * 2^{20}$ = 4 MB. The va_to_pa routine takes a virtual address and returns the physical address stored in the page_table. VPN is calculated by right shifting the va by 12 bits (top 20 bits of the va). Similarly, the PPN stored in the page table is left-shifted 12-bits to calculate the starting address of the physical pages. Finally, the virtual page offset is added to the address of the physical page to calculate the physical address.

Page table

- The OS creates a page table for every process
- a lot of entries in the page table are never used
  - most applications don't use the entire virtual address space
  - the virtual address space is same as physical address space
    - i.e., [0- 0xFFFFFFFF]
- Allocating space for all VPNs is wastage of memory

Page table

- two-dimensional page tables
  - The top 10-bits of the VA are used to index in a page directory
  - page-directory contains the physical addresses of a page table
  - The next 10-bits (after top 10 bits) are used to index in the page table
  - The corresponding entry in the page table contains the physical address



The top 10 bits of VA 0x10002000 (i.e., 0x40) are indexed in the page directory to fetch the address of the page table. The next 10-bits (after top 20 bits) of VA 0x10002000 (i.e., 0x2) are indexed in the page table (calculated from the page directory) to fetch the address of the physical page.

Page table

- From next class onwards, bring the xv6 code-listing in the class