東南大學
SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

Chapter 8. Main Memory

A/Prof. Kai Dong

Warm-up

Memory API

```
void func() {
        int x;
        int *x = (int *) malloc(sizeof(int));
}
```

- **Stack memory**
  - The allocations and deallocations of the stack memory are managed implicitly by the compiler. So it is sometimes called automatic memory.
- **Heap memory**
  - All allocations and deallocations are explicitly handled by the programmer. The cause of many bugs!

Warm-up

What is the Output?

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
 int *p = malloc(sizeof(int));
 assert(p != NULL);
 printf("(%d) memory address of p:
       %08x\n", getpid(), (
       unsigned) p);
 *p = 0;
 while (true) {
  Spin(1);
  *p = *p + 1;
  printf("(%d) p: %d\n", getpid(),
        *p);
 }
 return 0;
}

/* For this example to work,
     address space randomization
     should be disabled */
```

```
prompt> ./mem
(2134) memory address of p:
       00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
^C
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p:
       00200000
(24114) memory address of p:
       00200000
(24113) p: 1
(24114) p: 1
(24113) p: 2
(24113) p: 3
(24114) p: 2
(24114) p: 3
...
```

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a stall
- Cache sits between main memory and CPU registers

Background

Address Binding

- Addresses represented in different ways at different stages of a program’s life
  - Source code addresses usually symbolic
    - e.g., “int count”
  - Compiled code addresses bind to relocatable addresses
    - e.g., “14 bytes from beginning of this module”
  - Linker or loader will bind relocatable addresses to absolute addresses
    - e.g., “74014”
  - Each binding maps one address space to another

Background

Address Binding Time

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - Load time: Must generate relocatable code if memory location is not known at compile time
  - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)
    - Most general-purpose operating systems use this method

Background

Dynamic Loading & Linking

- Dynamic loading — a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format.
- Static linking — system libraries and program code combined by the loader into the binary program image
- Dynamic linking — linking postponed until execution time
- Small piece of code, stub, used to locate the appropriate memory-resident library routine

Background

Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address — generated by the CPU; also referred to as virtual address
  - Physical address — address seen by the memory unit
- Logical and physical addresses are identical in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- Logical address space is the set of all logical addresses generated by a program
- Physical address space is the set of all physical addresses generated by a program

Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Address Translation

- The C-language representation

```
void func() {
        int x = 3000;
        x = x + 3;
}
```

- The compiler turns it to assembly

```
128:    mov 0x0(%ebx), %eax       ; load 0+ebx into eax
132:    add $0x03, %eax           ; add 3 to eax register
135:    mov %eax, 0x0(%ebx)       ; store eax back to mem
```

- The following memory accesses take place.
    1. Fetch instruction at address 128
    2. Execute this instruction
    3. (load from address 0 + *ebx*)
    4. Fetch instruction at address 132
    5. Execute this instruction
    6. (no memory reference)
    7. Fetch the instruction at address 135
    8. Execute this instruction
    9. (store to address 0 + *ebx*)

Address Translation

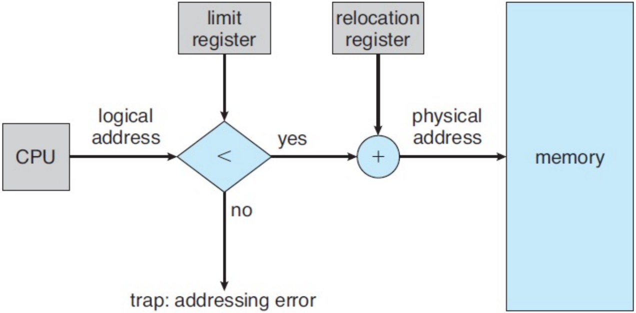
Base and Limit

- How can we relocate this process in memory in a way that is transparent to the process?
- A pair of **base and limit / bound registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

$$\textit{physical address} = \textit{virtual address} + \textit{base}$$

Address Translation

Base and Limit (contd.)



Address Translation

Memory Management Unit

- The base and limit registers are hardware structures kept on the chip (one pair per CPU).
- The part of the processor that helps with address translation is the **memory management unit** (**MMU**); as we develop more sophisticated memory management techniques, we will be adding more circuitry to the MMU.

Address Translation

Hardware & OS Support

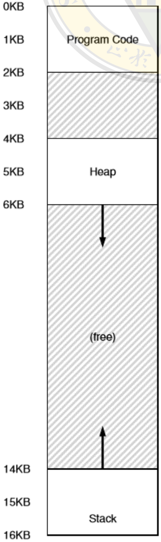
Hardware Requirements	Notes
Privileged mode	Needed to prevent user-mode processes from executing privileged operations
Base / limit registers	Need pair of registers per CPU to support address translation and bounds checks
Ability to translate virtual addresses and check if within bounds	Circuitry to do translations and check limits; in this case, quite simple
Privileged instructions to update base / limit	OS must be able to set these values before letting a user program run
Privileged instructions to register exception handlers	OS must be able to tell hardware what code to run if exception occurs
Ability to raise exceptions	When processes try to access privileged instructions or out-of-limit memory

OS Requirements	Notes
Memory management	Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via *freelist*
Base / limit management	Must set base / limit properly upon context switch
Exception handling	Code to run when exceptions arise; likely action is to terminate offending process

Address Translation

Summary

- Base-and-limit virtualization satisfies
  - Transparency !
  - Protection !
  - Efficiency ?
    - Yes in time, but no in space.
- **Internal fragmentation**
  - The space between the stack and heap of a process is wasted.
  - We are going to discuss segmentation, a slight generalization of base and bounds.



Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Segmentation

- **Segmentation**: generalized base / limit
- Have a base and bounds pair per logical segment of the address space, instead of having just one base and bounds pair in the MMU.
- Support a large address space with (potentially) a lot of free space between the stack and the heap.

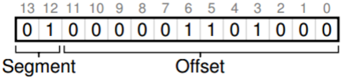
Segmentation

Generalized Base / Limit

- Segment registers:

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

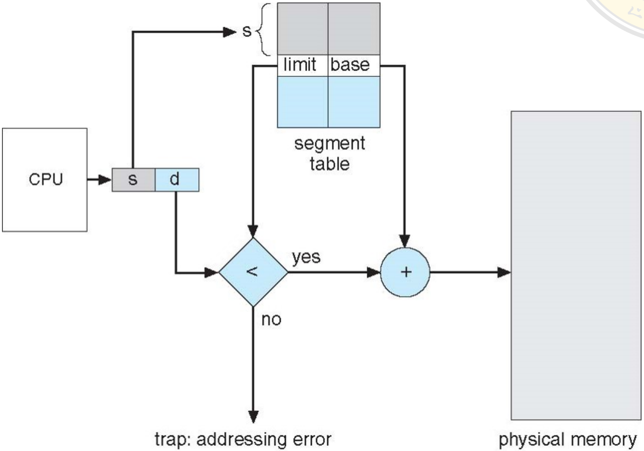
- How to perform address translation? Base + offset
- What if we tried to refer to an illegal address? A segmentation fault
- How does the hardware know the offset into a segment, and to which segment an address refers? Two bits to represent the segments. Others to represent the offset into a segment.



- What about stack? An extra hardware support.
- Any support for sharing? Protection bits.

Segmentation

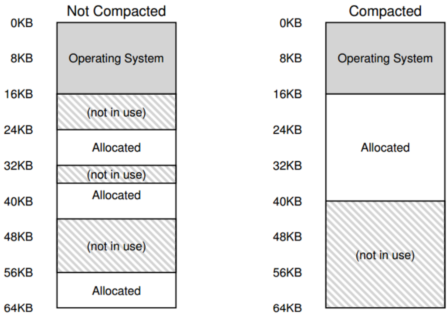
Segmentation Hardware



Segmentation

Segmentation OS Support

- What should the OS do on a context switch?
  - The segment registers must be saved and restored.
- How to manage free space in physical memory?
  - The **external fragmentation** problem.
  - Compaction: Rearranging the existing segments. But expensive.



Segmentation

Summary

- Segmentation is a solution to internal fragmentation.
- One problem with segmentation is external fragmentation.
- External Fragmentation — total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation — allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Contents

1. Background
2. Address Translation
3. Segmentation
4. **Free Space Management**
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Free Space Management

- Free Space Management is easy if the space is divided into fixed-sized units;
  - Allocate memory to processes using default settings.
