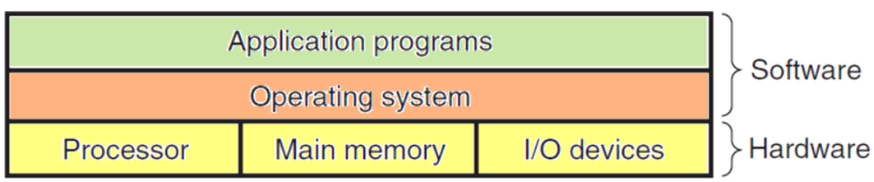
CSE320 System Fundamentals II
Virtual Memory

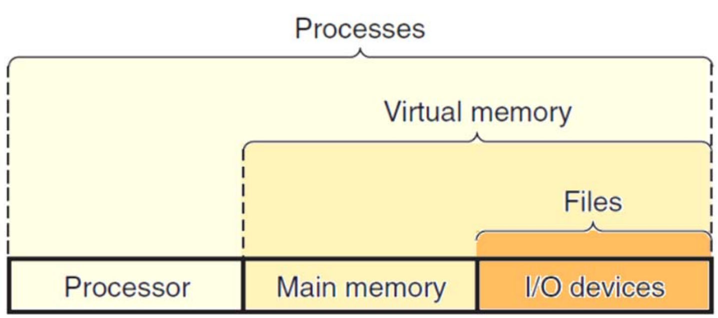
YoungMin Kwon

Virtual Memory

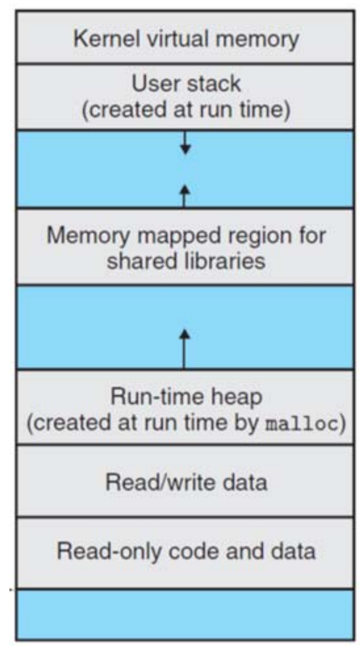
- Virtual Memory
  - Abstraction that provides each process the illusion that it has an exclusive use of the main memory



Layered view of a computer system.

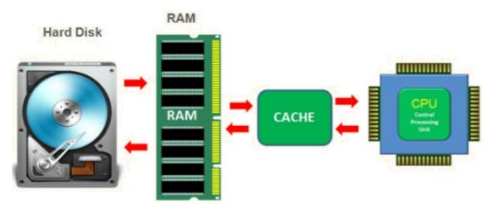


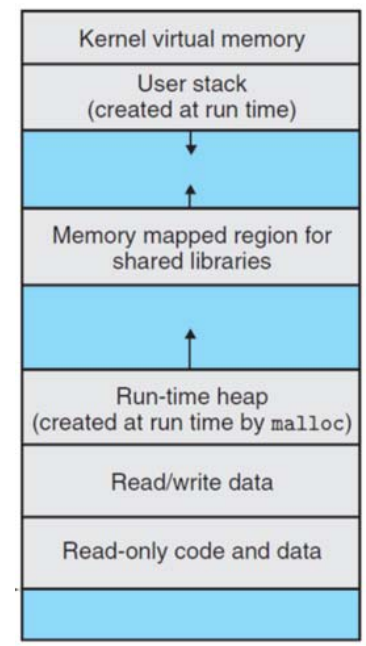
Abstractions provided by an operating system.



Virtual Memory

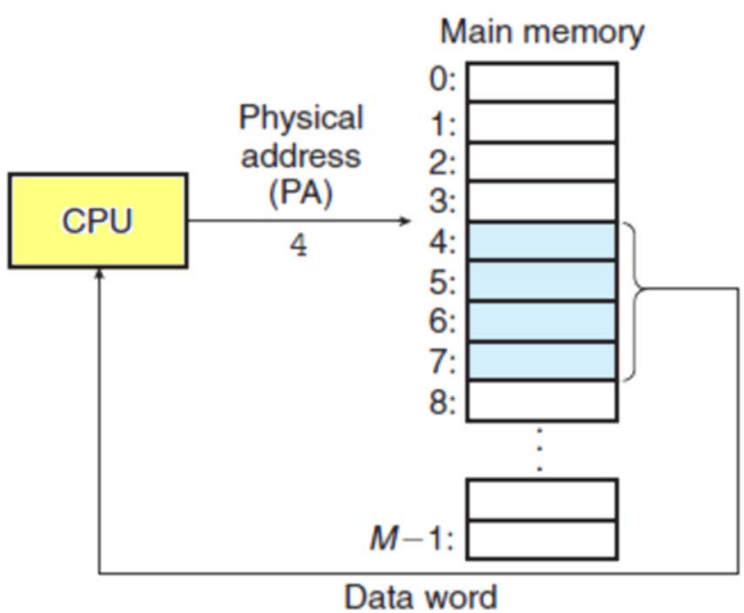
- Uses main memory efficiently
  - Cache for an address space stored on disks
  - Keeping only the active areas in main memory
- Simplifies memory management
  - Providing each process with a uniform address space
- Protects the address space
  - from corruption by other processes





