

CS162
Operating Systems and
Systems Programming
Lecture 12

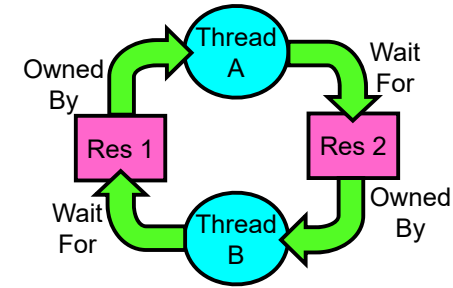
Address Translation

March 5th, 2020
Prof. John Kubiatowicz
<http://cs162.eecs.Berkeley.edu>

Recall: Starvation vs Deadlock



- Starvation: thread waits indefinitely
 - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
 - Thread A owns Res 1 and is waiting for Res 2
 - Thread B owns Res 2 and is waiting for Res 1



- Deadlock \Rightarrow Starvation but not vice versa
 - Starvation can end (but doesn't have to)
 - Deadlock can't end without external intervention

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.2

Recall: Four requirements for Deadlock

- **Mutual exclusion**
 - Only one thread at a time can use a resource.
- **Hold and wait**
 - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**
 - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **Circular wait**
 - There exists a set $\{T_1, \dots, T_n\}$ of waiting threads
 - » T_1 is waiting for a resource that is held by T_2
 - » T_2 is waiting for a resource that is held by T_3
 - » ...
 - » T_n is waiting for a resource that is held by T_1

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.3

Recall: Banker's Algorithm

- Banker's algorithm assumptions:
 - Every thread pre-specifies its *maximum* need for resources
 - » However, it doesn't have to ask for the all at once... (key advantage)
 - Threads may now request and hold dynamically up to the maximum specified number of each resources
- Simple use of the deadlock detection algorithm
 - For each request for resources from a thread:
 - » Technique: pretend each request is granted, then run deadlock detection algorithm, and grant request if result is deadlock free (conservative!)
 - Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \dots, T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
- Banker's algorithm prevents deadlocks involving threads and resources by stalling requests that would lead to deadlock
 - Can't fix all issues – e.g. thread going into an infinite loop!

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.4

Revisit: Deadlock Avoidance using Banker's Algorithm

- Idea: When a thread requests a resource, OS checks if it would result in ~~deadlock~~ **an unsafe state**
 - If not, it grants the resource right away
 - If so, it waits for other threads to release resources
- Example:

Thread A

```
x.Acquire();
y.Acquire();
...
y.Release();
x.Release();
```

Thread B

```
y.Acquire();
x.Acquire();
...
x.Release();
y.Release();
```

Thread B
Waits until
Thread A
releases
resources...

Recall: Does Priority Inversion Cause Deadlock?

- Definition: Priority Inversion
 - A low priority task prevents a high-priority task from running
- Does Priority Inversion cause Deadlock?
- Consider typical case (requires 3 threads):
 - 3 threads, T1, T2, T3 in priority order (T3 highest)
 - T1 grabs lock, T3 tries to acquire, then sleeps, T2 running
 - Will this make progress?
 - No, as long as T2 is running
 - But T2 could stop at any time and the problem would resolve itself...
 - So, this is *not* a deadlock (it is a livelock). But it could last a long time...
- Why is this a priority inversion?
 - T3 is prevented from running by T2

Priority Donation as a remedy to Priority Inversion

- What is *priority donation*?
 - When high priority Thread TB is about to sleep while waiting for a lock held by lower priority Thread TA, it may *temporarily donate* its priority to the holder of the lock if that lock holder has a lower priority
 - So, Priority(TB) => TA until lock is released
 - So, now, TA runs with high priority until it releases its lock, at which time its priority is restored to its original priority
- How does *priority donation* help the *priority inversion* scenario? [T1 has lock, T2 running, T3 blocked on lock]
 - Briefly raise T1 to the same priority as T3 => T1 can run and release lock, allowing T3 to run
 - Does priority donation involve taking lock away from T1?
 - NO! That would break semantics of the lock and potentially corrupt any information protected by lock!

Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
 - Protection
 - Multi-programming
 - Isolation
 - Memory resource management
 - I/O efficiency
 - Sharing
 - Inter-process communication
 - Debugging
 - Demand paging
- Today: Translation



Recall: Four Fundamental OS Concepts

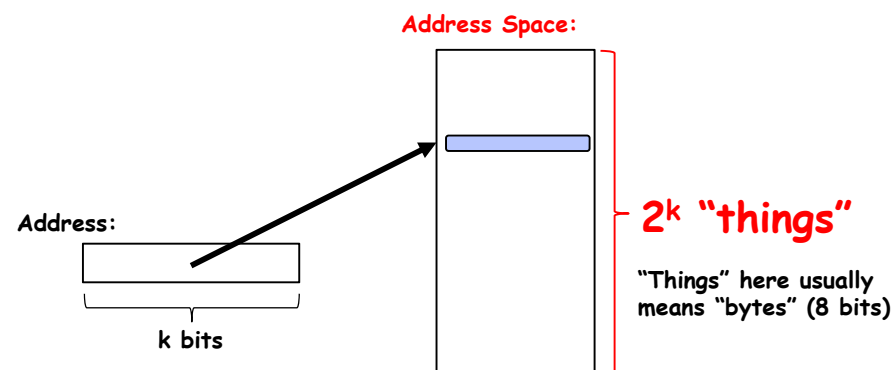
- **Thread: Execution Context**
 - Fully describes program state
 - Program Counter, Registers, Execution Flags, