Lecture 10: Virtual Memory

Professor G. Sandoval

Some slides adapted by G. Sandoval for CS3224, from Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.
Also some Slides by Brendan Dolan-Gavitt

- Virtual Memory
- XV6 Implementation
- Page Replacement Algorithms

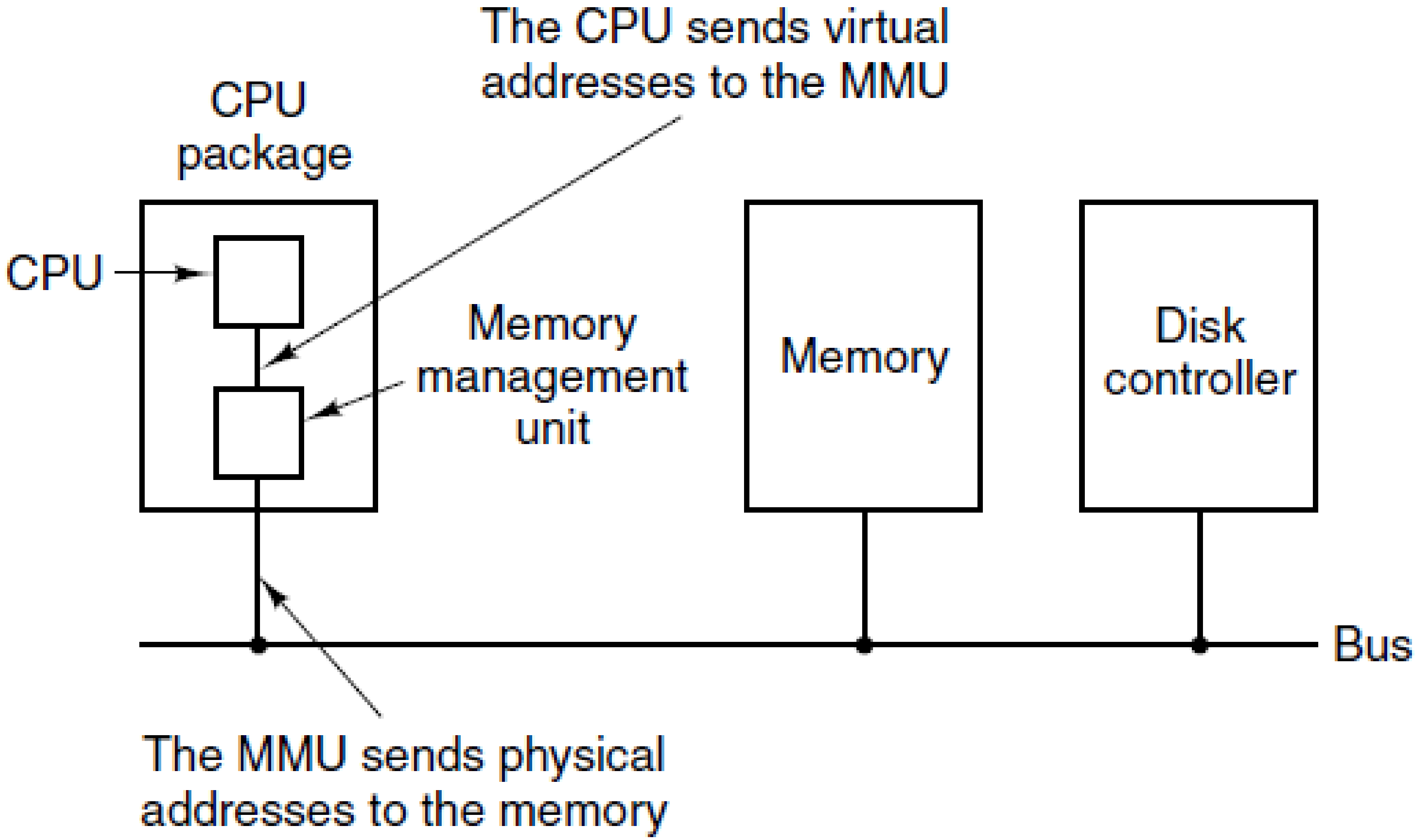
Virtual Memory

- Recall from last time – segmentation is no longer used to separate processes' memory from one another
- Instead, *virtual addressing* is used
- Each memory access no longer refers directly to physical memory, but instead is **mapped** to some actual physical address

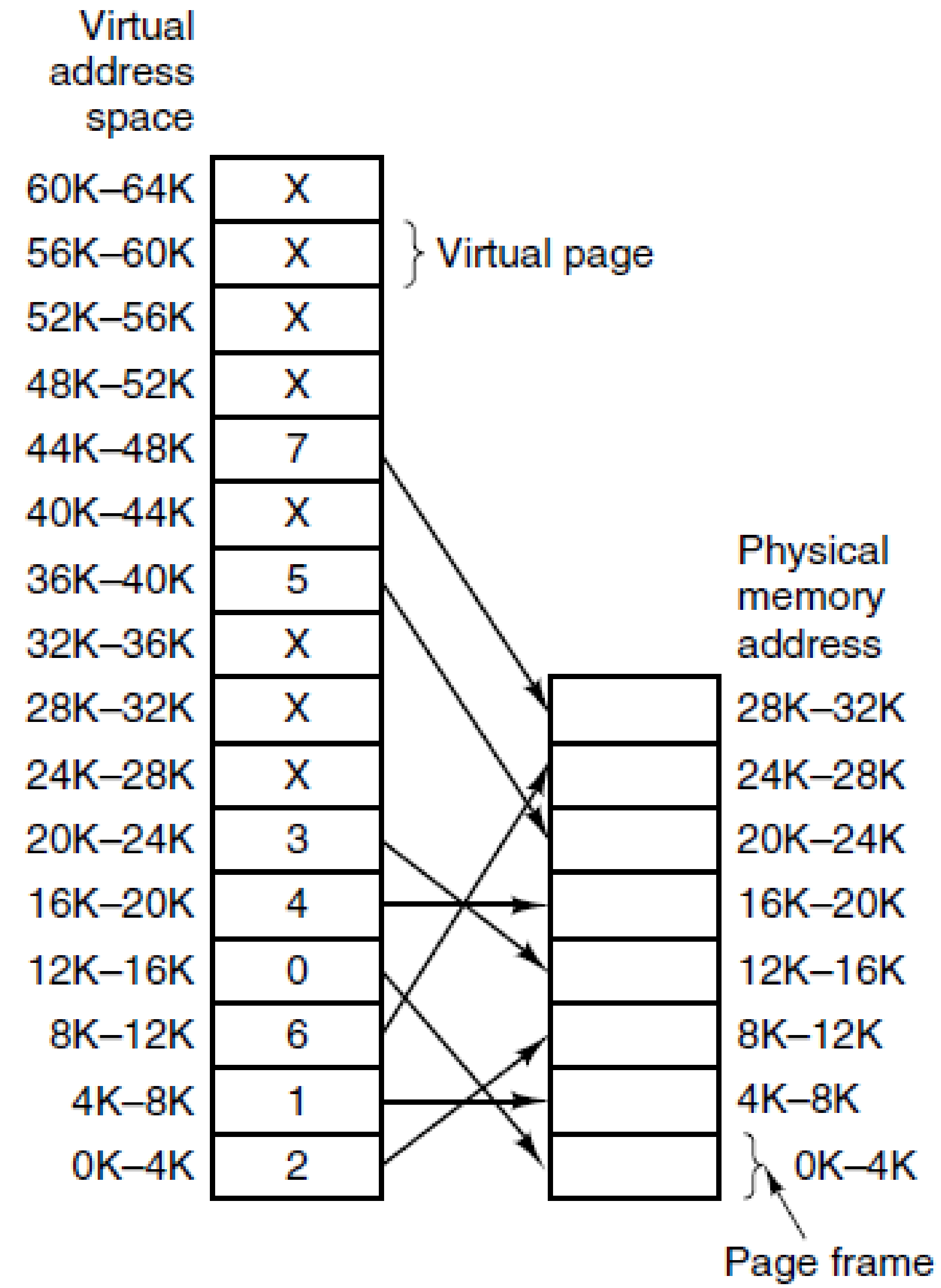
Paging

- Each program has it’s “own” address space, which is broken up into chunks called pages.
- Virtual memory is broken up into fixed-sized units (commonly, 0x1000 bytes (4096 decimal)) called *pages*
- Page sizes can vary though:
  - 32-bit x86 supports 4KB and 4MB pages
  - 64-bit x86 supports 4KB, 2MB, and 1GB pages
- The underlying physical pages of memory are called *page frames*

Paging


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Paging



The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287

Tanenbaum & Bo,Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

Page Faults

- What happens if we try to access a page that is not mapped?
- The MMU notices, and we raise a CPU exception called a *page fault*
- Control is passed to the OS (via the usual interrupt/exception handling mechanism) to decide what to do
  - Kill the process (*segmentation fault*)
  - Find some physical page to map to it

Programs Bigger than Memory

- Note that this gives us a way to have programs that don't all fit into memory at once
- We can just map in the parts of the program we're using right now
- If we hit code or data that isn't mapped, we can *swap* some other page to disk and update the mappings

Types of Page Fault

- **Minor page fault** – can be serviced by just creating the right mapping
- **Major page fault** – must load in a page from disk to service
- **Segmentation fault** – invalid address accessed; can't service so we usually just kill the program