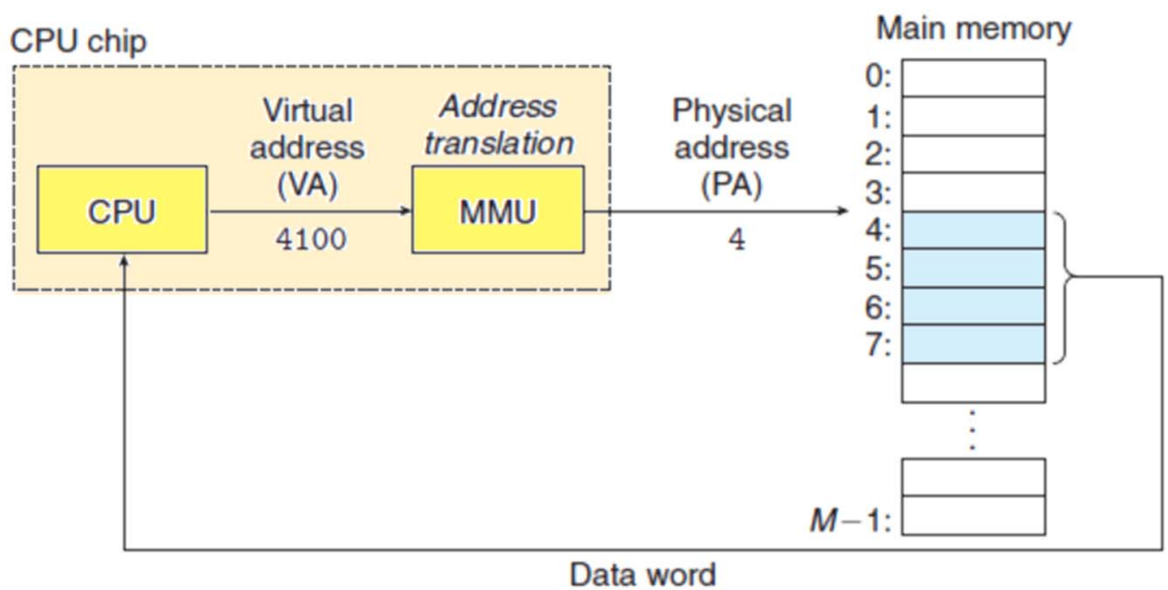
Physical and Virtual Addressing

- Physical addressing
  - CPU generates a physical address and passes it to main memory over the memory bus



Physical and Virtual Addressing

- Virtual addressing
  - CPU generates a virtual address
  - Virtual addresses are translated to physical addresses by the Memory Management Unit (MMU)

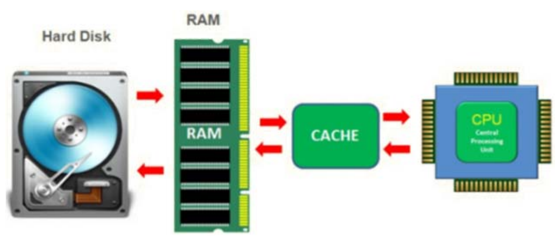


Address Space

- Address space
  - An ordered set of nonnegative addresses: {0, 1, 2, …}
- Linear address space
  - If the integers in the address space is contiguous
  - Virtual address space of N ($=2^n$) bytes: {0, 1, 2, …, N-1}
  - Physical address space of M($=2^m$) bytes: {0, 1, 2, …, M-1}
- Each byte of main memory has
  - a virtual address chosen from the virtual address space
  - a physical address chosen from the physical space

VM as a Tool for Caching

- Virtual memory
  - Organized as an array of N contiguous byte-size cells stored on disks
  - Contents of the array on disk are cached in main memory

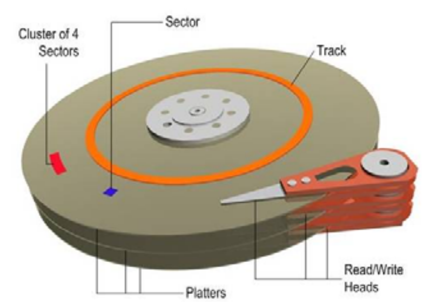


- Pages
  - Partition the virtual memory into blocks of size $P(=2^p)$ bytes, called pages, that serve as a transfer unit between disks and main memory

DRAM Cache Organization

- Disk vs DRAM
  - Disk is about 100,000 times slower than a DRAM
  - Reading the first byte from a disk sector is 100,000 times slower than reading successive bytes in the sector

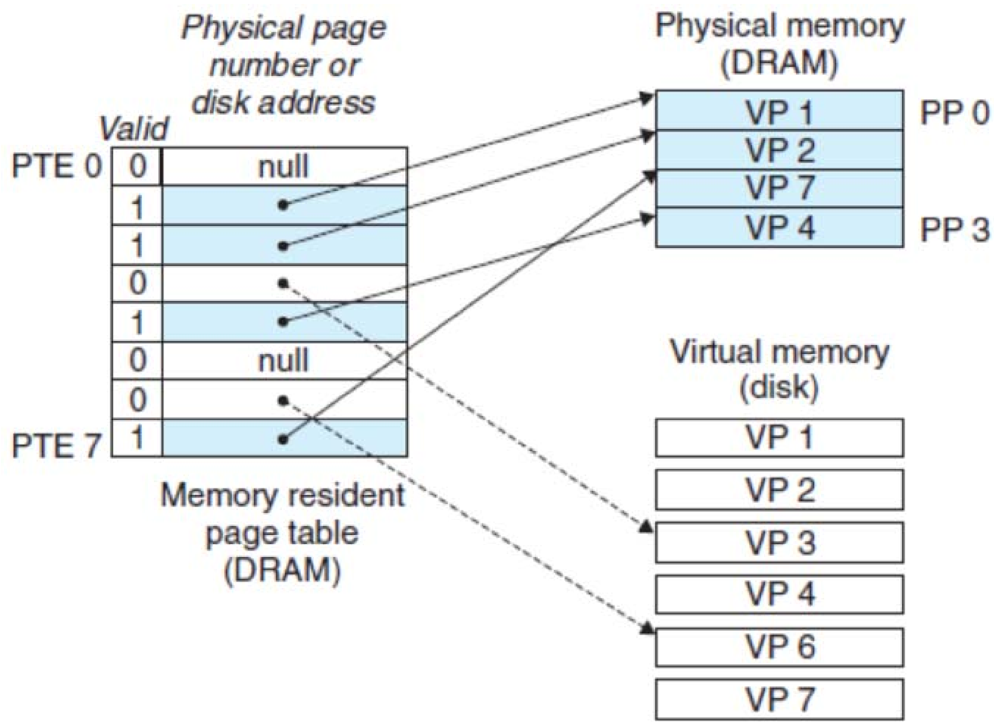
- Large miss penalty
  - Large virtual page (4KB ~ 2MB)
  - Fully associative cache
  - Sophisticated replacement algorithm
  - Write-back



