📑 Xv6 Memory Management Walkthrough

[This is still under construction]

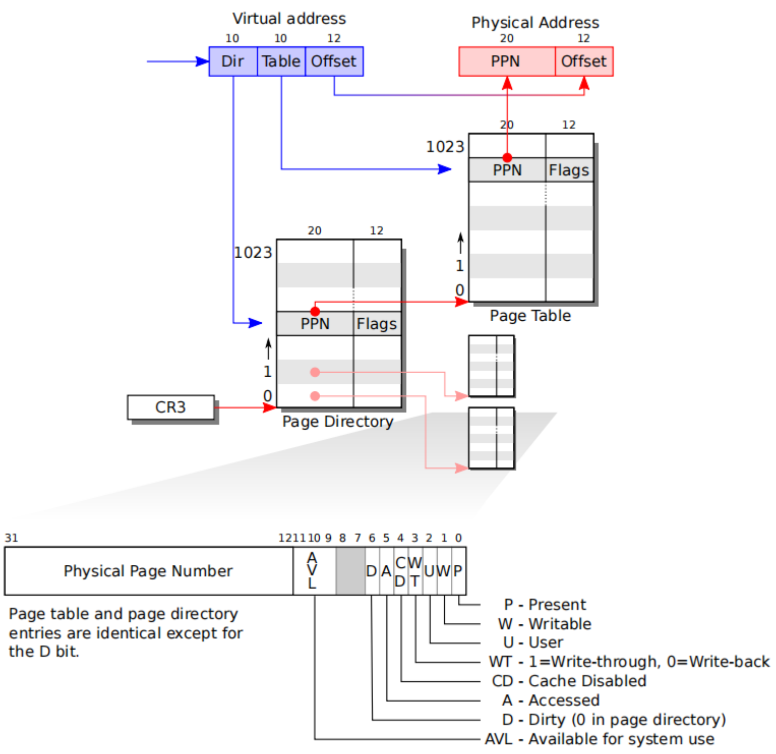
Terminology

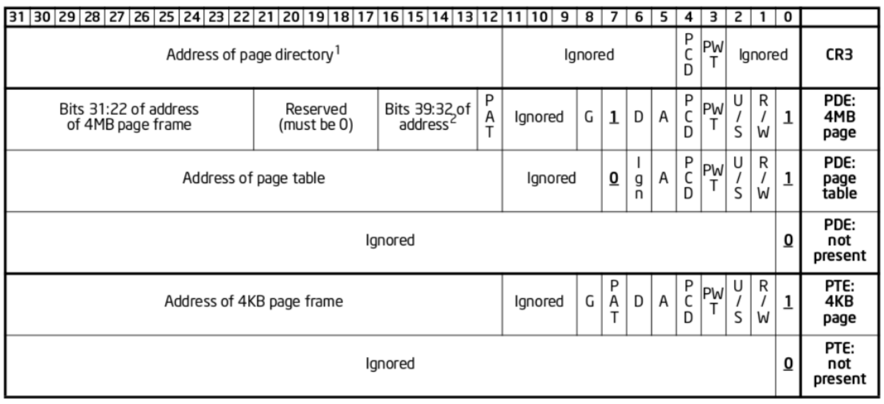
- VA : Virtual Address
- PA : Physical Address
- PD : Page Directory
- PDE : Page Directory Entry
- PT : Page Table
- PTE : Page Table Entry
- PPN : Physical Page Number

Overview of page table

All process works on virtual address. Machine's RAM is physical address. Page Table maps virtual address to physical address.

Xv6 does this in 2 steps.

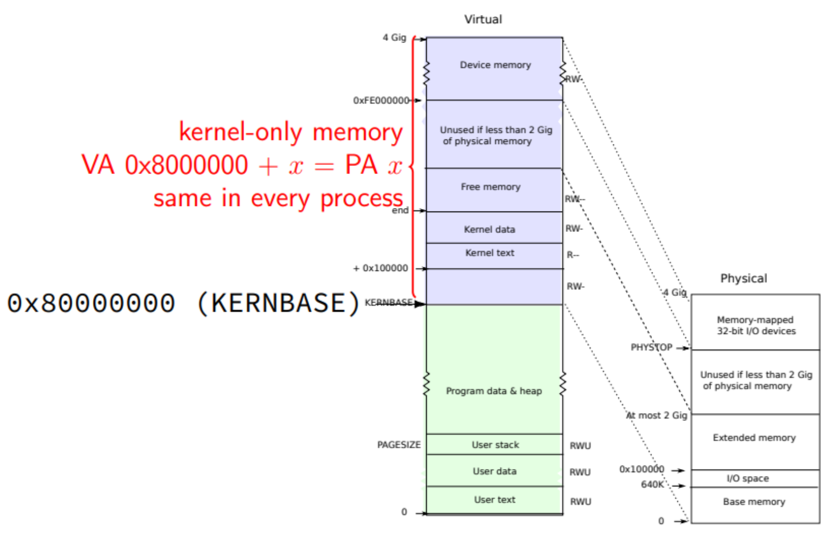




31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored																															0	**PTE: not present**

Memory Layout

Kernel Only Memory

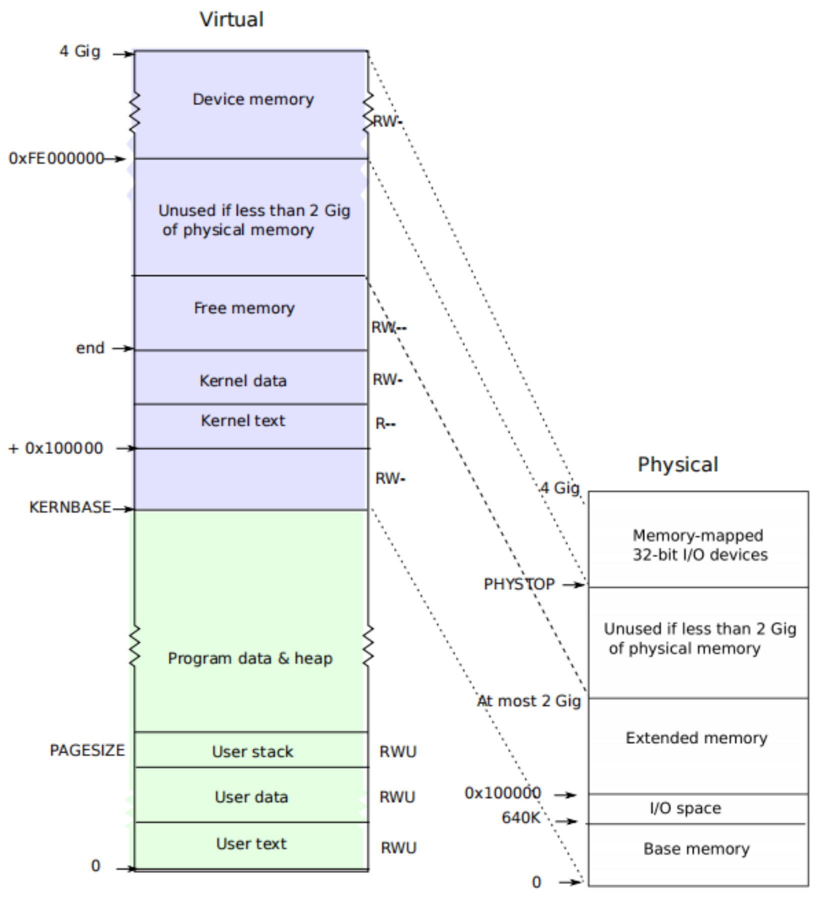


- virtual memory > `KERNBASE` (0x8000 0000) is for kernel

- always mapped as kernel-mode only
    - check `PTE_U` fag
    - **protection fault** for user-mode programs to access
- **physical memory address** `N` is mapped to `KERNBASE+N` or `0:PHYSTOP` to `KERNBASE:KERNBASE+PHYSTOP`
    - Note that although the size of physical memory is 4 GB, only 2 GB can be used by Xv6
- kernel code loaded into contiguous physical addresses

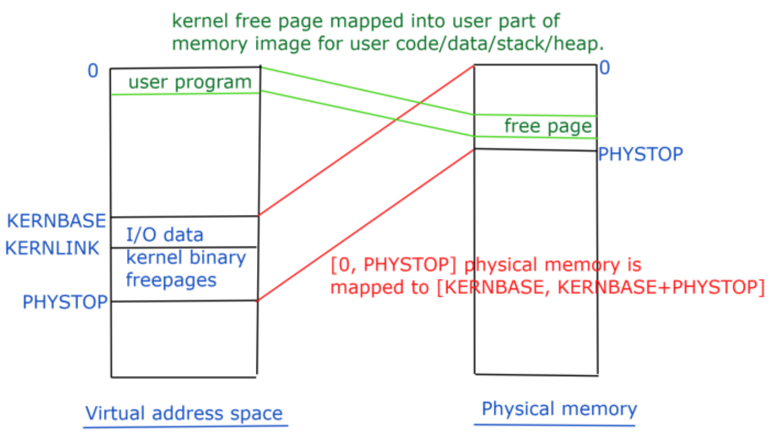
User Only Memory

Each process has separate page table. For any process, user memory VA range is `0:KERNBASE` where `KERNBASE` is 0x80000000 i.e. 2 GB of memory is available to process.



- The kernel code doesn't exactly begin at `KERNBASE`, but a bit later at `KERNLINK`, to leave some space at the start of memory for **I/O devices**. Next comes the kernel **code+read-only** data from the kernel binary. Apart from the memory set aside for kernel code and I/O devices, the **remaining memory is in the form of free pages** managed by the kernel. When any user process requests for memory to build up its user part of the address space, the kernel allocates memory to the user process from this free space list. That is, most physical memory can be mapped twice, once into

README.md


kernel free page mapped into user part of memory image for user code/data/stack/heap.
0
user program
KERNBASE
KERNLINK
I/O data
kernel binary
freepages
PHYSTOP
0
free page
PHYSTOP
[0, PHYSTOP] physical memory is mapped to [KERNBASE, KERNBASE+PHYSTOP]
Virtual address space
Physical memory

P2V and V2P

- `V2P(a)` (virtual to physical)
    - convert **kernel address** `a` to **physical address**
        - subtract `KERNBASE` (0x8000 0000)

```
#define V2P(a) (((uint) (a)) - KERNBASE)
```

- `P2V(a)` (physical to virtual)
    - convert **physical address** `a` to **kernel address**
        - add `KERNBASE` (0x8000 0000)

```
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
```

Functions of interest {Part 1 : building blocks}

PGROUNDUP

What it does is round the address up as a multiple of page number i.e. do a CEIL type operation.

```
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
```

`PGROUNDUP(620)` → ((620 + (1024 -1)) & ~(1023)) → 1024

PGROUNDDOWN

another related function is `PGROUNDDOWN` , works similarly. just makes the address round down as a multiple of page number

```
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

`PGROUNDDOWN(2400)` → (2400 & ~(1023)) → 2048

switchuvm

- `u` here stands for user
- basically OS loads the user process information to run it

- loads process's Page Table to `%cr3`

`kalloc`

This function is responsible to return an address of one **new, currently unused** page (4096 byte) in RAM. `kalloc` removes first free page from `kmem` and returns its (virtual!) address, where `kmem` points to the head of a list of free (that is, available) pages of memory.

Failure : If it **returns 0**, that means there are no available unused pages currently.

`walkpgdir`

Main job of this function is to get the content of 2nd level PTE. This is such an important function that we will go line by line.

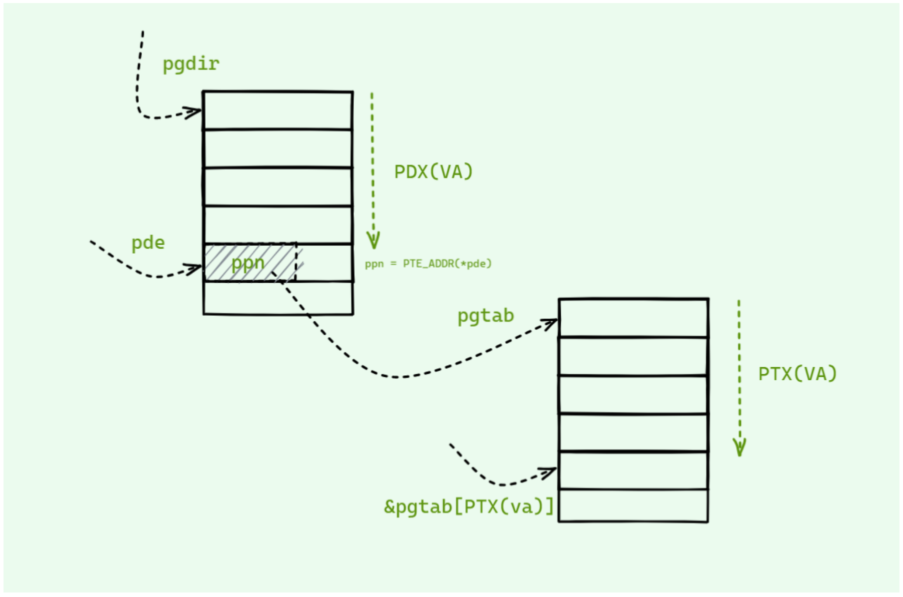
```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)]; // PDX(va) returns the first 10 bit. pgdir is level
  if(*pde & PTE_P){ // not NULL and Present
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde)); // PTE_ADDR return the first 20 bit
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)]; // PDX(va) returns the second 10 bit. So, pgtab[PT
}
```

Case 1 : when no allocation needed (`alloc = 0`)

Recall that