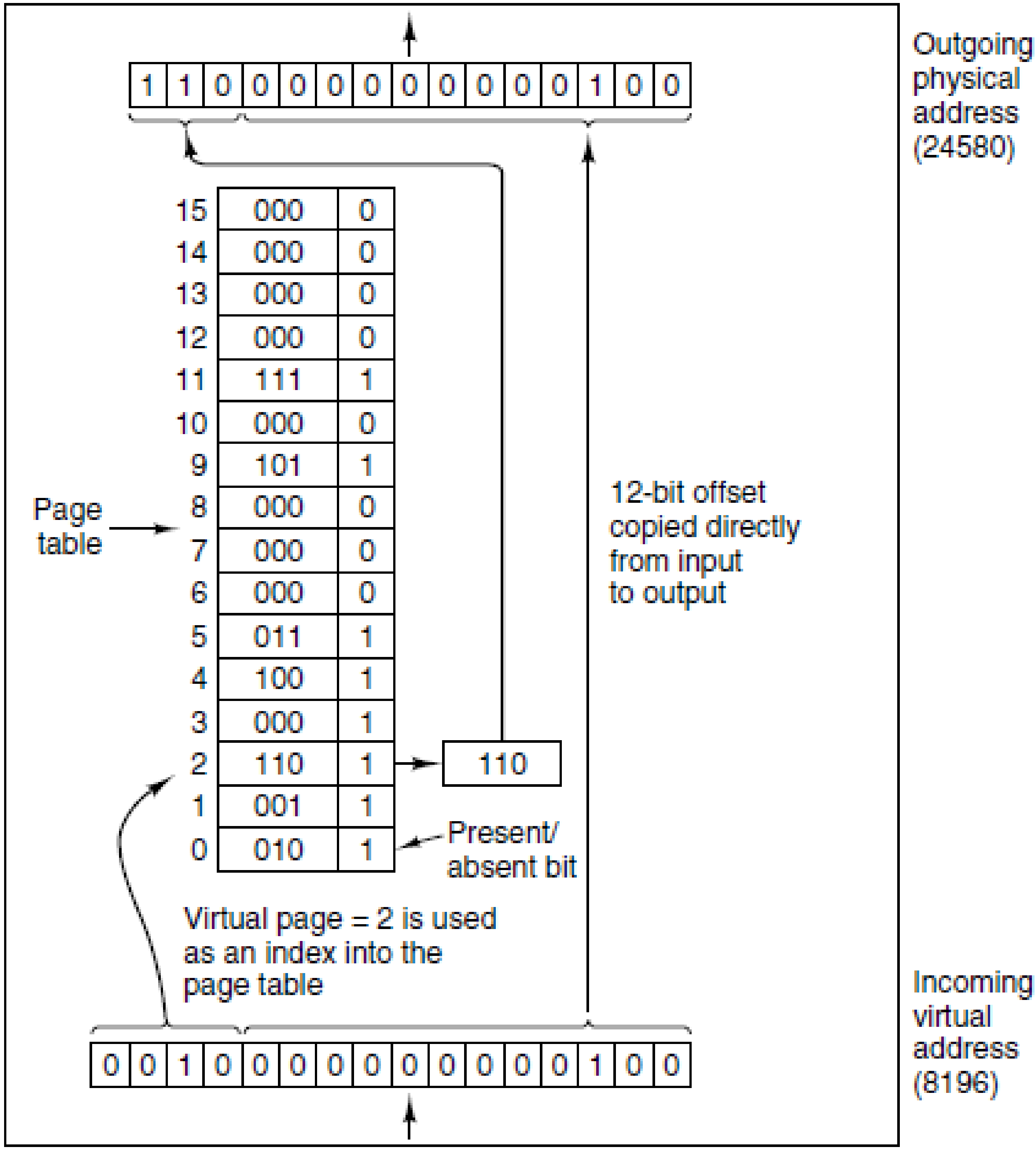
Page Tables

- The MMU has to maintain information about the virtual->physical mapping

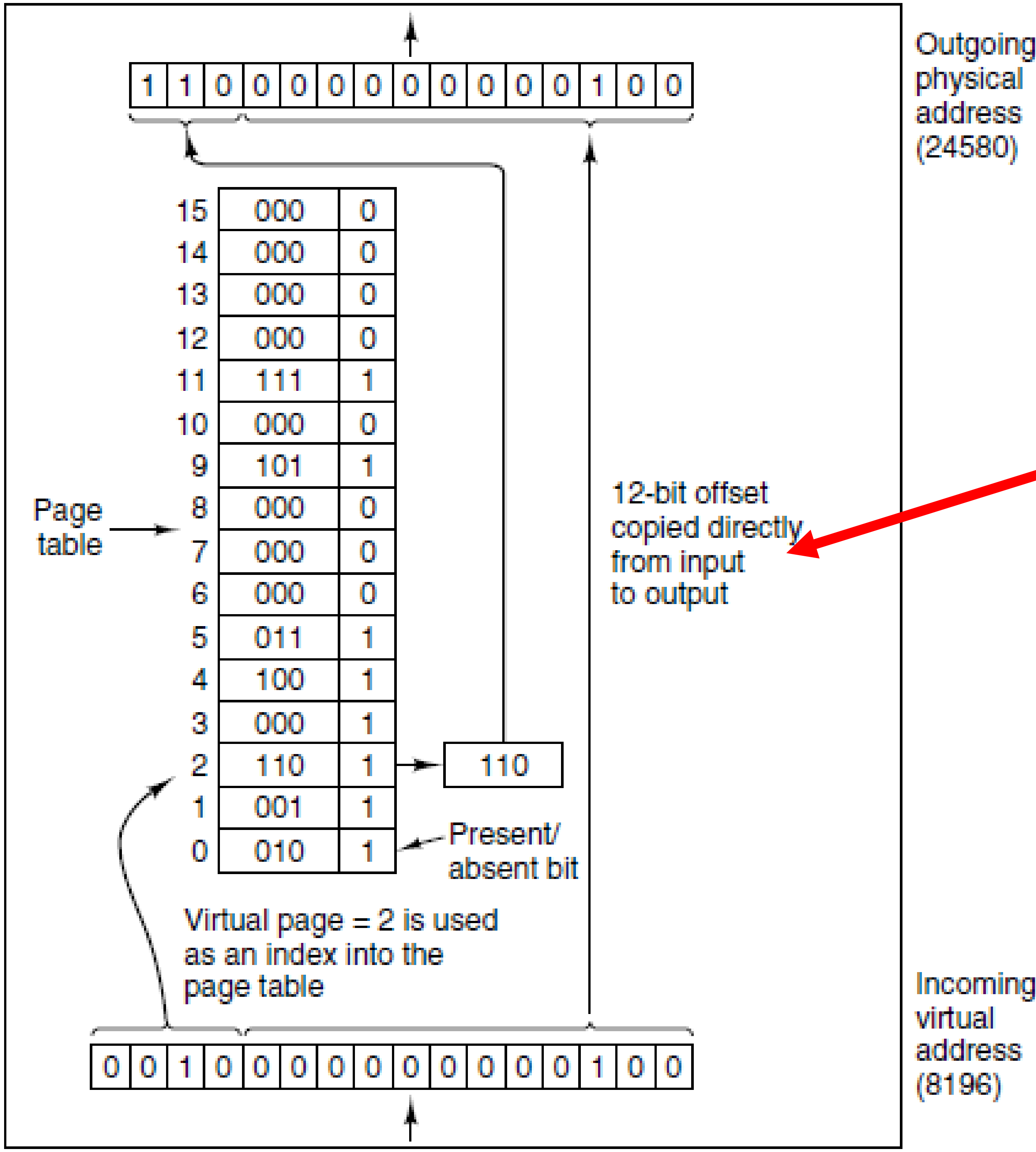
- In the simplest case, this could be a simple array that stores the physical page number for each virtual page number

- The virtual address would then be split into two parts: an index into the mapping table, and then the offset within the page

The internal operation of the MMU with 16 4-KB pages.

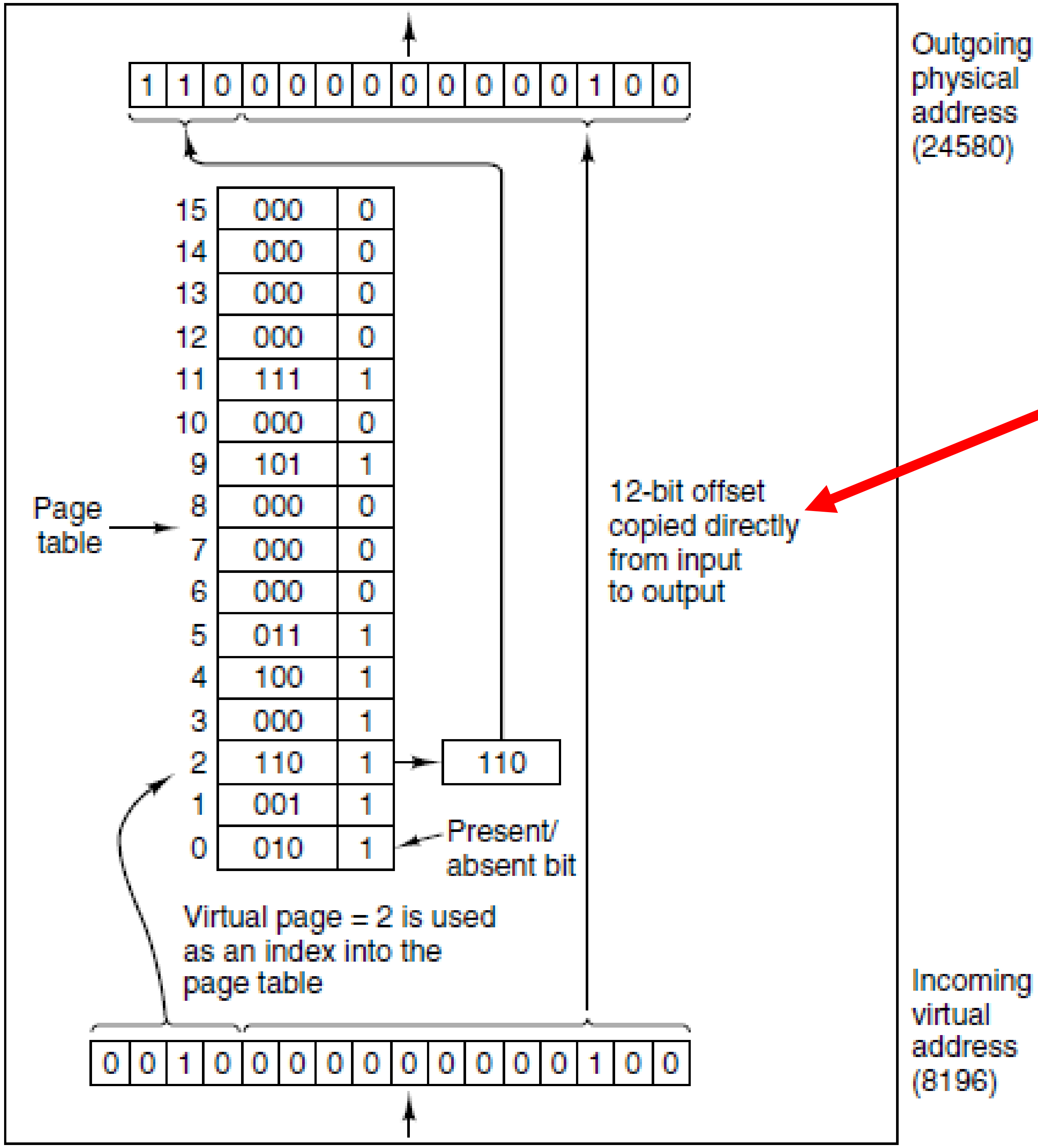


Tanenbaum & Bo,Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.



Outgoing physical address (24580)
1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Page table
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
110
Present/ absent bit
12-bit offset copied directly from input to output
Virtual page = 2 is used as an index into the page table
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Incoming virtual address (8196)

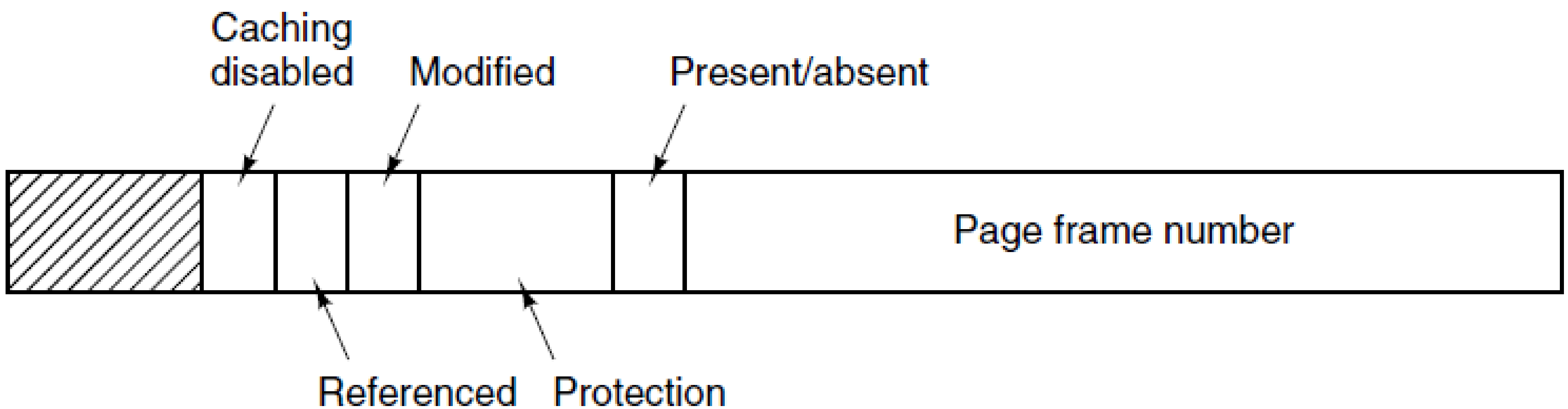
Why 12 bits?



Why 12 bits?