Page Tables

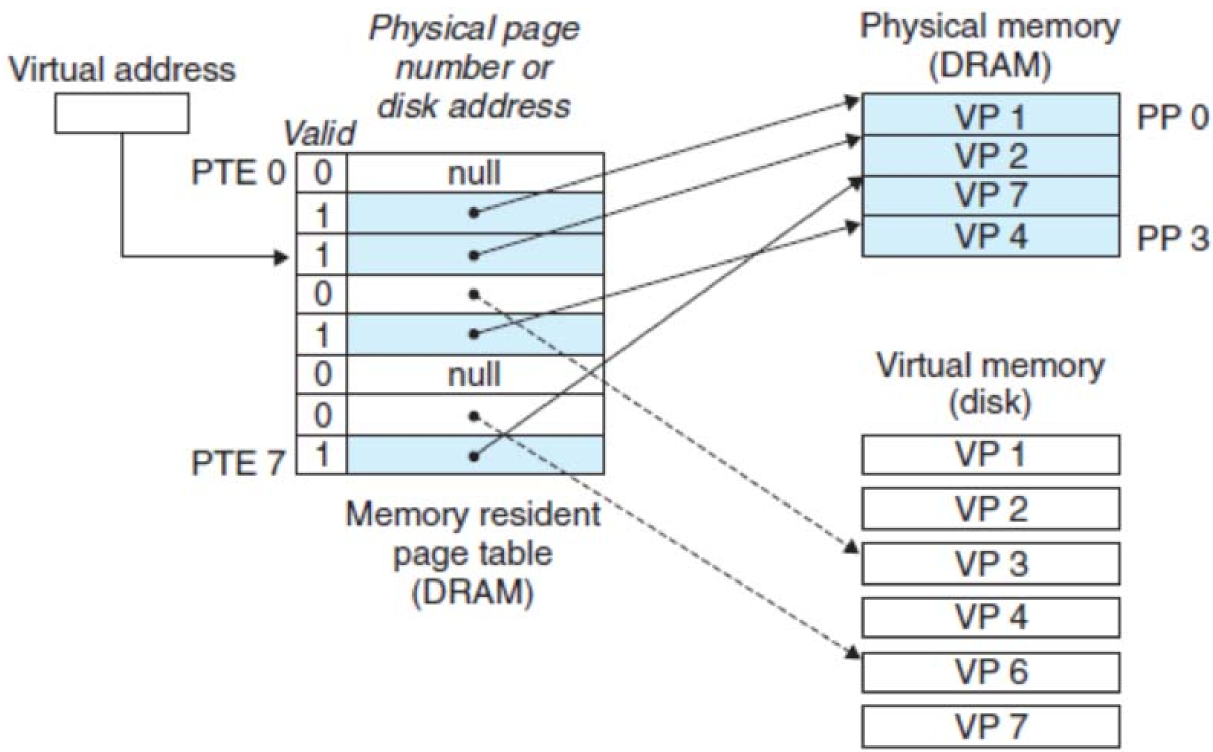
- Page table
  - Maps virtual pages to physical pages
  - Each process has a page table
- MMU
  - Reads the page table when it translates a virtual address to a physical address
- Operating system
  - Maintains the contents of the page table and transferring pages between disk and DRAM

Page Tables



- valid bit = 1: address ⇒ physical address
- valid bit = 0 & address = 0: page not allocated
- valid bit = 0 & address != 0: address ⇒ virtual page on disk

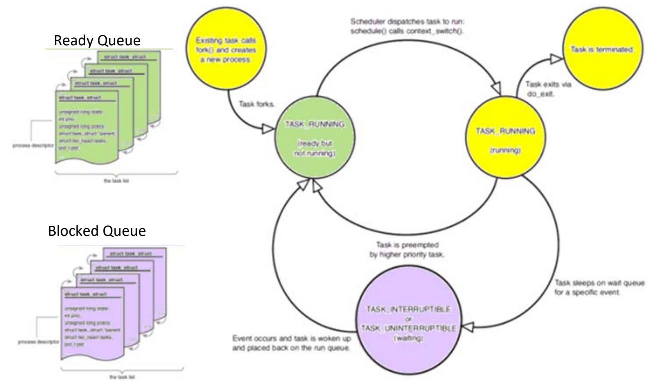
Page Hits



- If valid bit is set
  - MMU uses the physical address in PTE (page table element) to construct the physical address of the word

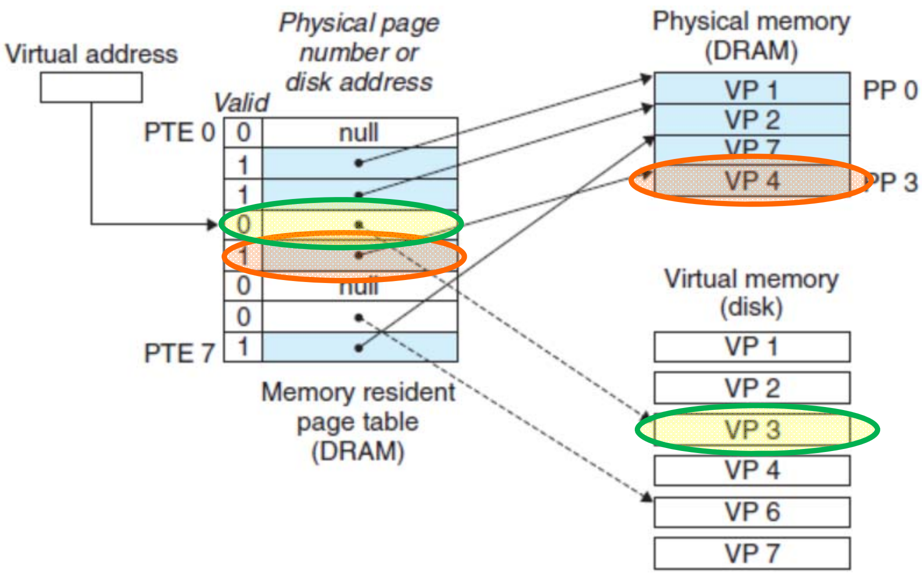
Page Faults

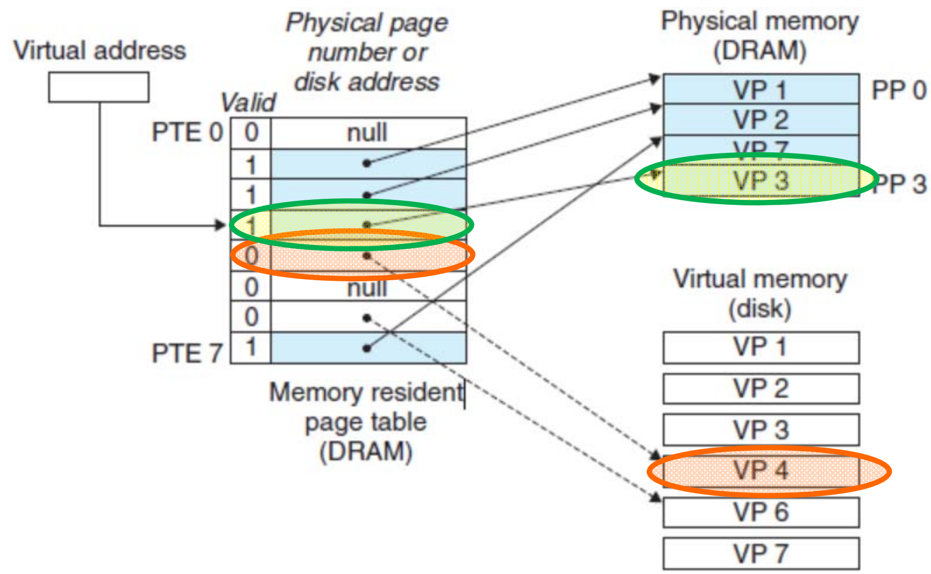
- If valid bit is not set
  - MMU triggers a page fault exception



