Paging is another way of providing memory isolation.
In the paging scheme, both virtual address space and physical address space are of 4 GB (2^32).

Software reads/writes to VA.
Hardware translates the VA to PA (RAM address).
Who creates VA – PA mapping? OS
The virtual and physical address spaces are divided into pages.
Each page is of 4 KB size, and page address is also 4 KB aligned (multiple of 4 KB).
OS creates a mapping from virtual page to physical page into a table called page table.
The paging hardware walks this table during runtime to convert the virtual address into the physical address.
The last 12 bits (which is effectively page offsets) of virtual and physical addresses are same.
The page-table contain a mapping from higher 20 bits of VA (VPN) to higher 20 bits of PA (PPN).
The most significant 10 bits of VPN are indexed in a table called page directory to get the address of another table called page table. The least significant 10 bits of VPN are indexed into the page table to get the PPN.

Below is the pseudo code of VA to PA translation.

```
unsigned *page_directory;
unsigned *page_table;
page_directory_offset = VA >> 22;
page_table_PA = page_directory[page_directory_offset];
page_table = (unsigned*)PA_to_VA(page_table_PA);
page_table_offset = (VA >> 12) & 0x3ff;
PPN = page_table[page_table_offset] >> 12;
```

Each entry of page directory and page tables is of 32 bits. However, the PPN is only 20 bits.
What are other bits in the page directory/table entries?
The other bits are different flags that store additional information about the PPN.
Some of these bits are:

0 (P) Present: 1 PPN is valid, 0 PPN is not valid
1 (R/W) Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry
2 (U/S) User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry
5 (A) Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry
6 (D) Dirty; indicates whether software has written to the 4-KByte page referenced by this entry

Where is page table stored?
In RAM

Would it be reasonable to have a page table to be a sequential array to VPN to PPN mappings?
How big is this array?
(2^20) * 4 = 4 MB
Would waste a lot of memory for small programs.

A downside of two-dimensional page tables: require two memory reference for VA – PA translation
How does hardware know the address of page directory?
The address of page directory is stored in %cr3 register.

How memory isolation is done in Processes?
Separate page table for each process.

How can a page be shared among different processes?
OS can map the same physical page in different processes page tables.

The kernel is shared between all the processes.
The kernel pages are mapped in each process address space.
The kernel pages are protected by user/supervisor flag in the page table.

Can kernel be mapped at different virtual addresses in different processes?

If the kernel is mapped at different virtual addresses in different processes then the address of global variables (e.g., ready_list) will be different when the different processes do the system calls because a system call handler uses the user-page tables while executing the system calls.

Can we have a different page table for the kernel?
Because the hardware does not change the page table automatically on every interrupt/exception/system call, the kernel stacks and interrupt handler wrappers still need to be mapped in the process address space. The interrupt handler wrappers can explicitly load cr3 with kernel's page table and jump to actual interrupt handlers that are only mapped in the kernel's page table.

Overhead of page table:
Page table scheme requires three memory access corresponding to a virtual memory dereference. To reduce this overhead, on-chip TLB is used. The TLB is very small (typically 512 entries). TLB caches the VPN to PPN mappings. Before walking the page table, the hardware checks if a mapping exists in TLB. If yes (also called a TLB hit), the hardware does not have to walk the page table, and thus two additional memory accesses are saved. For most of the programs the TLB hit rates are very high (>99.9%), and thus paging does not impose a severe impact on performance. TLB entries are not automatically get updated when the page table entries are modified. However, "invlpg" instruction can be used to invalidate a TLB entry on page table update. Loading a page table entirely flush the TLB. On x86, 4 MB pages also exist. In this case only one VPN-PPN mapping is required for entire 4 MB space, that limits the number of TLB entries significantly.

Paging in xv6:
xv6 reserves 2 GB virtual address space for OS in every process address space. It assumes that the available RAM size is less than 2 GB, and maps the entire RAM at virtual address 2 GB in every process address space. In this way, xv6 creates a linear mapping between virtual and physical address.

```
#define KERNEL_BASE 0x80000000       // 2 GB
V2P(x) (KERNEL_BASE + x)             // virtual address to physical address
P2V(x) (x – KERNEL_BASE)             // physical address to virtual address
```

Lowest 1 MB (0 – 0x10000) of the physical address space is reserved for I/O.

Kernel maps its code and data pages starting at location 0x80100000, and uses the rest of memory pages for dynamic memory allocation (kernel stacks for user processes, processes page tables, user pages, etc.). xv6 provides two high-level routines for memory allocation: kalloc() and kfree().

kalloc allocates a 4 KB virtual page in the kernel address space, and kfree frees the virtual page allocated by kalloc.

kinit2 and kinit1 take a memory range and frees all the virtual pages in this range. kinit1 and kinit2 are called by xv6 to add all virtual pages from the end of kernel data to RAM size in the freelist.

kfree() maintains a freelist of all freed virtual pages. Each free virtual page contains the address of the next virtual page.

Let us see how xv6 can create a page table using these APIs.

walkpgdir takes a page directory and a virtual address and returns the address of the page table entry where the physical page number is stored.

```
walkpgdir (unsigned *pgd, unsigned va) {
   int pgd_index = va >> 22;   /* compute the offset in the page directory */

   unsigned pgd_entry = pgd[pgd_index];  /* Fetch the PPN of the page table */

   unsigned *pgtable;
   if (!(pgd_entry & PTE_P)) /* page directory entry is not valid */
   {
       pgtable = kalloc();       /* allocate a page table page */
       memset(pgtable, 0, PAGE_SIZE);   /* set the contents of page table page to zero */

       /* set the page directory entry to point to page table */
       /* V2P(pgtable) is used to get the physical page */
       /* PTE_P, PTE_W, PTE_U are other flags that makes this entry valid, writable, and walkable by
            the hardware during the execution of user programs. */

       pgd[pgd_index] = V2P(pgtable) | PTE_P | PTE_W | PTE_U;
   }
   else
   {
       /* the page directory entry is valid */
       /* get the virtual address of the page directory */
       pgtable = P2V(PTE_ADDR(pgd_entry));
   }
   unsigned pgtable_idx = (va >> 12) & 0x3ff; /* compute the page table index */
   return &pgtable[pgtable_idx];   /* return the address of the page table entry corresponding to va */
}
```

mapkernelpage routine maps pa to virtual address va in a page table whose page directory is pgdir.

```
mapkernelpage(unsigned *pgdir, unsigned va, unsigned pa) {
     /* find the location of the page table entry corresponding to va */
    unsigned *pte = walkpgdir(va);
    /* update the page table entry with the physical address (last 12 bits must be zero) */
    *pte = pa | PTE_P | PTE_W;
}
```

The create_page_table routine creates a page directory and maps the RAM of size PHYSEND at the KERNEL_BASE.

```
#define KERNEL_BASE  0x80000000
#define PHYSEND  RAM_SIZE

create_page_table()
{
   /* allocate a page for page directory */
   unsigned *pgdir = kalloc();
   unsigned va;
   /* set the contents of page directory to zero */
   memset (pgdir, 0, PAGE_SIZE);
   /* map every physical page “x” between (0-PHYSEND)
        to its virtual address, i.e., V2P(x) */
   for (va = KERNEL_BASE; va < P2V(PHYSEND); va+=PAGE_SIZE)
             mapkernelpage(pgdir, va, V2P(va));
}
```

load_page_table loads the page table in the memory for hardware use:

```
load_page_table(unsigned *pgdir)
{
    /* set the %cr3 register to the physical address
        of the page directory */
     load_cr3 (V2P(pgdir));
}
```

So far, we have discussed how to the load the kernel in the address space of each page table. We will discuss in future lectures the other components of the process address space.

Next, we discuss how OS can optimize the implementation of the fork system call. It looks like that fork system call has to create a new address space for child process and copy the contents of all the pages from the parent process to the pages of the child process. However, because most of the times fork and exec are called one after another, it does not make sense, first to copy the parent process and then suddenly overwrite everything with a completely new application. To prevent the unnecessary copying the OS map the same physical pages in the parent and child process but revokes write access from them in both the processes. When a child or parent actually try to write to these pages, the OS makes another copy and map them with the write permission enabled. This optimization is called copy on write optimization.

We will discuss in next lectures, how the operating system detects writes on read-only pages.