$2^{12} = 4096$

Structure of a Page Table Entry (PTE)



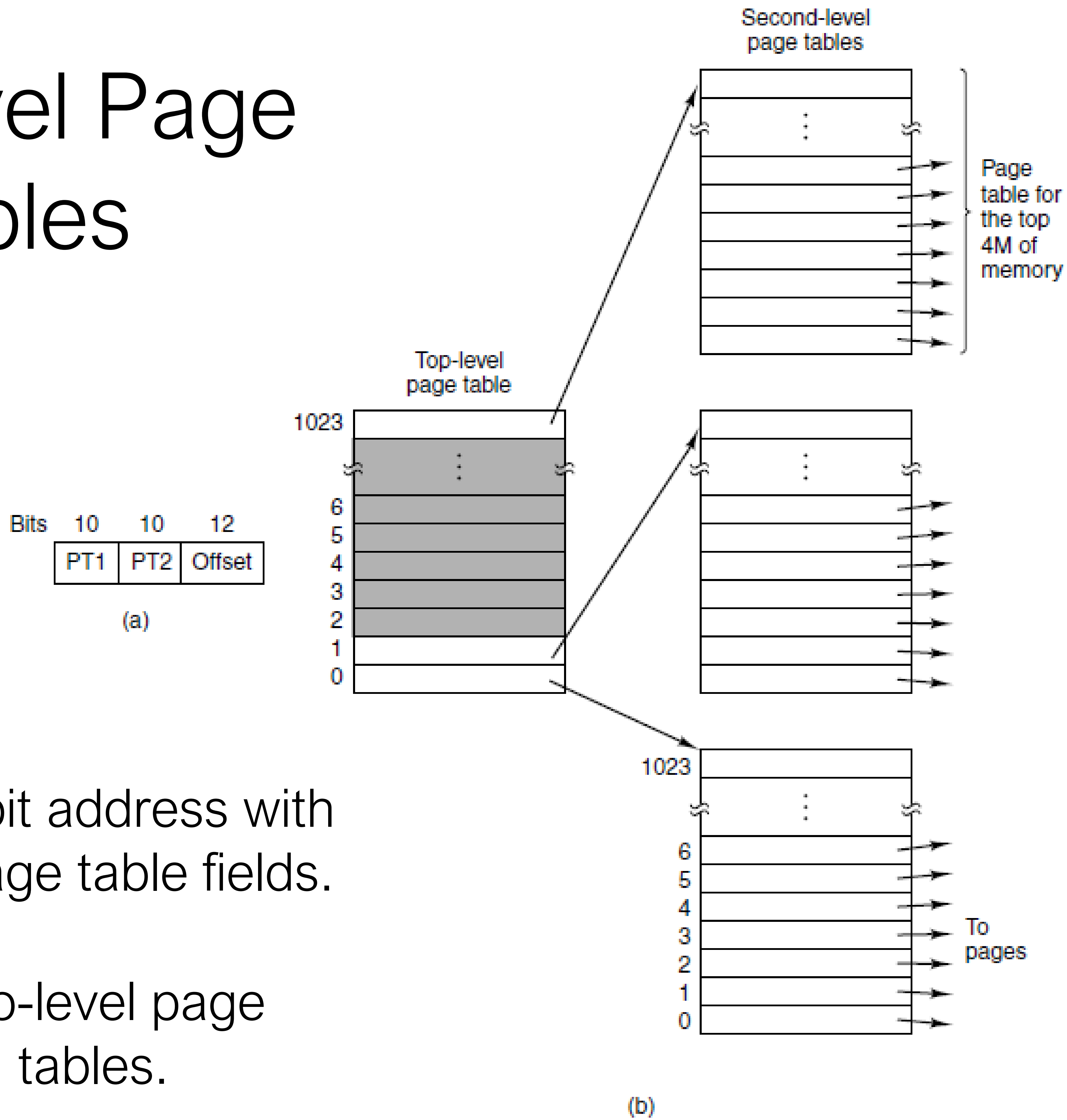
- Modified – has this page been written to?
  - If so, we will need to write to disk before evicting
- Referenced – has anyone used this page?
- Caching disabled – used if physical page is used for device I/O

Tanenbaum & Bo,Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

Real Page Tables

- In reality, this is very inefficient if the virtual address space is expected to be sparse (not many mapped pages)
- Instead, *multi-level page tables* are used
    - The virtual address now has multiple indexes
    - This allows us to only allocate tables for portions of the space that are used
- Remember: the page tables themselves are stored in memory!

Multilevel Page Tables



(a) A 32-bit address with two page table fields.

(b) Two-level page tables.

Tanenbaum & Bo,Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

Protection

- Because the OS can give processes different virtual address spaces, we have already solved the problem of *isolation*
- But we may want to protect processes from themselves in some cases:
  - Detect programmer errors before they do damage
  - Prevent attacks that exploit software vulnerabilities

Protection

- Simplest protection is to mark pages as read-only or read/write
  - Now, if someone attempts to modify read-only code or data, a page fault will occur
- Some processors (in x86-land, starting with the AMD64 in 2003) have a bit to prevent code from being executed on a certain page
  - This has been called variously the NX bit, the XD bit, Data Execution Prevention (DEP)
  - The idea is to prevent buffer overflows from being exploitable – the attacker won't be able to run his own code because it will be in a data region

Translation Lookaside Buffer

- Walking the page table hierarchy each time memory is accessed gets very expensive
  - If we have to do 2 table lookups for every memory access, we've just made memory 3x slower
- Instead, the CPU keeps a *small* set of mappings that it can translate directly without consulting the page tables
- Animation: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/pagingtlb.htm

Translation Lookaside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Tanenbaum & Bo,Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.

TLB Misses

Soft:

- The page referenced is not in the TLB but is in memory.
- 10-20 Machine instructions (2 nanosecs)

Hard:

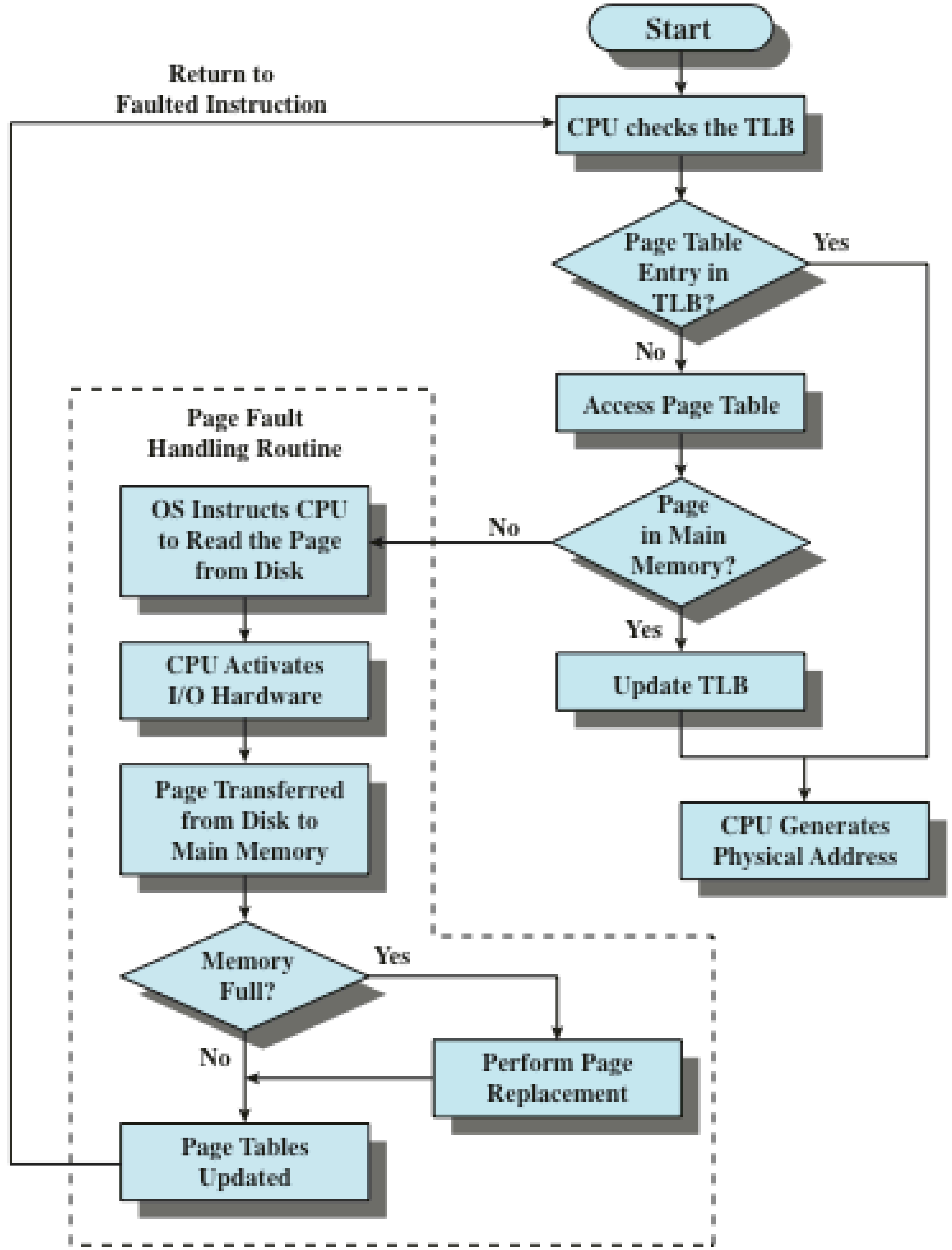
- Page is NOT in memory.
- Takes a few milliseconds (million times slower !!)

TLBs and Context Switching

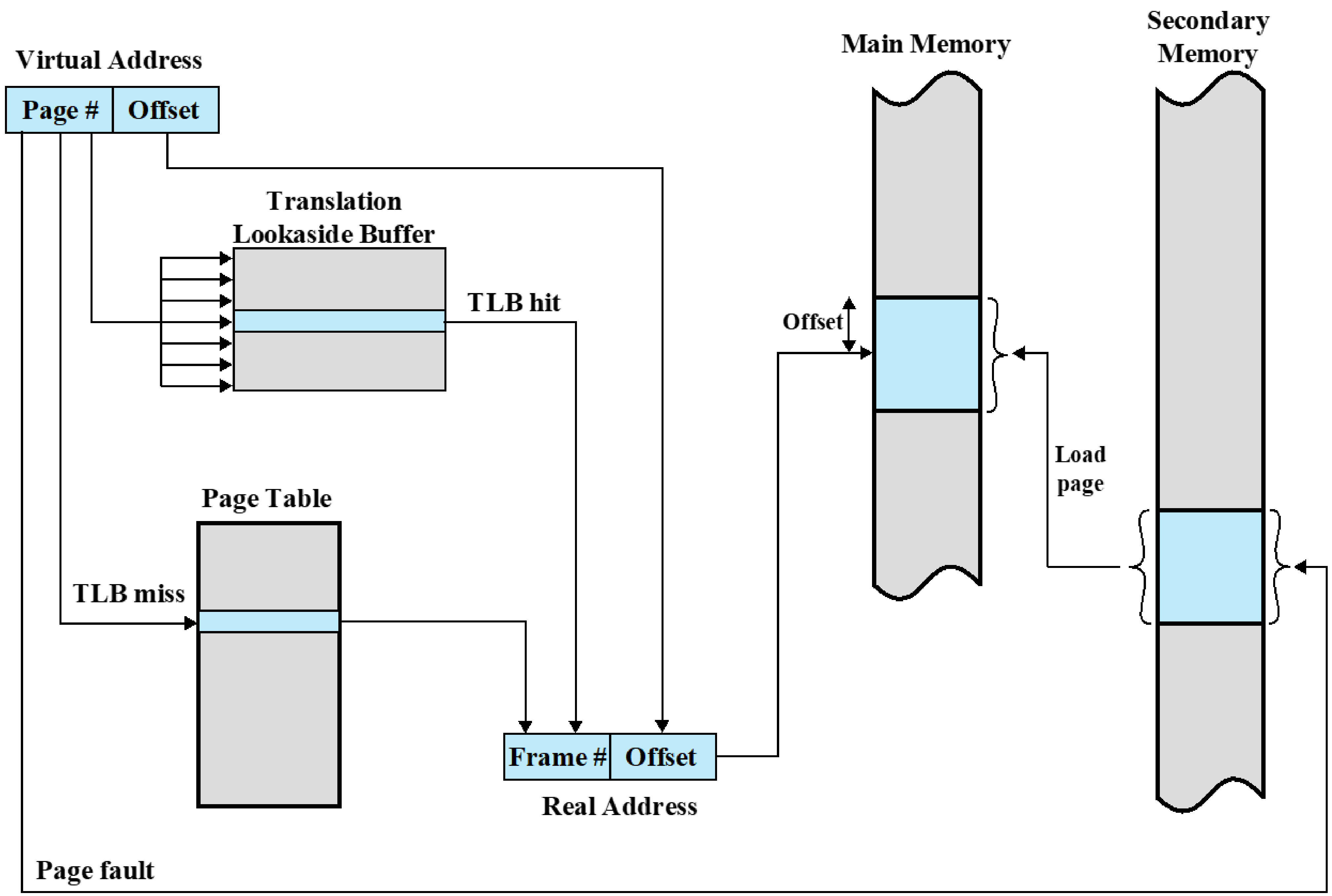
- TLBs map a virtual address to a physical one
- But once we change to a new process, these mappings are no longer valid, and a *TLB flush* occurs
- This makes context switching more expensive – the first few memory accesses a process makes will have to be serviced by walking the page tables