- It becomes more difficult when the free space consists of variable-sized units.
  - Allocate memory to different segments of processes.
- This problem is known as external fragmentation.
  - Compaction works, but is also expensive.

Free Space Management

Mechanisms & Policies

- What is behind the basic interface such as *malloc*() and *free*()?
  - Mechanisms: Splitting, coalescing.
  - Policies: Various strategies.
    - Best-fit (smallest fit)
    - Worst-fit
    - First-fit
    - Next-fit
- Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of decreasing time and storage utilization.
- Neither first-fit nor best-fit is clearly better than the other in terms of storage utilization, but first-fit is generally faster.

Free Space Management

Example

Given six memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Rank the algorithms in terms of how efficiently they use memory.

First-fit (next-fit)	Best-fit	Worst-fit
100	100	100
500 — 212 — 112	500 — 417	500 — 417
200	200 — 112	200
300	300 — 212	300
600 — 417	600 — 426	600 — 212 — 112

efficiency: best-fit > first-fit = worst-fit

Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

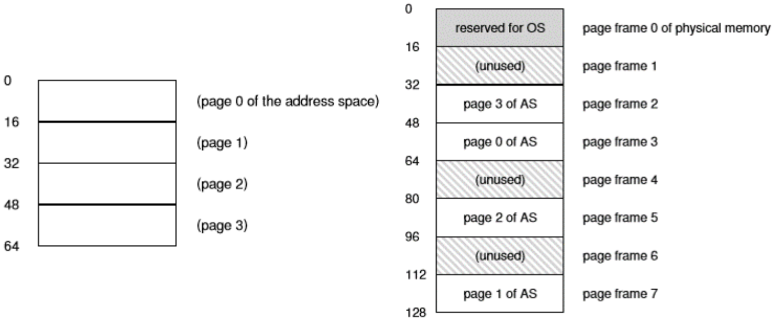
Paging

Why not Segmentation?

- **Segmentation**
  - to chop up space into different-size chunks.
  - The space itself can become fragmented, and thus allocation becomes more challenging over time.
- **Paging**
  - to chop up space into fixed-sized pieces, i.e., **pages**.
  - The physical memory can be viewed as an array of fixed-sized slots called page frames; each of these frames can contain a single virtual-memory page.

Paging

Paging Overview



Paging

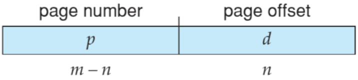
Why Paging?

- **Flexibility**: With a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space.
- **Simplicity**: The free-space management can be easy.

Paging

Address Translation

- **Page Table**: stores address translations for each of the virtual pages of the address space, thus letting us know where in physical memory each page reside.
  - Example in the previous slide: (Virtual Page 0 ⟶ Physical Frame 3), (VP 1 ⟶ PF 7), (VP 2 ⟶ PF 5), and (VP 3 ⟶ PF 2).
- Most (but not all) page tables we discuss are per-process data structures.
- Virtual address has two components: the **virtual page number** (VPN), and the **offset** within the page.



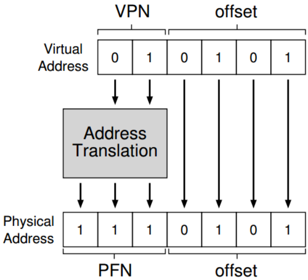
Paging

Address Translation (contd.)

- An address-translation example

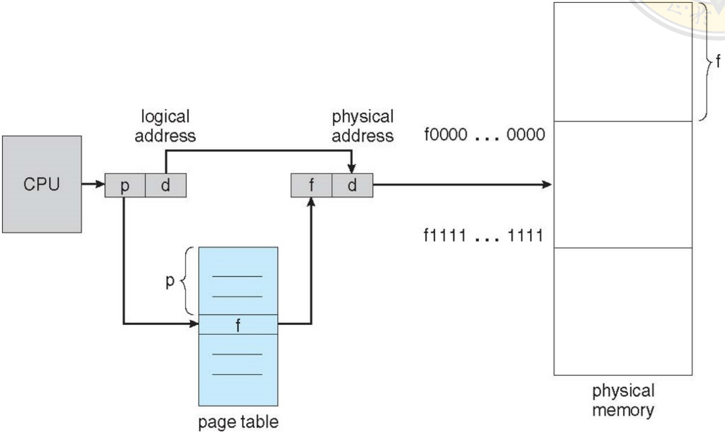
```
movl 21, %eax    ; 21 is a virtual address
```

  - $21D = 010101B \Rightarrow VPN = 01B, offset = 0101B$
  - Using our previous page table, we have (VP 1 ⟶ PF 7)
  - The **physical frame number** (PFN) is 7D (111B)
  - The physical address is $1110101B = 117D$



Paging

Paging Hardware



Paging

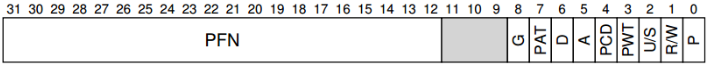
Where are Page Tables Stored?

- In **memory management unit** (MMU)?
  - Page tables can be terribly large.
    - Question: What's the size of a page table with 32-bit address space and 4KB pages? assuming we need 4 bytes per **page table entry** (PTE) to hold the physical translation plus any other useful stuff.
  - Page tables are stored for each process in kernel physical memory, NOT in MMU.
- In physical memory!
  - **Page-table base register** (PTBR) points to the page table
  - **Page-table length register** (PTLR) indicates size of the page table
- PTBR and PTLR in MMU.

Paging

What’s Actually in the Page Table?

- The OS indexes the array by the virtual page number (VPN), and looks up the **page-table entry** (PTE).
- Is VPN in PTE? See the x86 PTE example



- A PTE consists of:
  1. A 20-bit **PFN**.
  2. A **valid bit** indicates whether the particular translation is valid.
  3. A **present bit** (P) indicates whether this page is in physical memory or on disk.
  4. A **dirty bit** (D) indicates whether the page has been modified.
  5. A **reference bit** (A, a.k.a. accessed bit) tracks whether a page has been accessed.
  6. A user/supervisor bit (U/S) determines if user-mode processes can access the page.
  7. A few bits (PWT, PCD, PAT, and G) determine how hardware caching works for the page.

Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Translation Lookaside Buffer

Paging: also too Slow

- For every memory reference, paging requires us to perform one extra memory reference in order to first fetch the translation from the page table. Extra memory references are costly, and in this case will likely slow down the process by a factor of two or more.
- An example:

```
movl 21, %eax    ; 21 is a virtual address
```

  1. Obtain the physical address of the starting location of the page table from a page-table base register.
  2. Fetch the proper page table entry from the process's page table.
  3. Translate the virtual address (21) into the correct physical address.
  4. Fetch the data from the physical address and put it into register eax.

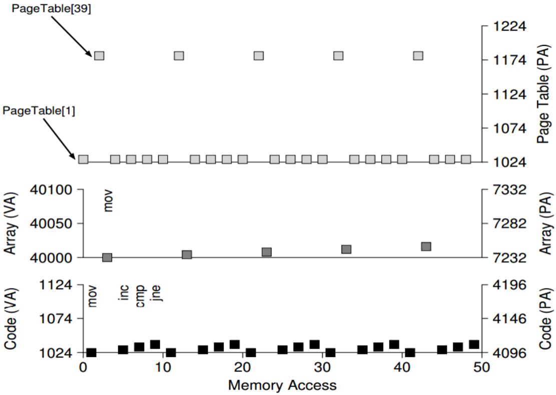
- Paging requires us to perform one extra memory reference!

Translation Lookaside Buffer

Paging: also too Slow (contd.)

```c
int array[1000];
...
for (i = 0; i < 1000; i ++)
  array[i] = 0;
```

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```



Translation Lookaside Buffer

- A **translation lookaside buffer**, or TLB is
  - part of the chip's memory-management unit (MMU);
  - simply a hardware cache of popular virtual-to-physical address translation;
  - also called an address-translation cache.
    - TLB hit
    - TLB miss

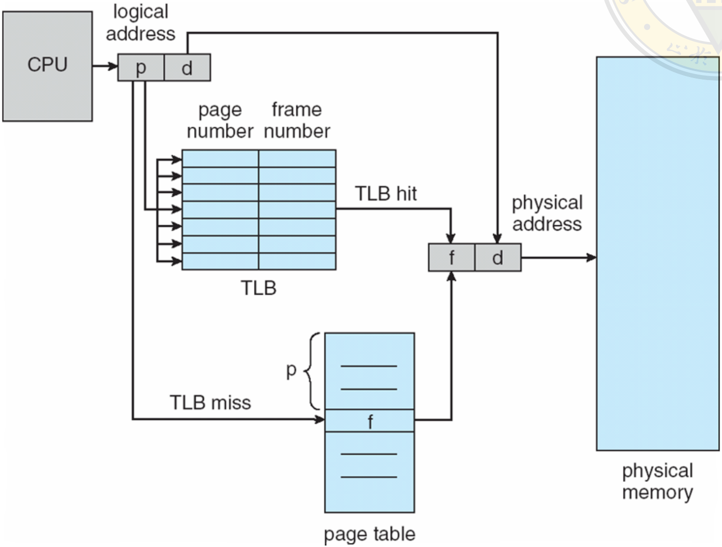
Translation Lookaside Buffer

TLB Basic Algorithm

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB hit
        if (CanAccess(TlbEntry.ProtectBits) == True)
                Offset = VirtualAddress & OFFSET_MASK
                PhysAddr = (TlbEntry.PFN<<SHIFT) | Offset
                AccessMemory(PhysAddr)
        else
                RaiseException(PROTECTION_FAULT)
else                    // TLB miss
        PTEAddr PTBR + (VPN *sizeof(PTE))
        PTE = AccessMemory(PTEAddr)
        if (PTE.Valid == False)
                RaiseException(SEGMENTATION_FAULT)
        else if (CanAccess(PTE.ProtectBits) == False)
                RaiseException(PROTECTION_FAULT)
        else
                TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
                RetryInstruction()
```

Translation Lookaside Buffer

Paging Hardware With TLB



Translation Lookaside Buffer

Who Handles the TLB Miss?