- Page fault exception handler in Kernel
  - Selects a victim page (if necessary)
    - If the victim page has been modified, copy it to disk
    - Clear the valid bit and update the victim PTE address
  - Copy the virtual page to the physical page and update the page
  - Restart the faulting instruction

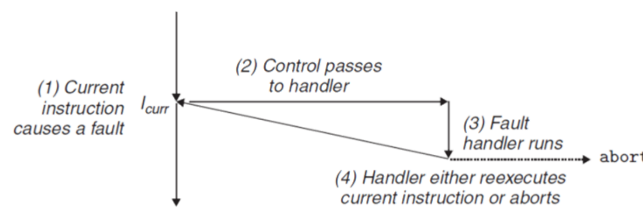
Page Faults





Before
Trying to read from VP3
VP4 is a selected victim page

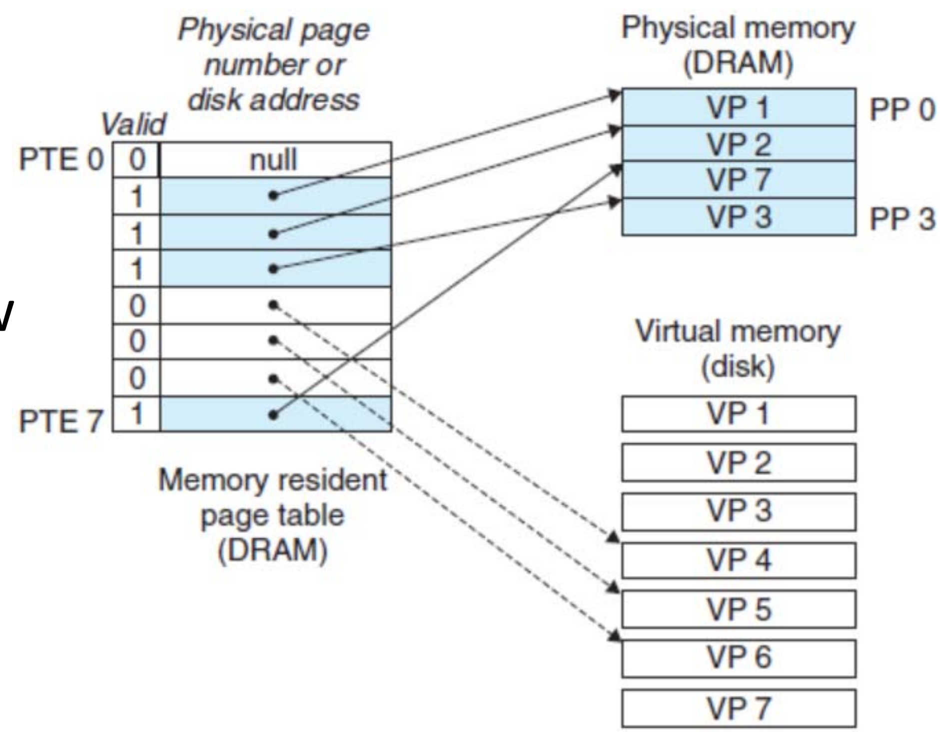
After
Page-hit when reading from VP3 again



SUNY Korea

Allocating a Page

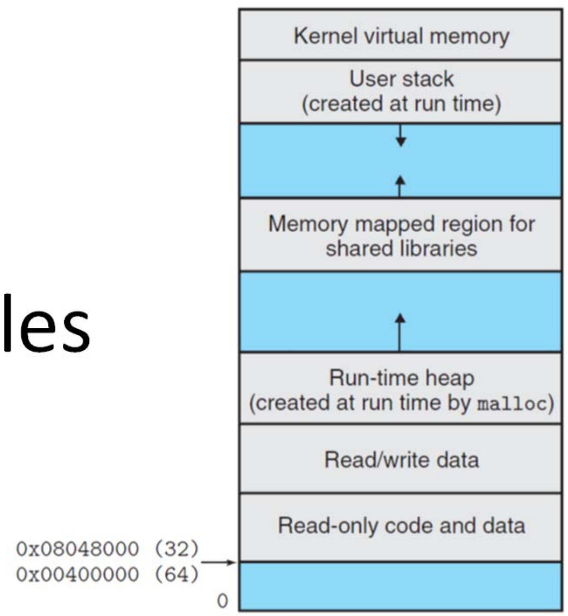
- Creating a new page of virtual memory
  - e.g. malloc will create a new page of virtual memory
  - Create room on disk
  - Update PTE to point to the newly created page on disk



VP5 is created on disk and PTE5 is pointing to the location

VM as a Tool for Memory Management:
Simplify LINKING

- Separate address space for each process
  - Each process has the same basic format for its memory image regardless of where code/data reside in memory

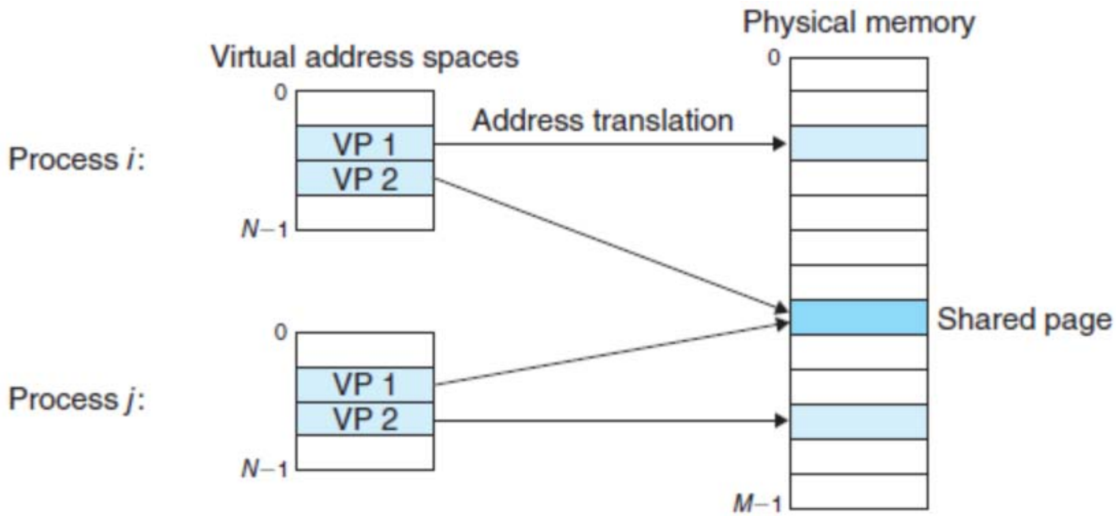


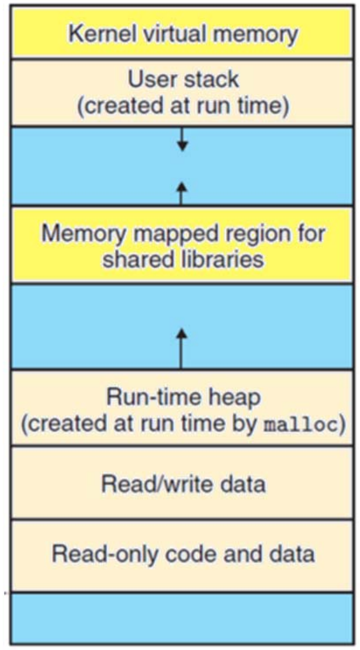
- The uniformity makes it easy to create fully linked executables

VM as a Tool for Memory Management: Simplify LOADING

- Linux: to load .text, .rodata and .data sections of an object file into memory
  - Allocate pages for code and data
  - Mark them as invalid
  - Point their PTE to the appropriate locations in the object file
  - The virtual memory system will automatically copy them into physical memory

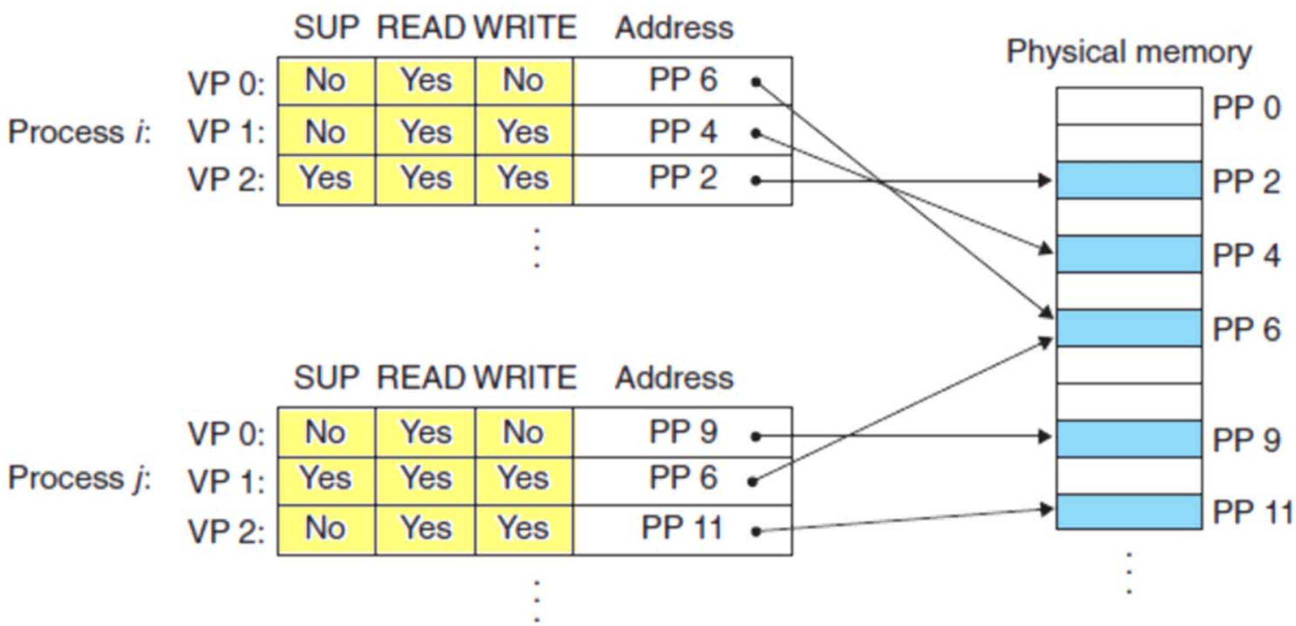
VM as a Tool for Memory Management: Simplify SHARING





- Rather than having separate codes for kernel, standard C libraries, *etc* in each process, they can be shared

VM as a Tool for Memory Protection



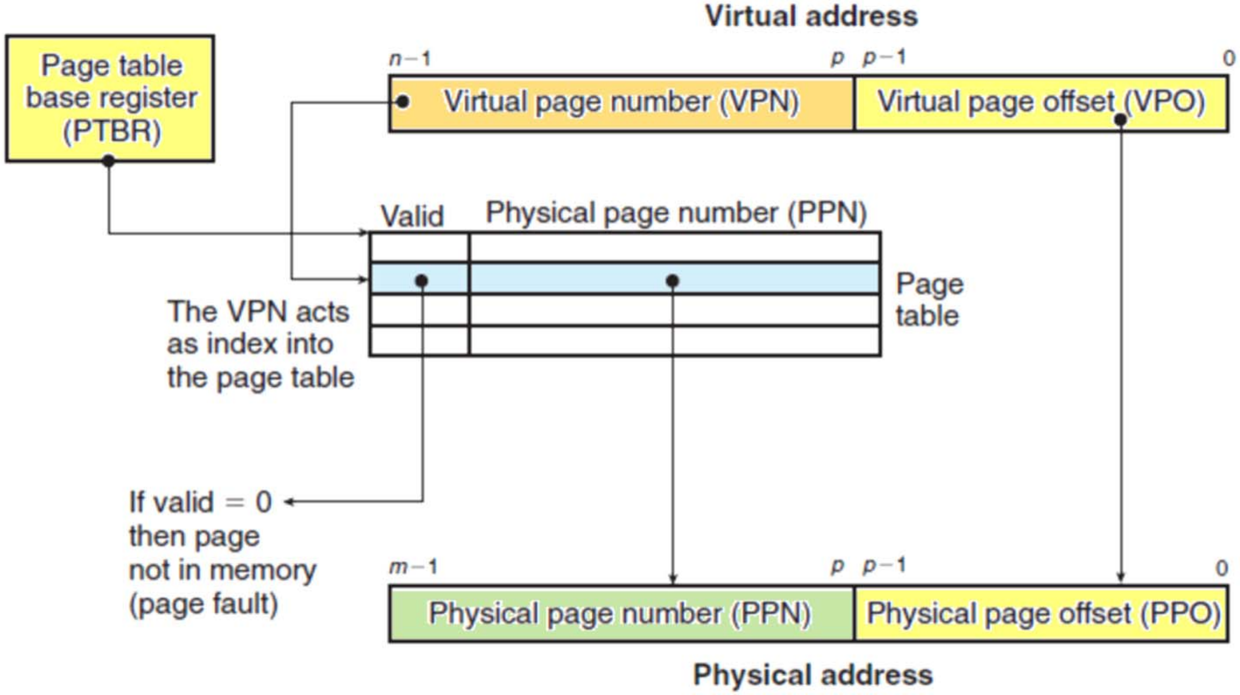
- Control the access to the contents of a virtual page by adding some additional permission bits
  - SUP: can be accessed in the kernel mode
  - READ, WRITE: read/write control