```
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/
```



- `PDX(va)` returns the first 10 bit. `pgdir` is level 1 page table. So, `pgdir[PDX(va)]` is level 1 PTE where there is PPN and offset

```
pde = &pgdir[PDX(va)];
```

- `PTE_ADDR` return the first 20 bit or PPN. PPN is converted to VPN using `P2V` for finding 2nd level PTE. `pgtab` is level 2 page table

```
pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
```

- `PDX(va)` returns the second 10 bit. So, `pgtab[PTX(va)]` is level 2 PTE where there is PPN and offset

```
&pgtab[PTX(va)];
```

Case 2 : creating second-level page tables (When `alloc = 1`)

- return **NULL** if **not trying to make new page table** otherwise use `kalloc` to allocate it

```
if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
    return 0;
```

- clear the new second-level page table. Make sure all those **`PTE_P`** bits are zero.

```
memset(pgtab, 0, PGSIZE);
```

- now that we have made the 2nd level page table, we have to put that address in 1st level page table. plus add some flags

```
*pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
```

`mappages`

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;
```

```
    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
      if((pte = walkpgdir(pgdir, a, 1)) == 0)
        return -1;
      if(*pte & PTE_P)
        panic("remap");
      *pte = pa | perm | PTE_P;
      if(a == last)
        break;
      a += PGSIZE;
      pa += PGSIZE;
    }
    return 0;
  }
```

As the name suggests, `mappages` adds a mapping from the virtual address (starting from given `VA` to `VA+SIZE`) to the physical address (starting from given `PA` to `PA+SIZE`)

- for each virtual page in range, **get its page table entry**. if 2nd level not present allocate it (recall `alloc=1` case of `walkpgdir`). failure happens if runs out of memory

```
  if((pte = walkpgdir(pgdir, a, 1)) == 0)
        return -1;
```

- make sure it's not already set

```
  if(*pte & PTE_P)
                panic("remap");
```

- set page table entry to valid value pointing to physical page at `PA` with specified permission (`perm`) and `P` for present

```
  *pte = pa | perm | PTE_P; // pa is first 20 bit, perm is flags, PTE_P is
```

- advance to next physical page (`PA`) and next virtual page (`VA`)

```
  a += PGSIZE;
  pa += PGSIZE;
```

Functions of interest {Part 2 : core functions}

allocuvm

Allocates user virtual memory (page tables and physical memory) to grow process. This function is responsible to **increase** the user's virtual memory in a specific page directory from `oldsz` to `newsz` . This function used for initial allocation plus expanding heap on request

```
  int
  allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
  {
    char *mem;
    uint a;

    if(newsz >= KERNBASE)
      return 0;
    if(newsz < oldsz)
      return oldsz;
    a = PGROUNDUP(oldsz);
    for(; a < newsz; a += PGSIZE){
      mem = kalloc();
      if(mem == 0){
        cprintf("allocuvm out of memory\n");
        deallocuvm(pgdir, newsz, oldsz);
        return 0;
      }
      memset(mem, 0, PGSIZE);
      if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
        cprintf("allocuvm out of memory (2)\n");
        deallocuvm(pgdir, newsz, oldsz);
        kfree(mem);
        return 0;
      }
    }
    return newsz;
  }
```

Returns `newsz` if succeeded, 0 otherwise.