- In the olden days, the hardware.
  - Intel x86 architecture
- In the modern era, the OS.
  - Sun’s SPARC

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB hit
        if (CanAccess(TlbEntry.ProtectBits) == True)
                Offset = VirtualAddress & OFFSET_MASK
                PhysAddr = (TlbEntry.PFN<<SHIFT) | Offset
                AccessMemory(PhysAddr)
        else
                RaiseException(PROTECTION_FAULT)
else                    // TLB miss
        RaiseException(TLB_MISS)
```

Translation Lookaside Buffer

Issues

- Return-from-trap
  - Which instruction to execute next?
  - Normally the next, but here to execute the instruction that caused the trap.
  - A different PC.
- TLB miss-handling
  - An infinite chain of TLB misses may occur. When?
  - Keep TLB miss handlers in physical memory, or reserve some entries in the TLB for permanently-valid translations.
- Why OS?
  - Flexibility
    - The OS can use any data structure it wants to implement the page table, without necessitating hardware change.
  - Simplicity
    - The hardware doesn't have to do much on a miss.

Translation Lookaside Buffer

TLB Contents

- What are TLB contents?

VPN	PFN	other bits

  - A **valid bit**, whether the entry has a valid translation or not.
  - **Protection bits**, how a page can be accessed.
  - A **address-space identifier**,
  - A **dirty bit**, whether or not modified.
  - etc.

Translation Lookaside Buffer

Issues with a Context Switch

- TLB contains virtual-to-physical translations that are only valid for the currently running process;
- How to ensure that the about-to-be-run process does not accidentally use translations from some previously run process?

VPN	PFN	valid	prot
10	100	1	rwx
-	-	0	-
10	170	1	rwx
-	-	0	-

- Flush the TLB on context switches, thus emptying it before running the next process. OR
- Use an address space identifier (ASID).

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	0	-	-
10	170	1	rwx	2
-	-	0	-	-

Translation Lookaside Buffer

Other Issues

- Another case: several entries for different processes that point to the same physical page:

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
-	-	0	-	-
50	101	1	r-x	2
-	-	0	-	-

  - Shared pages: An advantage of paging is the possibility of sharing common code.

- Replacement Policy
  - When we are installing a new entry in the TLB, we have to replace an old one.
  - Which one to replace? With hoping to minimize miss rate.

Translation Lookaside Buffer

Effective Access Time

- TLB Lookup $= \epsilon$ time unit
  - Can be < 10% of memory access time
- Hit ratio $= \alpha$
- Consider $\alpha = 80\%$, $\epsilon = 20ns$ for TLB search, $100ns$ for memory access
- **Effective Access Time** (EAT)

$$EAT = (1 + \epsilon)\alpha + (2 + 2\epsilon)(1 - \alpha)$$

- Consider $\alpha = 80\%$, $\epsilon = 20ns$ for TLB search, $100ns$ for memory access

$$EAT = 0.80 \times 120 + 0.20 \times 240 = 144ns$$

- Consider more realistic hit ratio → $\alpha = 99\%$, $\epsilon = 20ns$ for TLB search, $100ns$ for memory access

$$EAT = 0.99 \times 120 + 0.01 \times 240 = 121ns$$

Translation Lookaside Buffer

Page Size Issue

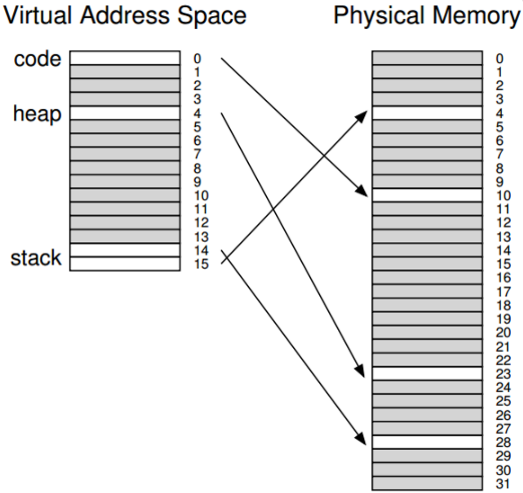
- Page size selection must take into consideration
  - fragmentation (small page)
  - table size / page faults (large page)
  - I/O overhead (large page)
  - locality (small page)
- Most systems use relatively small page sizes in the common case: 4KB (as in x86) or 8KB (as in SPARCv9).

Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Structure of the Page Table

- Most of the page table is unused, full of invalid entries. What a waste!



Structure of the Page Table

Hybrid Approach: Paging and Segments

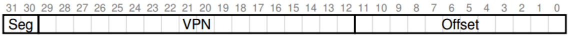
- Instead of having a single page table for the entire address space of the process, why not have one per logical segment?