Tagged TLBs

- On some architectures, each TLB entry can be associated with a *tag* that says what address space it belongs to
- Now we don't have to flush the TLB when switching address spaces
- This can help make context switching faster – some TLB entries might still be valid when we switch back to a process


Start
Return to
Faulted Instruction
CPU checks the TLB
Page Table
Entry in
TLB?
Yes
No
Access Page Table
Page Fault
Handling Routine
OS Instructs CPU
to Read the Page
from Disk
No
Page
in Main
Memory?
Yes
Update TLB
CPU Activates
I/O Hardware
Page Transferred
from Disk to
Main Memory
CPU Generates
Physical Address
Memory
Full?
Yes
No
Perform Page
Replacement
Page Tables
Updated

Use of a TLB



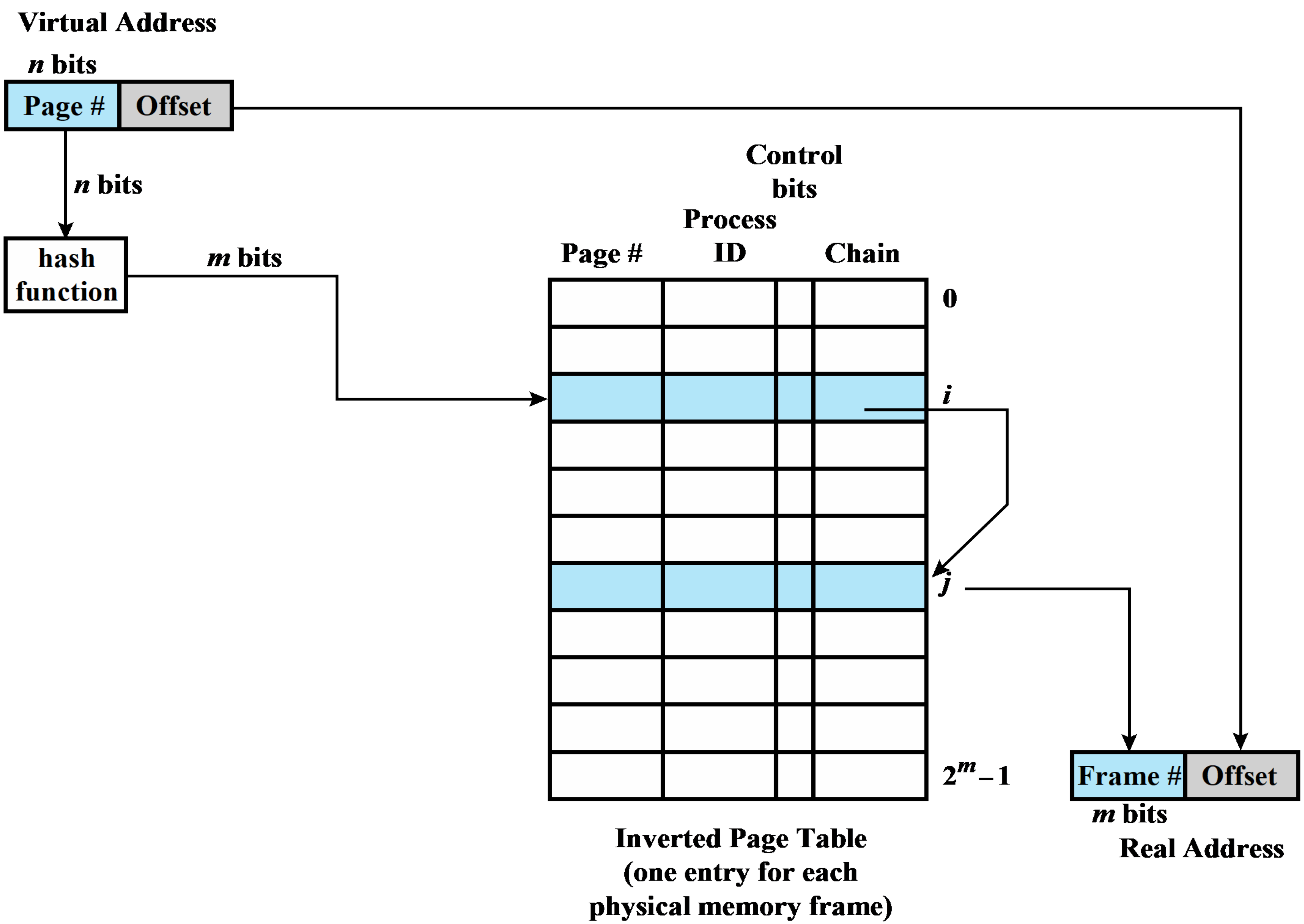
Virtual Address
Page #
Offset
Translation
Lookaside Buffer
TLB hit
Page Table
TLB miss
Frame #
Offset
Real Address
Main Memory
Offset
Secondary
Memory
Load
page
Page fault

Inverted Page Tables

- Instead of using one page table entry per *virtual page*, keep one entry per *physical page frame*
- Benefit: page tables are proportional to size of physical memory, not virtual address space
- Drawback: now we have to search the entire list to look up a mapping
- Used in some architectures: PowerPC, UltraSPARC, Itanium

Hashed Page Tables

- We can reduce the time it takes to lookup a page in an inverted table by using a *hashed page table*
- Basically just a hash table where the keys are virtual addresses and the values are the page directory entries
- Inverted and hashed page tables are common on (non-x86) 64-bit architectures


Virtual Address
n bits
Page #
Offset
n bits
hash function
m bits
Control bits
Process ID
Page #
Chain
0
i
j
$2^m - 1$
Frame #
Offset
m bits
Real Address
Inverted Page Table
(one entry for each
physical memory frame)

Page Table Overhead Calculations

- Using page tables has some overhead
- Just how much?
- Depends on the exact paging scheme used

Page Table Overhead Calculations

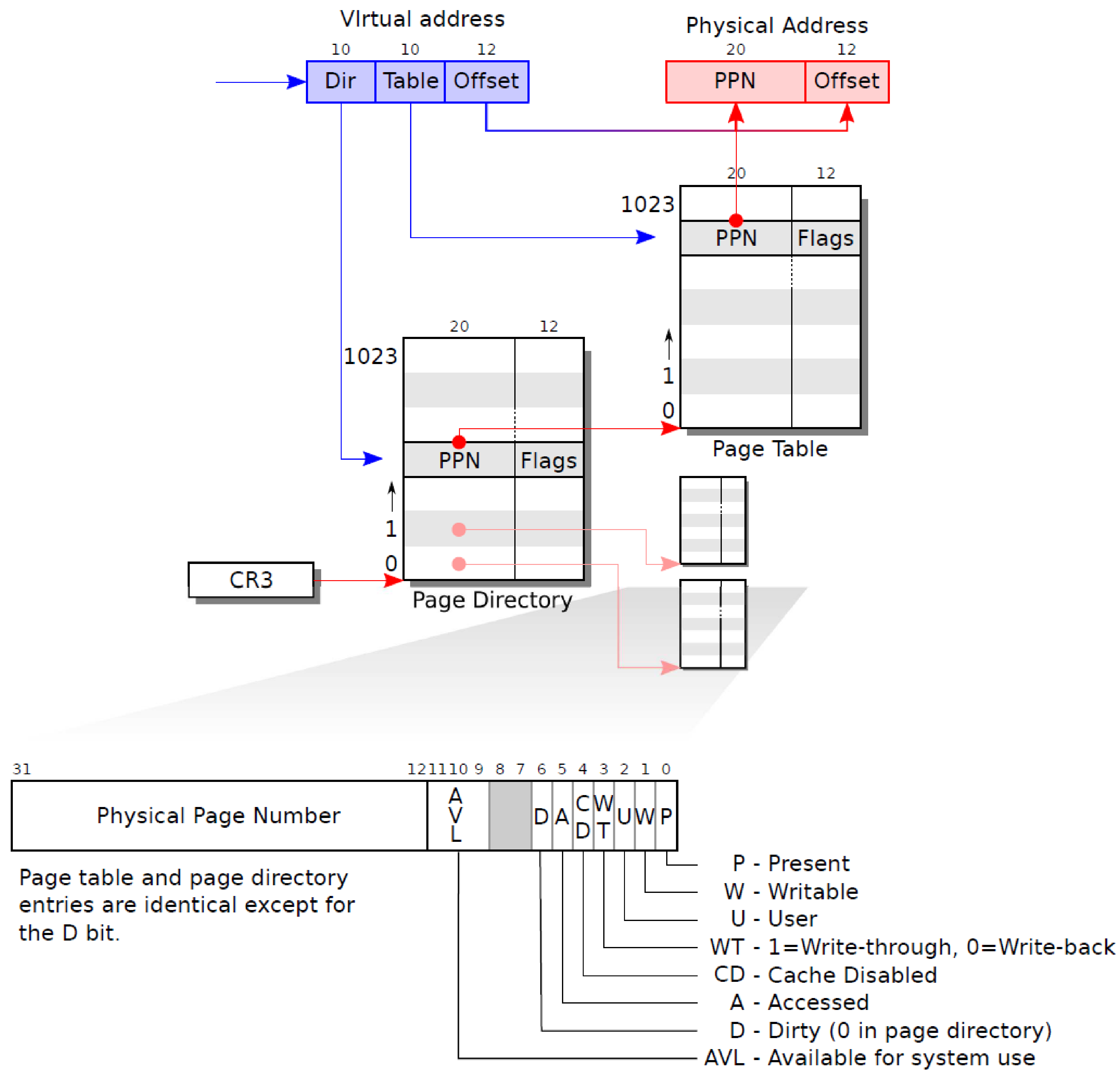
- "Worst" case:
  - single-level page table, 64-bit virtual address space, 4KB pages, 4 byte PTEs
  - 2^{64} / 4096 * 4 bytes = 2^{52} * 2^{2} = 2^{54} bytes = 16 *petabytes* of memory used for pages
    - 2^{50} = Petabyte or 1024 terabytes or million gigabytes
  - And that's just for one process!