Address Translation

- Address Translation
  - Mapping between an N-element virtual address space (VAS) and an M-element physical address space (PAS)

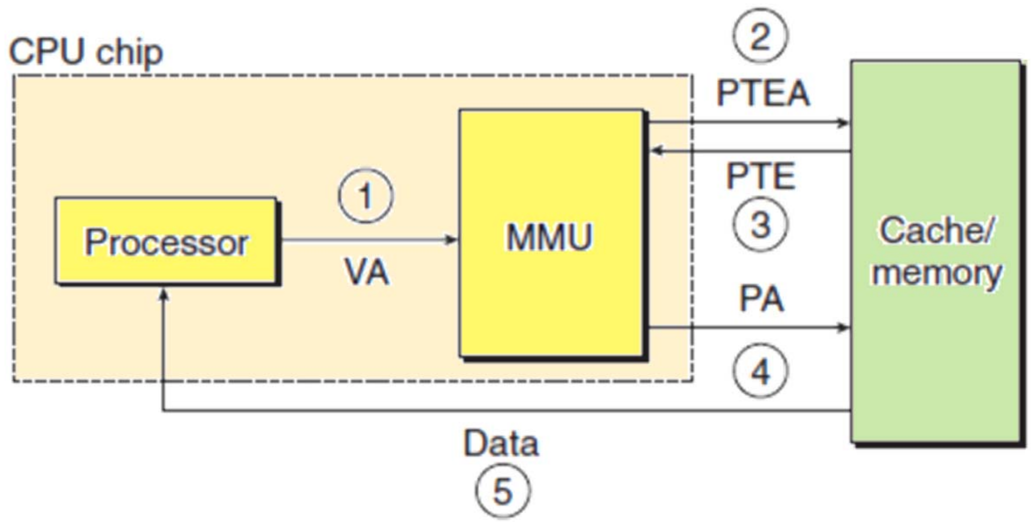
  - $MAP: VAS \rightarrow PAS \cup \varnothing$
  - $MAP(A) = A'$ if data at A are present at A' in PAS;
    $\varnothing$ otherwise

Address Translation



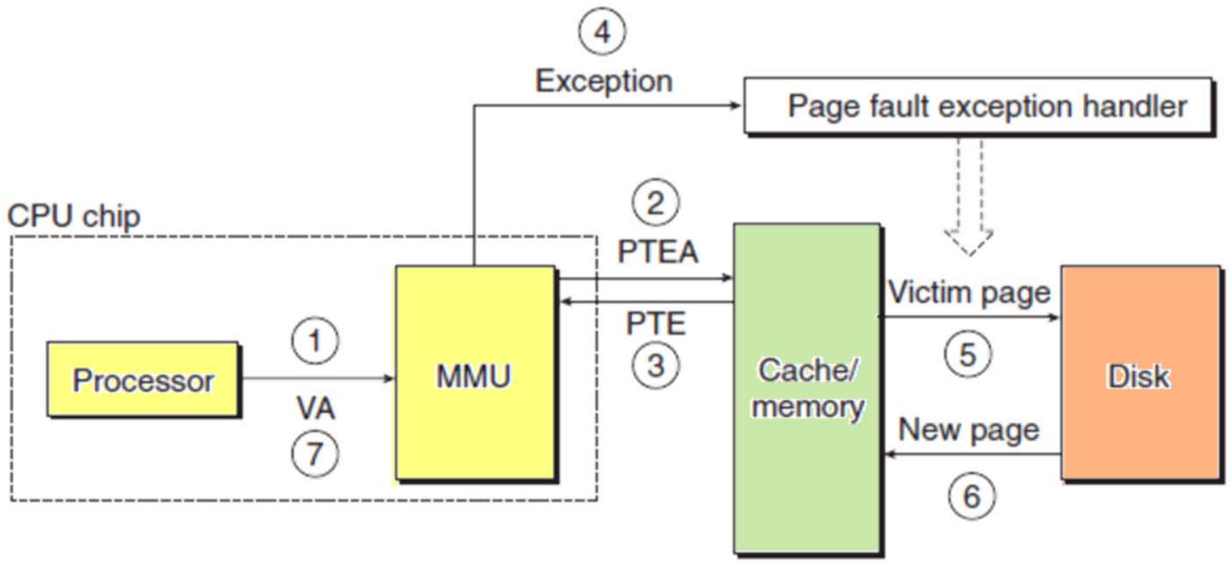
- Page Table Base Register (PTBR) points to the current page table
- Virtual address is divided into Virtual Page Number (VPN) and Virtual Page Offset (VPO)
- Corresponding physical address is the concatenation of Physical Page Number (PPN) and VPO

Address Translation: page-hit



1. Processor generates a virtual address and sends it to MMU
2. MMU generates PTEA and requests from cache/main memory
3. Cache/main memory returns PTE to MMU
4. MMU constructs physical address and sends it to cache/main memory
5. Cache/main memory returns the requested data to the processor

Address Translation: page-miss



- Step 1 .. 3 are the same as page-hit case

1. Valid bit in PTE is 0 → MMU triggers an exception → kernel's page fault exception handler runs
2. Fault handler identifies a victim page and pages it out to disk if dirty
3. Fault handler pages in the new page and updates the PTE in memory
4. Fault handler returns to the original process and restart the instruction that caused the page fault

Course Evaluation

Your feedback is important!