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
23	1	rw-	1	1
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Structure of the Page Table

Hybrid Approach: Paging and Segments

- One page table per logical segment (instead of per process).
  - Internal fragmentation is solved since segmentation is employed.
  - A pair of base and bound registers, telling the address and size of the segment, and those of the page table.
  - Example: Assume a 32-bit virtual address space with 4KB pages, and an address space split into four segments.



- Problem:
  - Segmentation assumes a certain usage pattern of the address space.
  - Page tables now can be of arbitrary size, thus finding free space for them in memory is more complicated.

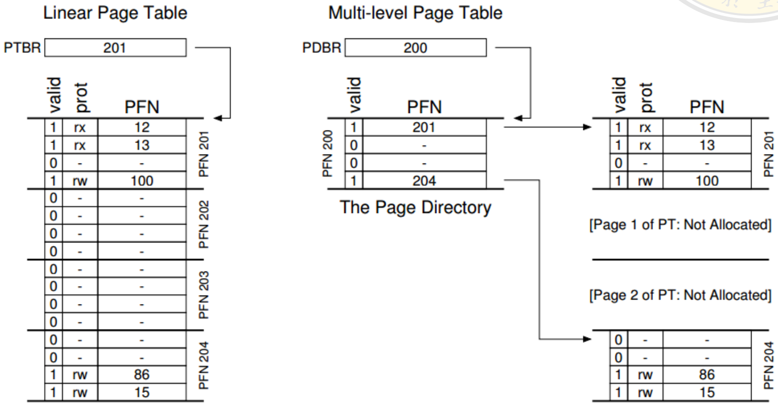
Structure of the Page Table

Hierarchical Page Tables

- Multi-level Page Tables
- Basic idea:
  - Chop up the page table into page-sized units;
  - If an entire page of page-table entries (PTEs) is invalid, don’t allocate that page of the page table at all.
  - A page directory is used to track whether a page of the page table is valid. It tells where a page of the page table is, or that the entire page of the page table contains no valid pages.
- A **page directory** (or outer page) table
  - consists of a number of **page directory entries** (PDE), one entry per page of the page table.
    - A PDE has a valid bit and a page frame number (PFN). If the PDE is valid, it means that at least one of the pages of the page table that the entry points to is valid.

Structure of the Page Table

Two Level Paging Example



Linear Page Table

PTBR 201

valid	prot	PFN	
1	rx	12	PFN 201
1	rx	13	
0	-	-	
1	rw	100	
0	-	-	PFN 202
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN 203
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN 204
0	-	-	
1	rw	86	
1	rw	15	

Multi-level Page Table

PDBR 200

	valid	PFN
PFN 200	1	201
	0	-
	0	-
	1	204

The Page Directory

valid	prot	PFN	
1	rx	12	PFN 201
1	rx	13	
0	-	-	
1	rw	100	

[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]

0	-	-	PFN 204
0	-	-	
1	rw	86	
1	rw	15	

Structure of the Page Table

Two Level Paging

- Advantages:
  - The multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus it is generally compact and supports sparse address spaces.
  - If carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table.
- Disadvantages:
  - Time-space trade-off: There is a cost to multi-level tables; on a TLB miss, two loads from memory will be required to get the right translation information from the page table, in contrast to just one load with a linear page table.
  - Complexity: handling the page-table lookup (on a TLB miss), the hardware or OS is more involved than a simple linear page-table lookup.

Structure of the Page Table

Two Level Paging Control Flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB hit
        if (CanAccess(TlbEntry.ProtectBits) == True)
                Offset = VirtualAddress & OFFSET_MASK
                PhysAddr = (TlbEntry.PFN<<SHIFT) | Offset
                Register = AccessMemory(PhysAddr)
        else    RaiseException(PROTECTION_FAULT)
else                    // TLB miss
        PDIndex = (VPN & PD_MASK) >> PD_SHIFT
        PDEAddr = PDBR + (PDIndex * sizeof(PDE))
        PDE = AccessMemory(PDEAddr)
        if (PDE.Valid == False)
                RaiseException(SEGMENTATION_FAULT)
        else
                PTIndex = (VPN & PT_MASK) >> PT_SHIFT
                PTEAddr + (PDE.PFN << SHIFT) + (PTIndex *sizeof(PTE))
                PTE = AccessMemory(PTEAddr)
                if (PTE.Valid == False)
                        RaiseException(SEGMENTATION_FAULT)
                else if (CanAccess(PTE.ProtectBits) == False)
                        RaiseException(PROTECTION_FAULT)
                else    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
                        RetryInstruction()
```