Virtual Address Translation in x86

- We will, for now, consider only:
  - 32-bit x86
  - 4KB and 4MB ("super") pages

x86 Paging Basics

- x86 virtual address translation uses a two-level page table:
  - A top-level *page directory* stores pointers to the *page tables*
  - *Page tables* contain the actual *page table entries (PTEs)* referring to physical page frames
- The current mappings in use are determined by the value of the CR3 CPU register, which stores the *physical address* of the page directory



VIrtual address
10 10 12
Dir Table Offset
Physical Address
20 12
PPN Offset
20 12
1023
PPN Flags
1
0
Page Table
20 12
1023
PPN Flags
1
0
CR3
Page Directory
31 12 11 10 9 8 7 6 5 4 3 2 1 0
Physical Page Number
AVL
D A CD WT U W P
Page table and page directory entries are identical except for the D bit.
P - Present
W - Writable
U - User
WT - 1=Write-through, 0=Write-back
CD - Cache Disabled
A - Accessed
D - Dirty (0 in page directory)
AVL - Available for system use

x86 PDEs and PTEs

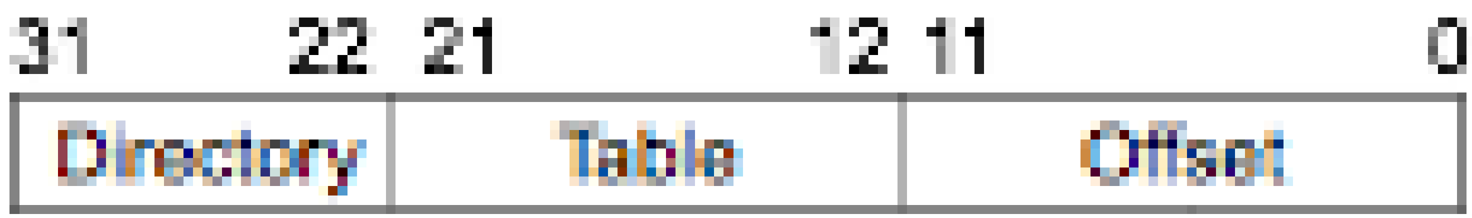
- The entries in a page directory and a page table have an almost identical format
- Each 32-bit
- The basic structure:
  - Physical address of a page table (for PDEs) or memory page (for PTEs)
  - Protection, caching, etc. flags
  - A bit to indicate present/not present

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

Worked Example

- Let's take an arbitrary address I got from running xv6: `0x8010c608`
- In Binary:
  `1000 0000 0001 0000 1100 0110 0000 1000`
- `CR3 = 0x003f f000`

Worked Example



1000000000 0100001100 011000001000

```
Directory index =  1000000000 =  512
Table index =    0100001100  =    268
Offset =    011000001000 =  1544
```

Page Directory

CR3 = 0x003ff000

Directory index = 512

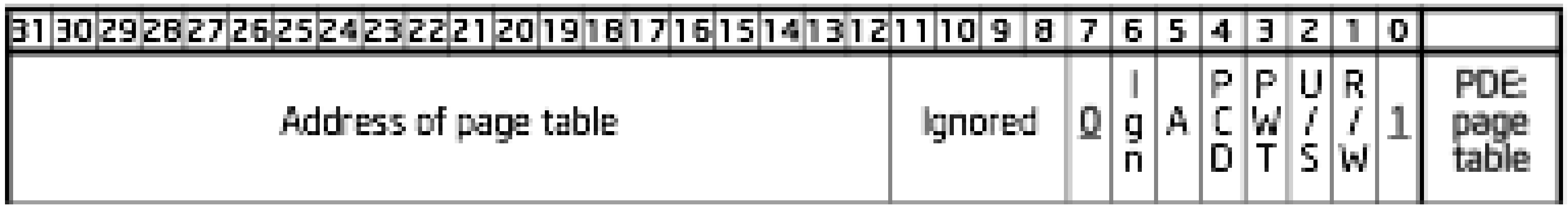
Note:
0x003ff000 + 512 * 4
= 0x003ff800

Address	Entry
0x003ff000:	0x00000000
0x003ff004:	0x00000000
0x003ff008:	0x00000000
0x003ff00c:	0x00000000
0x003ff010:	0x00000000
[...]	
0x003ff800:	0x003fe027
0x003ff804:	0x003fd027
0x003ff808:	0x003fc027
0x003ff80c:	0x003fb027
0x003ff810:	0x003fa027
0x003ff814:	0x003f9027
[...]	

Page Directory Entry

0x003fe027

`0000 0000 0011 1111 1110 0000 0010 0111`



Address of page table: 0x3fe000

(A)	Accessed: Yes
(PCD)	Cache disabled: No
(PWT)	Write through caching: No
(U/S)	User-accessible: Yes
(R/W)	Read/Write: Yes
(P)	Present: Yes

Page Table

Address of page table:
0x3fe000

Table index = 268

Note:
0x003fe000 + 268 * 4
= 0x003fe430

Address	Entry
0x003fe000:	0x00000063
0x003fe004:	0x00001003
0x003fe008:	0x00002003
0x003fe00c:	0x00003003
0x003fe010:	0x00004003
[...]	
0x003fe430:	0x0010c063
0x003fe434:	0x0010d063
0x003fe438:	0x0010e063
0x003fe43c:	0x0010f063
0x003fe440:	0x00110063
0x003fe440:	0x00111063
[...]	

Physical Page

Address of page:
0x10c000

Offset = 1544 =
0x608

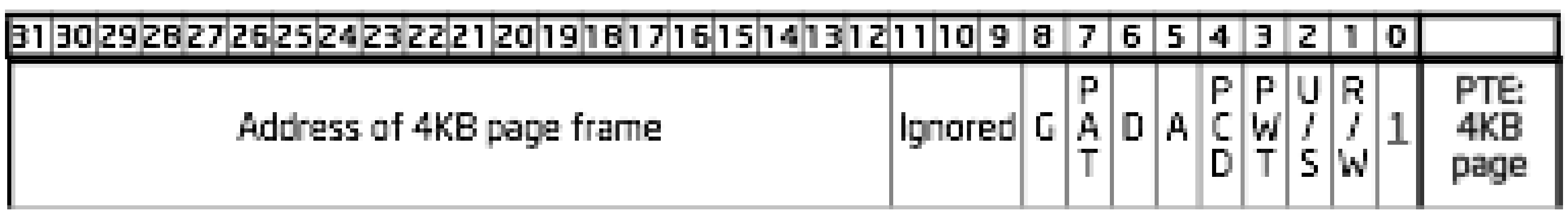
Data: 0x8010c628

```
0000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
                                   [...]
0000600: 8029 1180 8202 0000 28c6 1080 524a 1080  .)......(...RJ..
0000610: 3a87 ff07 0000 0000 38c6 1080 b448 1180  :.......8....H..
0000620: 2824 1180 0100 0000 38c6 1080 8038 1080  ($......8....8..
0000630: 0000 4080 0000 008e 48c6 1080 2138 1080  ..@.....H...!8..
0000640: 0000 0000 54c6 1080 f87b 0000 0000 0000  ....T....{......
0000650: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000660: 0000 0000 d483 1080 0000 0000 0000 0000  ................
0000670: 5c1f 1080 6721 1080 ff23 1080 8924 1080  \...g!...#...$..
0000680: 635e 1080 7555 1080 4d67 1080 5b65 1080  c^..uU..Mg..[e..
0000690: 0000 0000 0200 0000 0100 0000 0100 0000  ................
00006a0: acc8 1080 4c03 1180 0000 0000 e803 0000  ....L...........
00006b0: ad03 0000 c800 0000 1e00 0000 0200 0000  ................
00006c0: 2000 0000 3a00 0000 0000 0000 0000 0000   ...:...........
00006d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Page Table Entry

0x0010c063

`0000 0000 0001 0000 1100 0000 0110 0011`



Address of physical page: 0x10c000

(G) Global: No

(PAT) PAT page: No

(D) Dirty: Yes

(A) Accessed: Yes

(PCD) Page Cache Disable: No

(PWT) Page Write Through: No

(U/S) User-accessible: No

(R/W) Read/Write: Yes

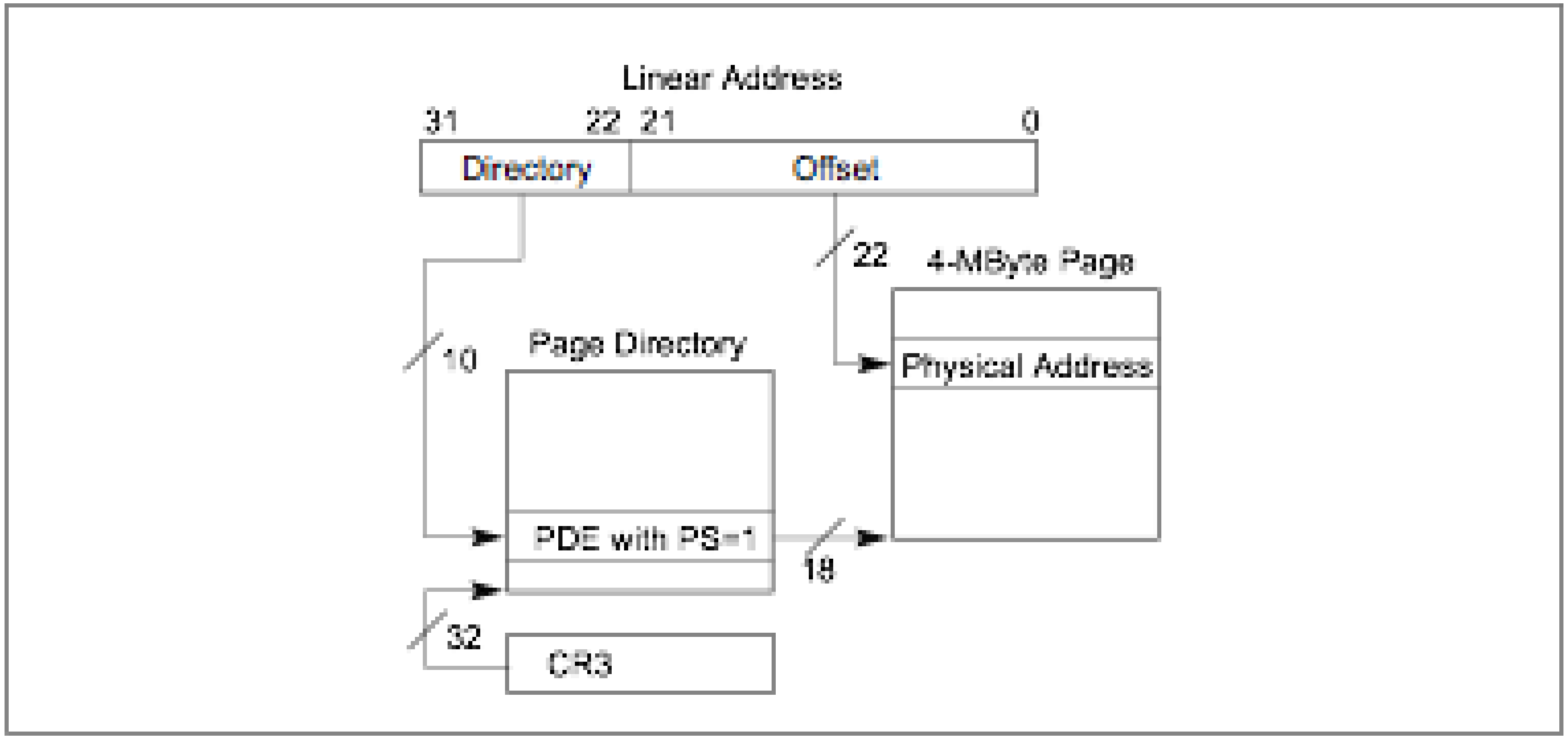
(P) Present: Yes

Super Pages

- If the Page Size Extension (PSE) bit is set in CR4, we can optionally have entries in the page directory point directly to 4MB pages

- This can be beneficial because it reduces paging overhead (only one level of lookup, more data mapped)

Translation with Super Pages



Running the Numbers

- Basics: super pages use 21 bits for offset because 2^{21} = 4MB
- How many entries in a 4KB page table?
  - 1024 entries = 4MB addressed by one page table
  - So each page directory entry addresses 4MB of memory whether it points to a page table or a super page
- How many page directory entries?
  - We want to address 4GB of memory => 4GB/4MB = 1024
- That's why the available sizes are 4KB and 4MB pages

Paging Overhead in x86

- Supposing we have mapped 512MB worth of virtual address space using 4KB pages

- Each page table covers 4MB of memory

- So space required:

  sizeof(1 page directory) +
  (512MB / 4MB) * sizeof(1 page table)
  = 4KB + 128 * 4KB = 4KB + .5 MB = ~0.503 MB

- Virtual Memory
- XV6 Implementation
- Page Replacement Algorithms

Virtual Address Translation in xv6

- xv6 mostly uses 4KB pages (one page table per process, and one page table specifically for the scheduler)
- Early on in boot, it uses entrypgdir, which creates 4MB mappings

entrypgdir

main.c