- Please submit your Course Evaluation at
  https://stonybrook.campuslabs.com/eval-home/

Exercise

- The program in the following slides is a simulated cache and a simulated paging mechanism
- Download cache.c from the course webpage and implement the functionalities marked as TODO comments

```c
#define ADRSBITS    (20)                    //1MB
#define VPNBITS     (10)                    //# of bits in Virtual Page Number
#define VPOBITS     (ADRSBITS - VPNBITS)    //# of bits in Virtual Page Offset
#define PTEBYTES    ((1 + VPNBITS + 7)/8)   //# of bytes in PTE. 1: valid bit,
                                            //  7, 8: round up
#define VPNVALID    (1 << (1+VPNBITS))      //valid bit of VPN
#define PAGESIZE    (1 << VPOBITS)          //size of each page
#define PAGECOUNT   (1 << VPNBITS)          //# of pages
#define MAXPROCESS  (4)                     //max number of process

typedef unsigned short word_t;
typedef unsigned int   dword_t;

typedef struct {
    char valid;
    int tag;
    int bufSize;
    char *buf;
} cache_line_t;

typedef struct {
    int nl;  //# of lines (E)
    cache_line_t *lines;
} cache_set_t;
```

```c
typedef struct {
    int m, s, b, t;
    int ns;  //# of sets 2^s (S)

    //masks for set index, tag, and block offset
    //e.g. if s==3, mask_set - (1<<s)-1 = 7
    int mask_set, mask_tag, mask_off;
    cache_set_t *sets;
} cache_t;

typedef struct {
    int pid;
    word_t PTBR;    //page table base register
} process_t;

typedef struct {
    char *mem;      //physical memory
    char *disk;     //disk
    cache_t Lx;     //cache: physical address -> contents
    cache_t TLB;    //translation lookahead buffer
    word_t PTBR;    //page table base register
    dword_t *rmap;  //page frame to PTE (physical adrs) map
    word_t mask_vpn, mask_vpo;
} mem_sys_t;
```

```c
/////////////////////////////////////////////////////////////
// cache_t related functions
//
void init_cache_line(cache_line_t *cl, int bufSize) {
    cl->valid = 0;
    cl->tag = 0;
    cl->bufSize = bufSize;
    cl->buf = (char*)malloc(bufSize);
}
void init_cache_set(cache_set_t *cs, int numLines, int bufSize) {
    int i;
    cs->nl = numLines;
    cs->lines = (cache_line_t*)malloc(sizeof(cache_line_t)*numLines);
    for(i = 0; i < numLines; i++)
        init_cache_line(cs->lines + i, bufSize);
}
void init_cache(cache_t* c, int m, int s, int b, int E) {
    int i;
    memset(c, 0, sizeof(cache_t));
    //TODO: delete above line with memset,
    // initialize c->m, c->s, c->b, c->t, c->ns,
    // c->mask_set, c->mask_tag, c->mask_off

    c->sets = (cache_set_t*)malloc(sizeof(cache_set_t)*c->ns);
    for(i = 0; i < c->ns; i++) {
        //TODO: initialize c->sets+i by init_cache_set
        init_cache_set(c->sets + i, E, 1 << b);
    }
}
```

```c
// return the cache line that contains the adrs
//       or NULL if none does
cache_line_t *find_cache_line(cache_t *c, dword_t adrs) {
    dword_t set = 0;//TODO: find set index from adrs
    dword_t tag = 0;//TODO: find tag from adrs
    cache_set_t *cs = &c->sets[set];
    int i;
    for(i = 0; i < cs->nl; i++)
        if( cs->lines[i].valid &&
            (c->t == 0 || cs->lines[i].tag == tag) )
            return &cs->lines[i];
    return NULL;
}

//read the word at adrs if adrs is cached; otherwise
//simply return 0 without trying to load from memory
int try_read_from_cache(cache_t *c, dword_t adrs, word_t *w);

//write w at adrs if adrs is cached; otherwise
//simply return 0 without trying to load from memory
int try_write_to_cache(cache_t *c, dword_t adrs, word_t w);
```

```c
//return if there is a cacheLine for adrs; otherwise
//pick a cacheLine and load it with the block from memory
void load_cache_line(cache_t *c, dword_t adrs, char *mem) {
    dword_t set = 0; //TODO: find set index from adrs
    dword_t tag = 0; //TODO: find tag from adrs
    dword_t off = adrs & c->mask_off;
    cache_set_t *cs = &c->sets[set];

    //TODO: 1. try to find the line that contains adrs
    if(cl != NULL)
        return;

    //TODO: 2. try to find an invalid line
    int i;
    for(i = 0; i < cs->nl; i++) {
        //TODO: find an invalid line
    }
```

```c
    //3. evict a line
    if(cl == NULL) {
        cl = &cs->lines[rand() % cs->nl]; //randomly pick a line
        if(mem != NULL) {
            //write back to mem
            //TODO: construct padrs using cl->tag, c->s, c->b, and set
            //TODO: copy from c->buf to mem + padrs using memcpy
        }
    }

    if (mem != NULL) {
        //load the line from mem
        //TODO: copy from mem + adrs - off to cl->buf using memcpy
    }

    //TODO: update cl->valid and cl->tag

    return;
}

//write w to a loaded cacheLine
void write_to_cache(cache_t *c, dword_t adrs, word_t w, char *mem) {
    load_cache_line(c, adrs, mem);
    ON_FALSE_EXIT(try_write_to_cache(c, adrs, w), "cache is not loaded");
}
```

```c
//copy the cache contents to memory
//invalidate all cache lines if invalidate is true
void flush_cache(cache_t *c, char *mem, int invalidate) {
    int i, j;
    for(i = 0; i < c->ns; i++) {
        //flush set i
        int nl = c->sets[i].nl;
        for(j = 0; j < nl; j++) {
            //flush line j of set i
            cache_line_t *cl = &c->sets[i].lines[j];
            if(cl->valid && mem != NULL) {
                //TODO: construct padrs using cl->tag, c->s, c->b, and set index
                //TODO: copy from c->buf to mem + padrs using memcpy
            }
            if(invalidate)
                cl->valid = 0;
        }
    }
}
```

```
////////////////////////////////////////////////////////////
// Virtual memory related functions
//

//try to read from Lx cache if padrs is cached; otherwise
//load cacheLine from memory and try to read from Lx again