- walks the virtual address space between the old size and new size in page-sized chunks. For each new logical page to be created, it allocates a new free page from the kernel, and adds a mapping from the virtual address to the physical address by calling `mappages`
- allocate a new, zero page

  ```
  mem = kalloc();
  ```

- add page to second-level page table, also do the allocation. recall that `mappages` call `walkpgdir` with `alloc = 1`

  ```
  mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
  ```

- There are indeed 2 cases where **this function can fail**:

  Case 1: `kalloc` (kernel allocation) function failed. This function is responsible to return an address of a **new, currently unused** page in RAM. If it **returns 0**, that means there are no available unused pages currently.

  Case 2: `mappages` function failed. This function is responsible of making the **new allocated page to be accessible by the process** who uses the given page directory by **mapping that page with the next virtual address available in the page directory**. If this function fails that means it failed in doing so, probably due to the page directory being already full.

  In both cases, `allocuvm` didn't managed to increase the user's memory to the size requested, Therefore, `deallocuvm` is undoing all allocations until the point of failure, so the virtual memory will remain unchanged, and returns an error it self.

deallocuvm

deallocuvm looks at all the logical pages from the (bigger) old size of the process to the (smaller) new size, locates the corresponding physical pages, frees them up, and zeroes out the corresponding PTE as well.

copyuvm

Once a child process is allocated, its memory image is setup as a complete copy of the parent's memory image by a call to `copyuvm`

- `setupkvm` : sets up kernel virtual memory

- walks through the entire address space of the parent in page-sized chunks, gets the physical address of every page of the parent using a call to `walkpgdir`
- allocates a new physical page for the child using `kalloc`
- copies the contents of the parent's page into the child's page, **adds an entry to the child's page table** using `mappages`
- returns the child's page table
- has similar failure cases as discussed in `allocuvm`

User Process Memory Management initialization

- `userinit(void)` → creates the first user process known as `init`
  - `setupkvm` : Set up kernel part of a page table
  - `inituvm` : allocates one physical page of memory, copies the `init` executable into that memory, and sets up a page table entry for the first page of the user virtual address space
  - When the `init` process runs, it executes the `init` executable, whose main function **forks** a shell and starts listening to the user

  I guess this explain why we get this output when we do `control + P` after xv6 boots

  ```
  sleep  init 80104347 801043f5 80104f5d 80106051 80105d93
  sleep  sh 80104310 801002ea 80101030 801050b6 80104f5d 80106051 80105d
  ```

- all other case (meaning for all other user processes)
  - created by the `fork` system call
    - calls `copyuvm` : memory image is setup as a complete copy of the parent's memory image. it return the child's page table
    - after `copyuvm` entire memory of the parent has been cloned for the child, and the child's new page table points to its newly allocated physical memory

Grow/shrink the userspace part of the memory image

- sbrk → sys_sbrk [SYSTEM CALL]
  - growproc
    - allocuvm : to grow
    - deallocuvm : to shrink
    - switchuvm

Resource

- Xv6 book
- Memory Management in Xv6
- https://www.cs.virginia.edu/~cr4bd/4414/F2018/slides/20181011--slides-1up.pdf
- https://www.cs.virginia.edu/~cr4bd/4414/F2019/slides/20191015--slides-1up.pdf
- https://iitd-plos.github.io/os/2020/lec/l15.html
- https://pdos.csail.mit.edu/6.828/2009/lec/l5.html
- http://course.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=onur-447-spring13-lecture18-virtual-memory-iii-afterlecture.pdf

- https://github.com/YehudaShapira/xv6-explained
- https://stackoverflow.com/questions/56258056/what-does-deallocation-function-in-xv6s-allocation-function
- https://www.cs.columbia.edu/~junfeng/11sp-w4118/lectures/mem.pdf