```
// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.
// Use PTE_PS in page directory entry to enable 4Mbyte pages.
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

entry.S

```
# Set page directory
movl    $(V2P_WO(entrypgdir)), %eax
movl    %eax, %cr3
```

entrypgdir

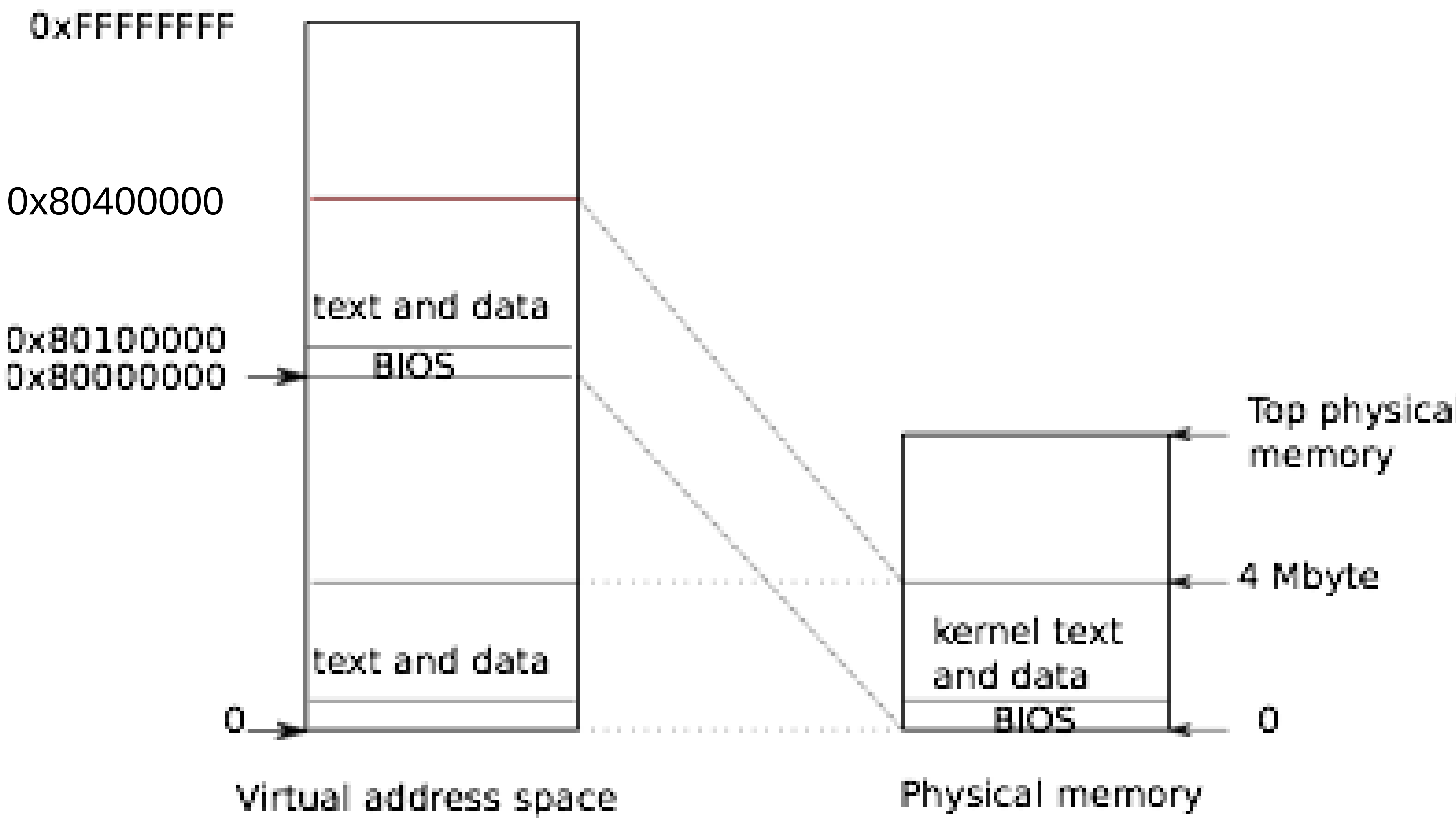
main.c

```
// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.
// Use PTE_PS in page directory entry to enable 4Mbyte pages.
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

0x80000000 >> 22 = 512

entry.S

```
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
```


0xFFFFFFFF
0x80400000
text and data
0x80100000
BIOS
0x80000000
text and data
0
Virtual address space
Top physical memory
4 Mbyte
kernel text and data
BIOS
0
Physical memory

xv6 Process Page Tables

- entrypgdir suffices for early boot, but once boot is done xv6 sets up a more complicated page table
  - I/O space
  - Read-only space for kernel code and r/o data
  - Kernel writeable data & memory

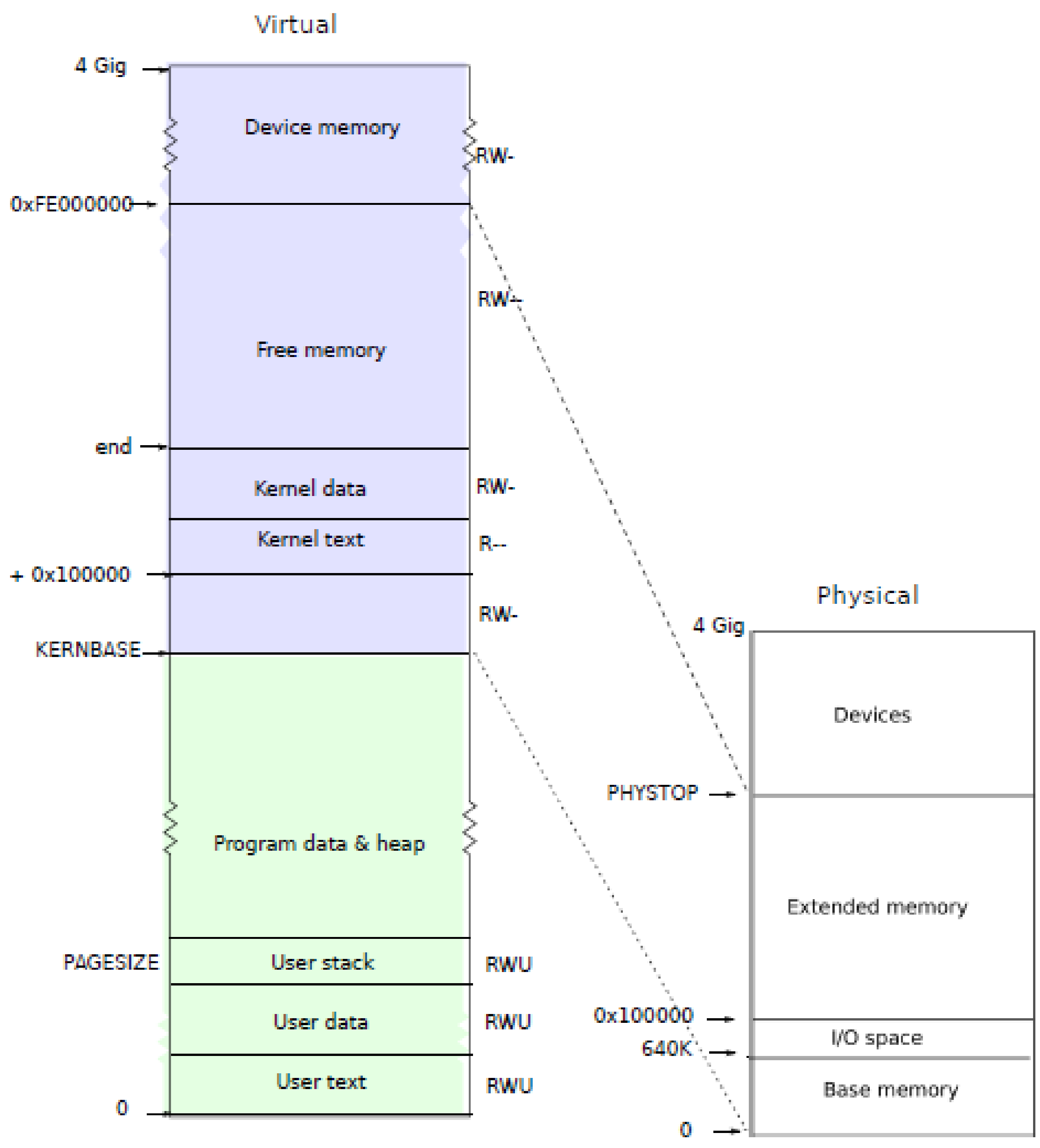
kmap

```
// This table defines the kernel's mappings, which are present in
// every process's page table. (vm.c)
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

Implementing the Map

```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```



Virtual
4 Gig
Device memory
RW-
0xFE000000
Free memory
RW-
end
Kernel data
RW-
Kernel text
R--
+ 0x100000
RW-
KERNBASE
Program data & heap
PAGESIZE
User stack
RWU
User data
RWU
User text
RWU
0
Physical
4 Gig
Devices
PHYSTOP
Extended memory
0x100000
I/O space
640K
Base memory
0

Creating the First Process Address Space

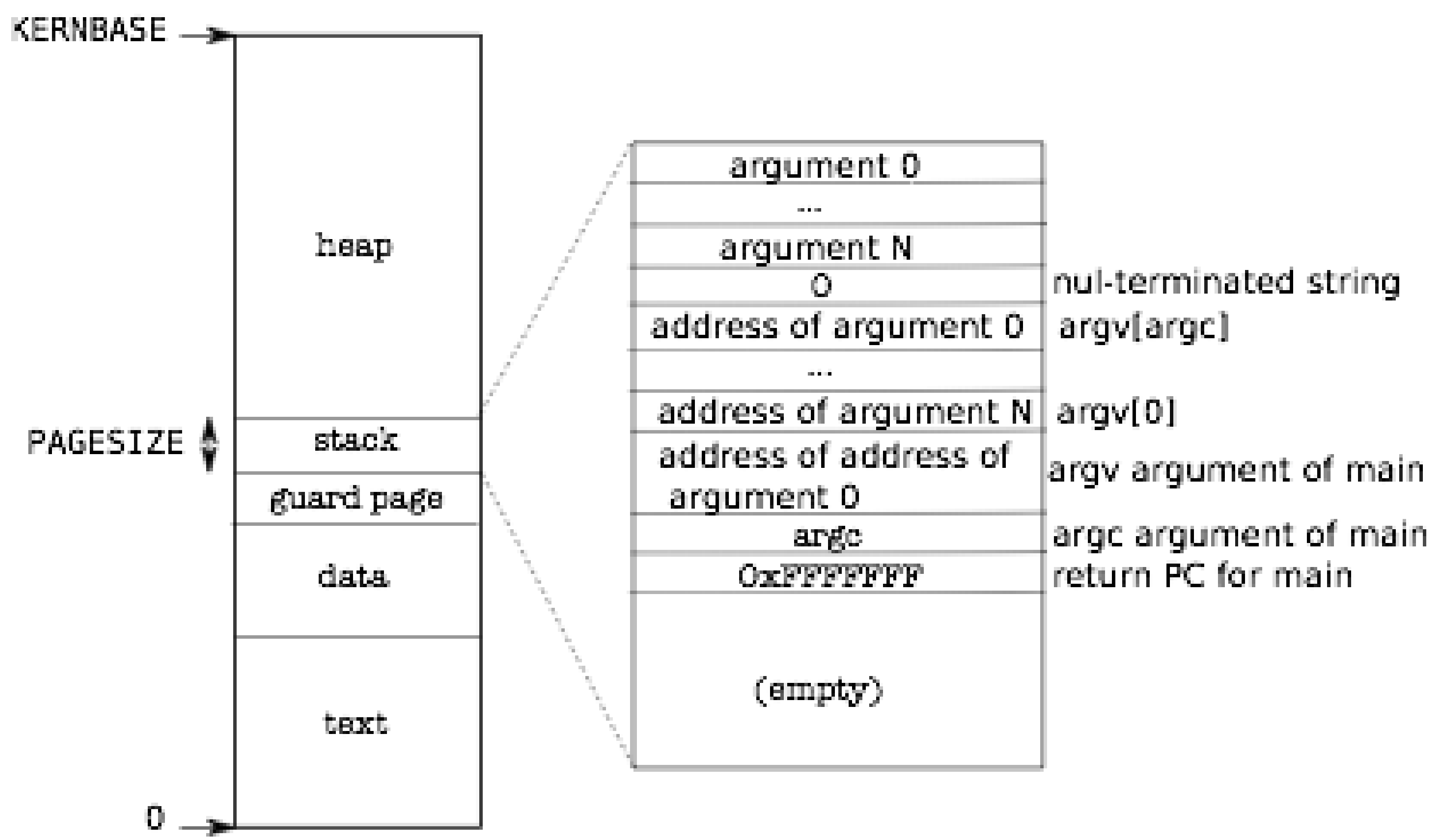
Note: only 4096 bytes given to process to start with