Structure of the Page Table

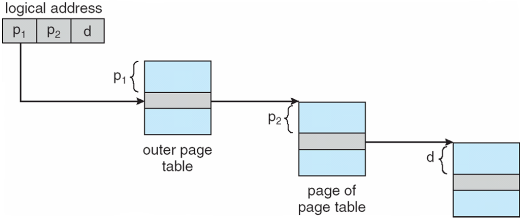
Two Level Paging Example

- Imagine a small address space of size 16KB, with 64-byte pages.
- Thus, we have a 14-bit virtual address space, with 8 bits for the VPN and 6 bits for the offset. (why?)
- A linear page table would have $2^8 = 256$ entries, even if only a small portion of the address space is in use. (why?)
- Assume each PTE is 4 bytes in size.
- Thus, our page table is 1KB (256×4 bytes) in size. Given that we have 64-byte pages, the 1KB page table can be divided into 16 64-byte pages; each page can hold 16 PTEs. (why?)
- We need 4 bits to indicate the page directory index, 4 bits to indicate the page table index in each PDE, and 6 bits to indicate the offset. (why?)

Structure of the Page Table

Popular Page Size

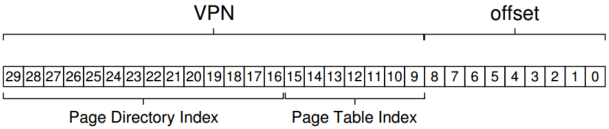
- Why 4KB page size?
- Hint: x86 architecture: 32 bit address space, two-level page table, 4 Byte PTE (why?).



  - $2^{p_1} \times 4Byte = 2^d Byte \Rightarrow$ outer page table in one page
  - $2^{p_2} \times 4Byte = 2^d Byte \Rightarrow$ inner page table in one page
  - $p1 = p2 = 10, d = 12$
  - page size $= 2^d Byte = 4KB$

Structure of the Page Table

Multi Level Page Tables



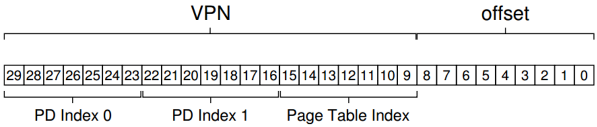
VPN

offset

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

Page Directory Index

Page Table Index



VPN

offset

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

PD Index 0

PD Index 1

Page Table Index

Structure of the Page Table

In Class Exercise

Consider a system with 64 MB of physical memory, 32-bit physical addresses, 32-bit virtual addresses, and 4 KB physical page frames.

(a) Using a single-level paging scheme, what is the maximum number of page table entries for a page table in this system?

(b) Using a two-level paging scheme with a 1024-entry outer-page table, how many bits are needed in the logical address to represent the outer page table? How many bits are needed in the logical address in order to represent the inner page table? How many bits are used to represent the offset within a page?

(c) Suppose a TLB is used with the two-level paging scheme described in part (b), and the TLB has a 90% hit rate. If the TLB access time is 10 ns and memory access time is 100 ns, what is the effective memory access time of the system?

Structure of the Page Table

Key

Consider a system with 64 MB of physical memory, 32-bit physical addresses, 32-bit virtual addresses, and 4 KB physical page frames.

(a) Using a single-level paging scheme, what is the maximum number of page table entries for a page table in this system?

2^{20}

(b) Using a two-level paging scheme with a 1024-entry outer-page table, how many bits are needed in the logical address to represent the outer page table? How many bits are needed in the logical address in order to represent the inner page table? How many bits are used to represent the offset within a page?

10, 10, 12

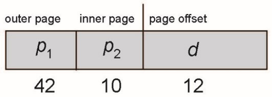
(c) Suppose a TLB is used with the two-level paging scheme described in part (b), and the TLB has a 90% hit rate. If the TLB access time is 10 ns and memory access time is 100 ns, what is the effective memory access time of the system?