void read_from_physical_mem(mem_sys_t *ms, dword_t padrs, word_t *w);

//try to write to Lx cache if padrs is cached; otherwise
//load cacheLine from memory and try to write to Lx again
void write_to_physical_mem(mem_sys_t *ms, dword_t padrs, word_t w) {
    //TODO: implement write back, write allocate policy by
    // 1. try to write to cache
    // 2. on cache miss, load a cacheLine with padrs
    // 3. try to write to cache again
}

//read pte from TLB if vnp*PTEBYTES is cached; otherwise,
//read pte from physical memory and cache it at TLB
void read_PTE(mem_sys_t *ms, word_t vpn, word_t *pte) {
    //TODO: get pte by
    // 1. try to read pte from TLB
    //    use try_read_from_cache(&ms->TLB, vpn*PTEBYTES, pte)
    // 2. on cache miss, get pte from the physical mem and update TLB
    //    to read from physical mem use read_from_physical_mem.
    //    Note, vpn*PTEBYTES is not a padrs
    //    to update TLB: write_to_cache(&ms->TLB, vpn*PTEBYTES, *pte, NULL)
}
```

```c
//read the page at virtual address vadrs from Disk
void read_page_from_disk(mem_sys_t *ms, dword_t vadrs) {
    word_t vpn = (vadrs >> VPOBITS) & ms->mask_vpn;
    word_t vpo = vadrs & ms->mask_vpo;
    word_t pte, pfn; //pte, page frame number
    dword_t padrs, doff;
    read_PTE(ms, vpn, &pte);
    if(!(pte & VPNVALID)) {
        word_t fn; //page frame number
        //1. find an empty page frame
        for(fn = 2*MAXPROCESS; fn < PAGECOUNT; fn++) //0 .. 2*MAXPROCESS-1: page tables
            if(ms->rmap[fn] == (dword_t)-1)
                break;
        //2. if not found, pick a page and swap it out
        if(fn == PAGECOUNT) {
            fn = (rand() % (PAGECOUNT-2*MAXPROCESS)) + 2*MAXPROCESS;
            doff = (ms->rmap[fn]/PTEBYTES) << VPOBITS;
            //TODO: copy the picked page frame to disk by
            // 1. flush Lx cache to memory before copying the page to disk
            // 2. compute padrs as the beginning of the page frame address for fn
            // 3. copy the page at ms->mem+padrs to disk at ms->disk+doff using memcpy
            // 4. invalidate the pte pointing to fn by writing 0 to it:
            //    use write_to_physical_mem and rmap
            // 5. invalidate TLB
        }
```

```
        //update pte, TLB, rmap
        pte = fn | VPNVALID;
        write_to_physical_mem(ms, ms->PTBR + vpn*PTEBYTES, pte); //update page table
        write_to_cache(&ms->TLB, vpn*PTEBYTES, pte, NULL);       //update TLB
        ms->rmap[fn] = ms->PTBR + vpn*PTEBYTES;                  //update rmap
    }
    pfn = pte & ms->mask_vpn;
    doff = ((ms->PTBR + vpn*PTEBYTES)/PTEBYTES) << VPOBITS;
    //TODO: copy the page for vadrs from disk to memory by
    // 1. invalidate Lx cache because the contents of the page will be replaced.
    //    for simplicity, invalidate the entire cache rather than find/invalidate
    //    only the lines for the replaced page
    // 2. compute padrs as the beginning of the page frame address for pfn
    // 3. copy the page in disk at ms->disk+doff to mem at ms->mem+padrs using memcpy
}
```

```c
//write the page at virtual address vadrs to Disk
void write_page_to_disk(mem_sys_t *ms, dword_t vadrs) {
    word_t vpn = (vadrs >> VPOBITS) & ms->mask_vpn;
    word_t vpo = vadrs & ms->mask_vpo;
    word_t pte, pfn; //pte, page frame number
    dword_t padrs, doff;
    read_PTE(ms, vpn, &pte);
    ON_FALSE_EXIT(pte & VPNVALID, "writing an invalid page");
    pfn = pte & ms->mask_vpn;
    doff = ((ms->PTBR + vpn*PTEBYTES)/PTEBYTES) << VPOBITS;
    //TODO: copy the page frame from memory to disk by
    // 1. flush Lx cache to memory before copying the memory to disk
    // 2. compute padrs as the beginning of the page frame address
    // 3. copy the page in memory at ms->mem+padrs to disk at ms->disk+ms->doff
    //    using memcpy
}

//read w from a virtual address vadrs
void read_from_virtual_mem(mem_sys_t *ms, dword_t vadrs, word_t *w);

//write w to a virtual address vadrs
void write_to_virtual_mem(mem_sys_t *ms, dword_t vadrs, word_t w);
```

```
/////////////////////////////////////////////////////////////
// Memory system and process
//
void init_mem_sys(mem_sys_t *ms) {
    ms->mem  = (char*)malloc(1 << ADRSBITS);
    ms->disk = (char*)malloc(MAXPROCESS * (1 << ADRSBITS));
    ms->rmap = (dword_t*)malloc(PAGECOUNT * sizeof(dword_t));
    memset(ms->rmap, 0xff, PAGECOUNT * sizeof(dword_t));
    ms->mask_vpn = (1 << VPNBITS)-1;
    ms->mask_vpo = (1 << VPOBITS)-1;
    ms->PTBR = 0;
    init_cache(&ms->Lx, ADRSBITS, 6/*s*/, 8/*b*/, 8/*E*/);
    init_cache(&ms->TLB, VPNBITS, 0/*s*/, 1/*b: word_t size*/, 16/*E*/);
}

void init_process(process_t *proc, int pid, mem_sys_t *ms) {
    proc->pid = pid;
    proc->PTBR = pid * 2*PAGESIZE;            //2 pages per page table
    ms->rmap[pid*2]   = VPNVALID | proc->PTBR; //mark the frames as being used
    ms->rmap[pid*2+1] = VPNVALID | (proc->PTBR+PAGESIZE);
    memset(ms->mem + proc->PTBR, 0, 2*PAGESIZE);
}

void switch_to_process(process_t *proc, mem_sys_t *ms) {
    flush_cache(&ms->TLB, NULL, 1/*invalidate*/);
    ms->PTBR = proc->PTBR;
}
```

```c
int main() {
    ON_FALSE_EXIT(2*PAGESIZE >= sizeof(word_t)*(1 << VPNBITS),
                  "a page cannot contain a page table");
    ON_FALSE_EXIT(PTEBYTES == sizeof(word_t), "PTEBYTES != sizeof(word_t)");

    srand(10);
    unit_test_cache();
    unit_test_virtual_memory();
    printf("SUCCESS!!!\n");
}
```

```
Expected result:
$ ./a.out
-- Testing cache initialization...
-- Testing writeToCache...
-- Testing writeToCache 2nd line...
-- Testing writeToCache eviction...
-- Testing flushCache...
-- Testing initMemSystem...
-- Testing switchToProcess...
-- Testing writeToVirtualMem...
-- Testing writeToVirtualMem by proc[1]...
-- Testing readFromVirtualMem by proc[0]...
-- Testing page swapping...
SUCCESS!!!
```