```
// Load the initcode into address 0 of pgdir.
// sz must be less than a page.
// #define PGSIZE          4096
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```

User-Space Layout

Layout set up in exec()



Guard Page

```
int
exec(char *path, char **argv)
{
[...]
  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second
  // as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;
[...]
}
```

Other Features of x86 Paging

- When a page directory / page table entry is *non-present*, its format is unspecified by Intel

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

Non-Present PDEs/PTEs

- Since the CPU/MMU ignore these parts of the PDE/PTE, we can store stuff in them

- Common to use that space to store metadata about the non-present page
  - Example: if the page is available on disk, give info for how to retrieve it

Paging Tricks

- Having virtual address translation around lets us do lots of interesting things aside from basic process isolation

- We'll go through a few applications people have found that make use of the paging hardware

Paging Tricks: Self-reference

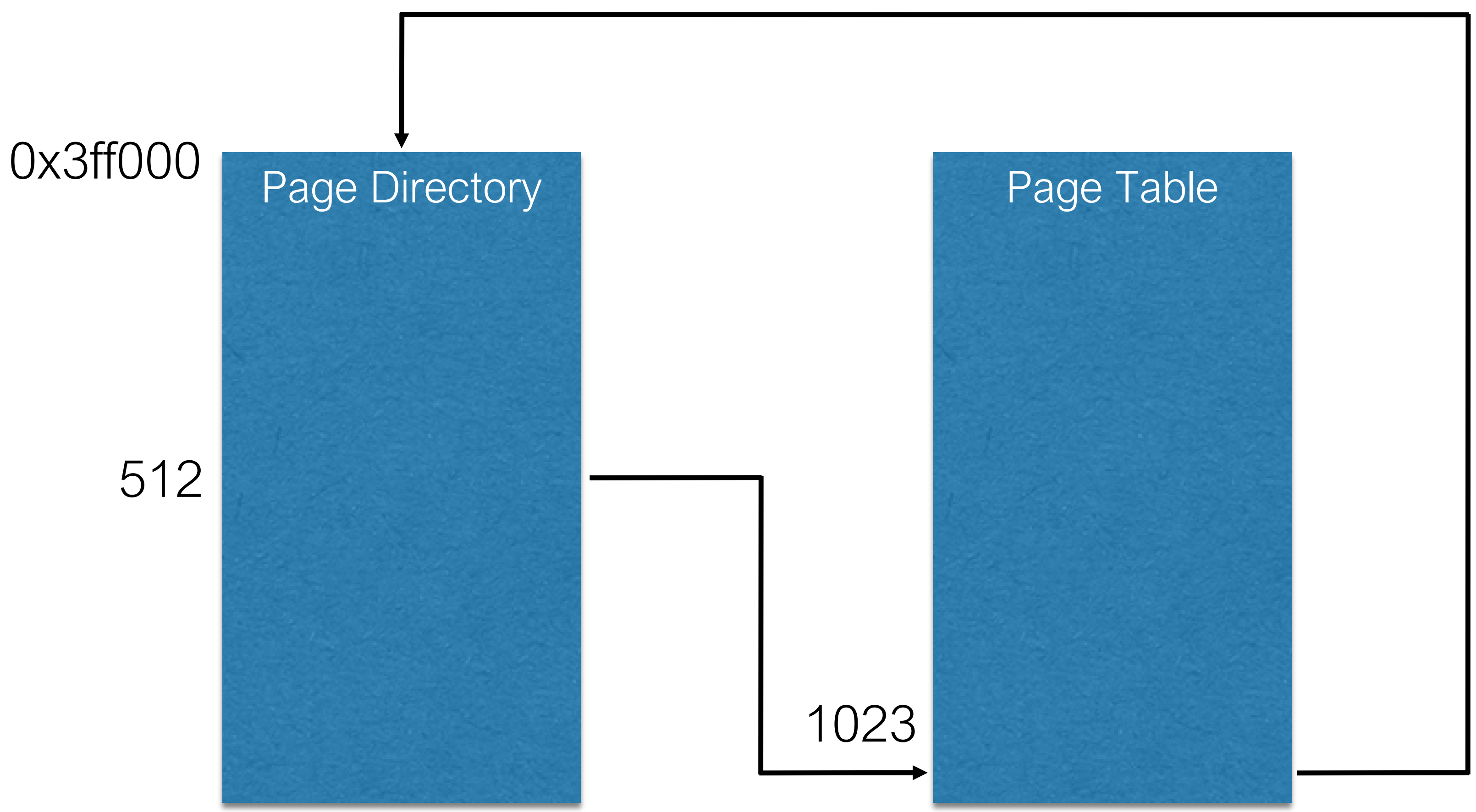
- Once we've put the processor into virtual address mode, we can *only* use virtual addresses

- We need to be able to read and write page tables by referencing them through virtual addresses
  - Corollary: *there must be a virtual address that maps back to the page directory itself*

Paging Tricks: Self-reference

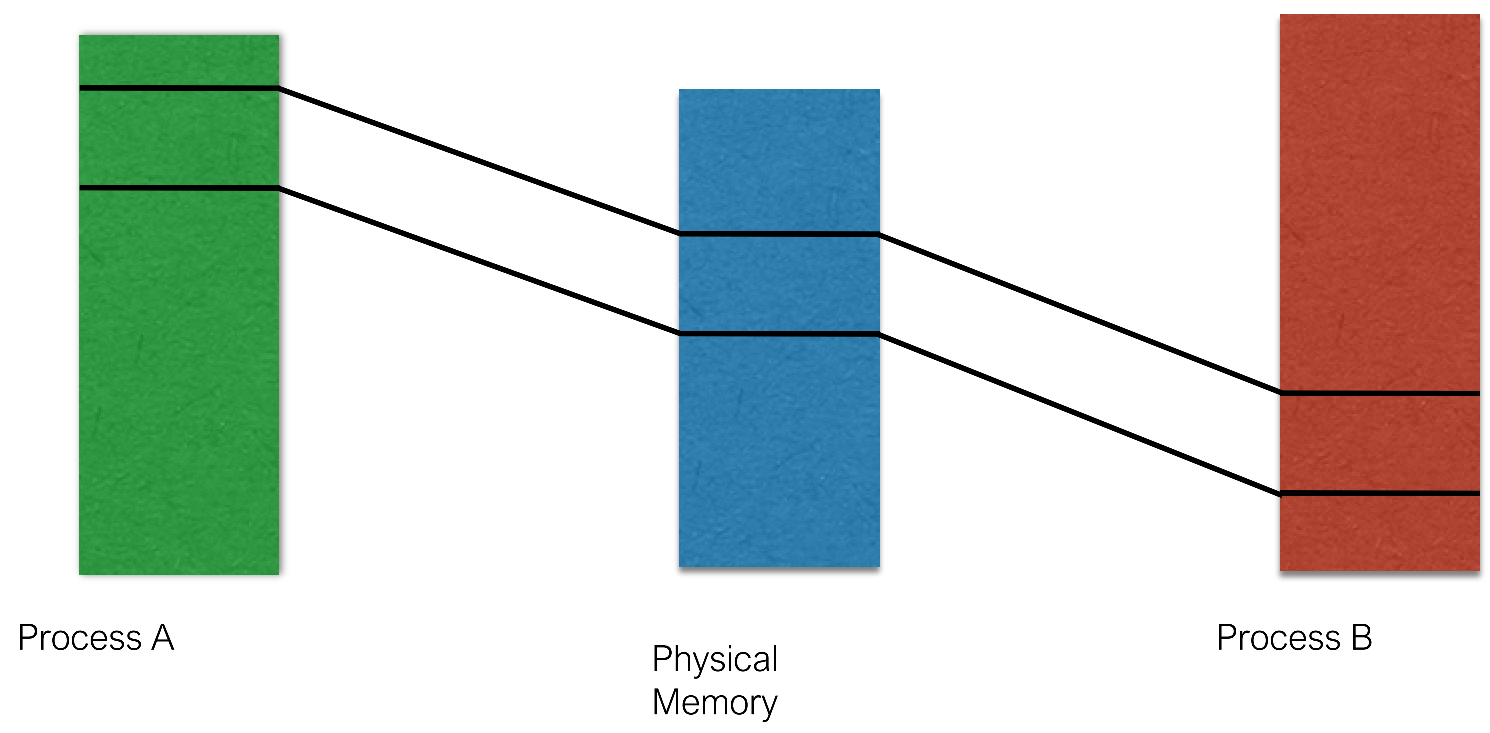
- There is! In xv6, 0x80000000 + PhysAddr = VirtAddr (for low kernel memory only)
- So if our page table is at physical address 0x3ff000, we can access it through 0x803ff000
- Page tables have to be set up with this mapping:

Mapping page table at VA 0x803ff000 PA 0x3ff000



Paging Tricks: Shared Memory

- This one is pretty simple: just make virtual addresses of two processes point to the same physical memory



Paging Tricks: Copy-on-write

- In UNIX, fork() creates a complete copy of the memory of another process

- This is really inefficient – takes time to make the copy, wastes memory unless the new process makes changes

Paging Tricks: Copy-on-write

- Instead: set up new process's mappings so they point to the same pages as the old process
    - **But: make the mapping read-only using page permissions**

- **Now if the process tries to write to one of its pages, we trap into the kernel (via a page fault), make a real copy, and then perform the write**

Paging Tricks: Memory-mapped files

- Normal way of reading a file: read() some data into a buffer, work on it, write() it back out
- Instead we might want to *map* a file into memory – virtual address X+N would refer to the Nth byte in the file
- We can again use paging for this – mark the pages corresponding to the file not present, then when someone tries to read them, read in from disk
- We can write back changes either immediately (by making pages write protected) or when the process exits

Paging Tricks: Lazy Memory Allocation

- Programmers are terrible people and sometimes allocate more memory than they really need
- Example:
  - allocating a large buffer for user input, then only using the very beginning
- We'd like to not give them any more than they are actually using

Paging Tricks: Lazy Memory Allocation

- Solution: make the system call that allocates memory essentially do nothing
  - But keep track of the fact that the process has allocated that virtual address range
- If the process actually tries to use the memory, we will get a page fault and can then allocate a page on demand
- If the pages are never touched, will never allocate

Paging Tricks: Lazy Memory Allocation

- Beware – we can end up handing out more memory than we actually have this way
- This is called *overcommitting*
- Linux actually does this intentionally – malloc() almost never fails
- But if you try to use all that memory, and the system runs out, the dreaded OOM-Killer will be invoked

Paging Tricks: Memory Breakpoints

- When using a debugger, we might want to watch all changes to a particular piece of memory
- E.g., to find out what code sets a global variable
- x86 has hardware support for triggering a debugger when memory is accessed
  - But it's limited to just 4 memory locations
  - No good for monitoring larger data structures, e.g.