$EAT = 90\% \times (10 + 100) + 10\% \times (20 + 300) = 99 + 32 = 131$

Structure of the Page Table

64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is *4KB* (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

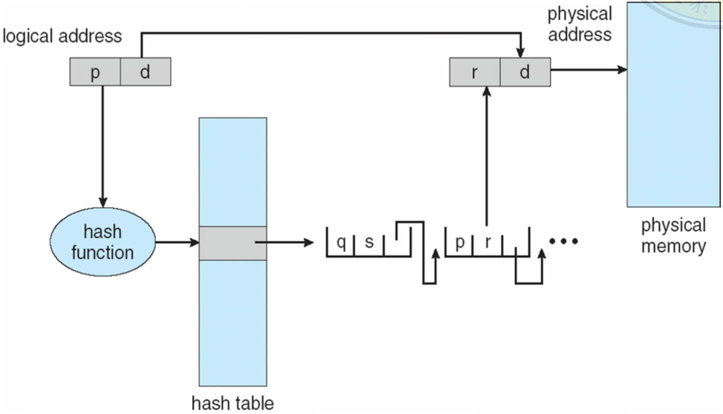
Structure of the Page Table

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  1. the virtual page number
  2. the value of the mapped page frame
  3. a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Structure of the Page Table

Hashed Page Tables (contd.)



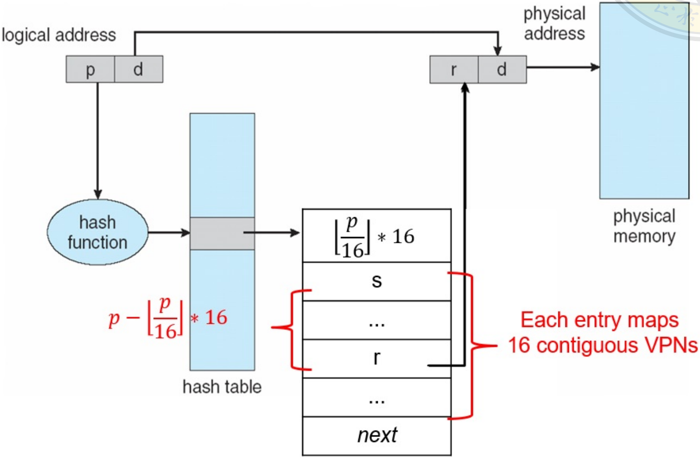
Structure of the Page Table

Clustered Page Tables

- For 64-bit addresses
  - Multi-level page tables require too many levels.
  - Hashed page tables also require too much space:
    - VPN and next pointer
- Clustered page tables: Variation of hashed page tables
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for sparse address spaces (where memory references are non-contiguous and scattered)

Structure of the Page Table

Clustered Page Tables (contd.)



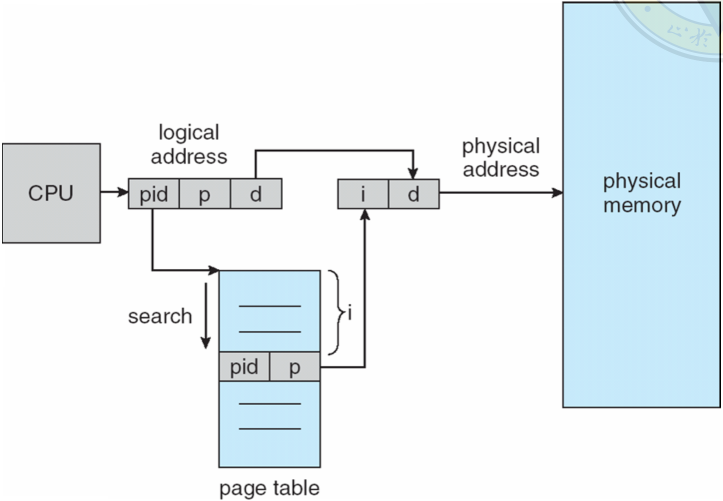
Structure of the Page Table

Inverted Page Tables

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs. Why?
- Use hash table to limit the search to one — or at most a few — page-table entries. What value can be hashed?
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Structure of the Page Table

Inverted Page Tables (contd.)



Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Swapping

- Page tables may still be too big.
- Some systems place such page tables in kernel virtual memory, thereby allowing the system to swap some of these page tables to disk.
- We will detail swapping in the next chapter.

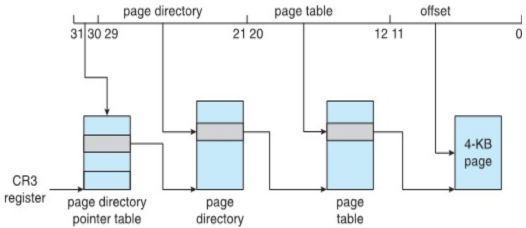
Contents

1. Background
2. Address Translation
3. Segmentation
4. Free Space Management
5. Paging
6. Translation Lookaside Buffer
7. Structure of the Page Table
8. Swapping
9. Example Architectures

Example Architectures

Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create page address extension (PAE), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a page directory pointer table
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits - 64GB of physical memory



page directory
page table
offset
31
30 29
21 20
12 11
0
CR3 register
page directory pointer table
page directory
page table
4-KB page