Paging Tricks: Memory Breakpoints

- With a bit of help from the OS, we can do better
- Any time we set a memory breakpoint, mark the entire page non-present (or read-only for memory write breakpoints)
- Any access will trap into the OS, which can notify our debugger; when the debugger continues, the OS can finish handling the fault

For more details: *How to do a million watchpoints: Efficient Debugging using Dynamic Instrumentation,* Zhao et al.

- Virtual Memory
- XV6 Implementation
- Page Replacement Algorithms

Page Replacement Algorithms

- If we can't fit everything in memory, eventually we will have to choose something to evict and write to disk
- As with scheduling algorithms, this is a well-studied area and there are lots of strategies
- This general principle shows up any time we have a limited-size cache – strategies described here apply to all of them

The "Optimal" Algorithm

- At the time of a fault, consider the set of pages in memory
- Given the code executing, they will each be referenced some number of instructions from now
- To choose one to evict, just pick the one that's furthest from being referenced

The "Optimal" Algorithm

"Prediction is very difficult, especially about the future."

– Niels Bohr

- **Problem**: requires the OS to predict the future
- Still, it's useful as a goal to work toward
- We can benchmark other algorithms by how close they get to being optimal

Not Recently Used

- We saw that page table entries contain bits that indicate whether the page was recently *referenced* or *modified (R & M)*

- We can use these bits to track which pages in memory have not been touched in a while

- If a page hasn't been used in a while, it may be a good candidate for eviction

Implementing Not Recently Used

- If we just rely on the CPU to mark referenced pages, over time eventually all pages would be referenced
- Instead we can have the OS periodically clear the referenced bit (say, every clock tick)
- Now if the referenced bit is 0, we know the page has not been referenced in at least one clock tick

First In, First Out (FIFO)

- Simple algorithm:
  - Keep a linked list of pages in the order they were brought into memory
  - Tail is most recent, head is oldest
  - To evict, throw out the one at the head of the list
- **But**: just because it's the oldest doesn't mean it's not being used!

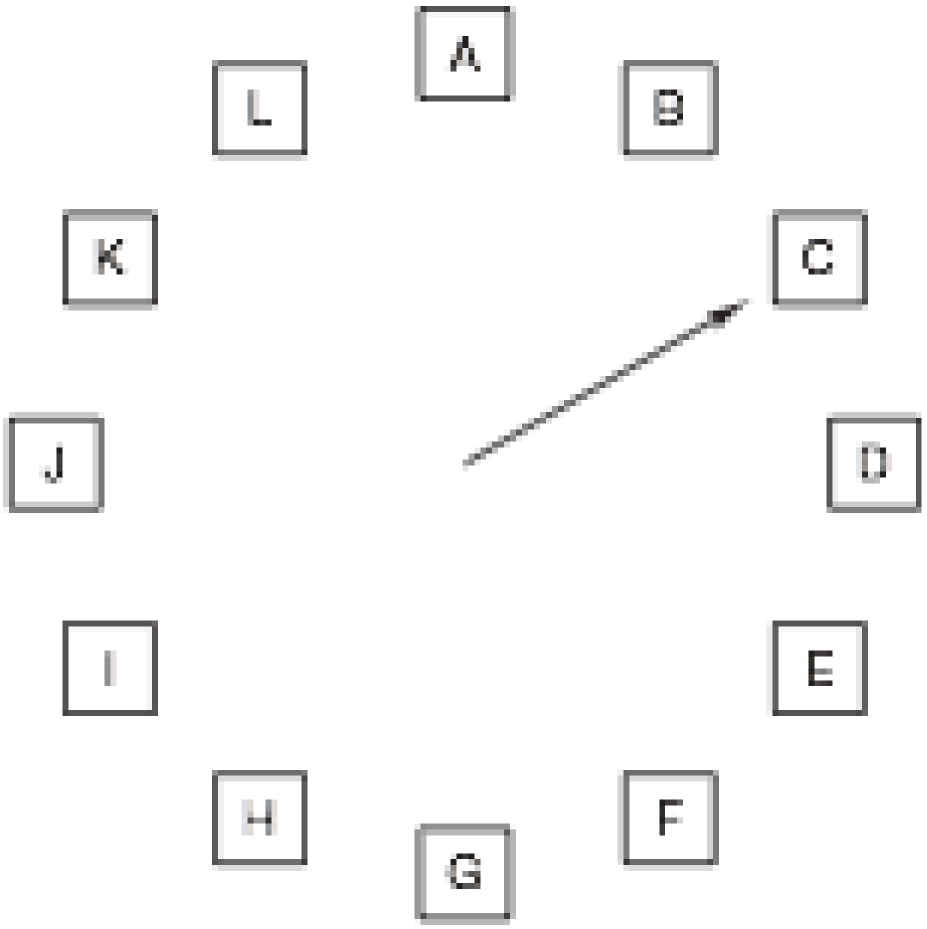
Second Chance

- Since we have the referenced bit available, we can do slightly better than FIFO
- Before throwing out the oldest page, check if it's been referenced
- If it has, clear the referenced bit and move it to the back of the list (as though it were a new page)
- Keep looking through the list for something to evict

Clock

- FIFO/Second Chance is inefficient because it may have to move around entries in the list often
- As an optimization, we can imagine the pages arranged in a circle and keep a pointer (or index) indicating the position of the "clock hand"
- Clock hand points to the oldest page – so now we can apply Second Chance by just updating the position of the clock hand

Clock



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page
R = 1: Clear R and advance hand

Least Recently Used (LRU)

- Observation: pages that have been used a lot recently will probably be used again soon
- So a good strategy might be: throw out the *least recently used* page
- Very expensive to implement, though

LRU Implementation

- One possibility: keep a list of pages (as with FIFO) and update it every time memory is referenced

- Or, if we have extra hardware support – have hardware write a timestamp to the PTE every time every time a page is referenced

- Neither of these is particularly appealing

Not Frequently Used (NFU)

- We can (approximately) simulate LRU in software reasonably cheaply though
- Maintain a list of counters, one per page
- At each clock tick, if a page has been referenced, update its counter
- Now we have a rough count of how often each page has been referenced over time

NFU Problems

- A page might be referenced very often early on, and then never referenced again
- But its count will remain high for a long time, preventing it from getting evicted
- Meanwhile, a page that is referenced periodically (say every 20 ticks) may have a lower count but be in active use

Aging

- To solve this problem, we can have counter values decay over time
- Each clock tick, shift all counters right by one bit
- Add in the reference bit as the leftmost bit
- To evict, just choose the lowest counter value – because more recent references are in the more significant digits, they have greater weight

LRU Approximation

- Aging is only an approximation of an actual LRU algorithm:
  - We don't get any information about how often something was referenced between two clock ticks
  - If our counter is N bits, our memory only extends back N clock ticks – so we can't distinguish how old things are past that point

Working Set

- Most processes don't reference pages randomly scattered around memory
- Instead, they have ***locality of reference*** – tend to reference only a small set of pages in a given time period
- We call the set of pages currently being used by a process its ***working set***
- If you don't have enough memory to hold the working set, you will constantly be swapping – known as ***thrashing***

Prepaging

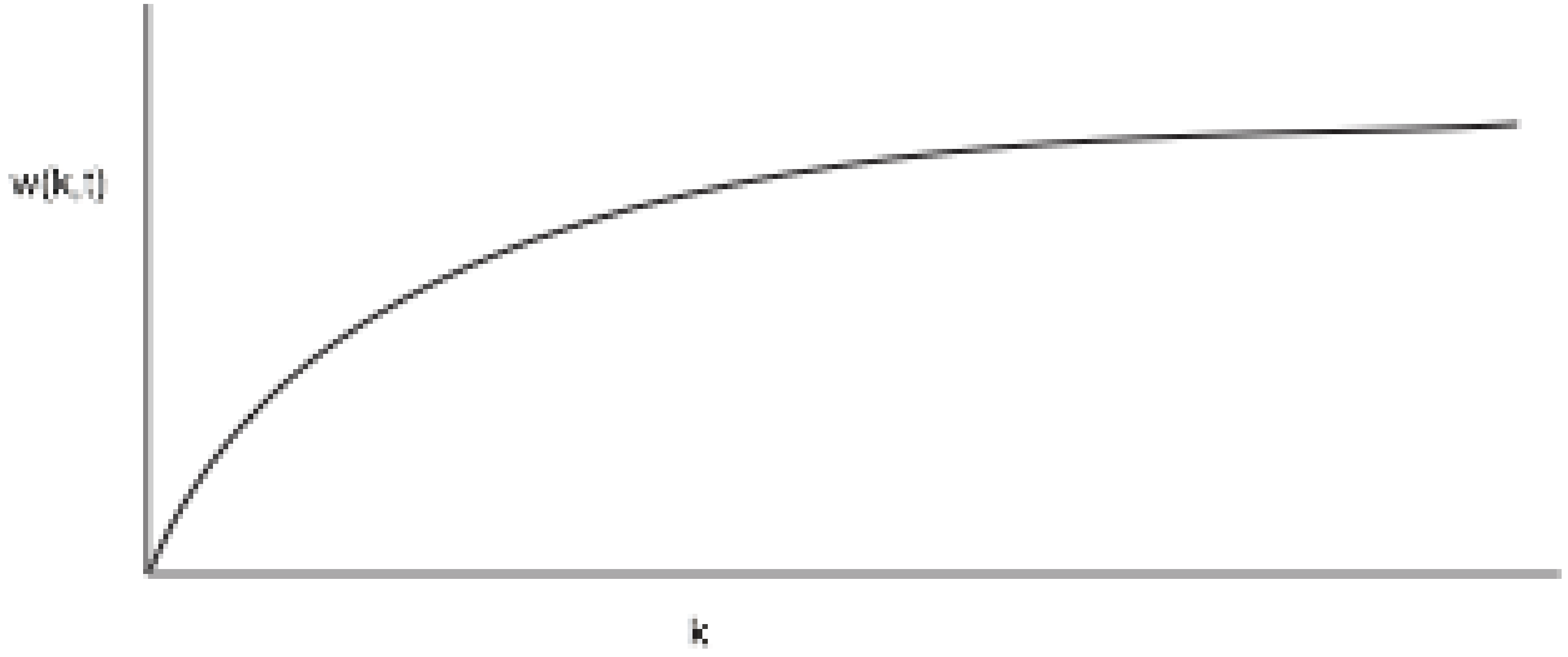
- When we want to bring a process back in after swapping it in, we have the option of doing nothing
  - When the process runs it will just continually page fault until all pages in the working set are in memory
- Downside: each page fault requires some time to process – this is *slow*
- Instead we can try to load the process's working set all at once; this is called *prepaging*

Size of the Working Set

- The working set *w(k,t)* is more precisely defined as the number of pages referenced in the last *k* memory references at time *t*
- If the process never accessed the same page twice, we would have w(k,t) = k
- But because of reference of locality, w(k,t) stops increasing as k gets large

Working Set – Algorithm

- We can take advantage of the fact that the working set of a process is roughly a fixed size after some warmup period



Tracking the Working Set

- For performance, we would like to load the exact working set into the process when it's paged back in
- In theory, we can just fix some value *k* and then track the pages touched in the last *k* memory references
- But once again, doing *anything* on every memory reference is very very slow

Approximating the Working Set

- As we saw with LRU, we may be able to make do with an *approximation* of the working set

- For example: pages referenced in the last τ ms

  - Note: only track time when the process is actually executing – its *current virtual time*

- Now we can take advantage of the CPU features that set the "referenced" bit automatically for us

Working Set – Algorithm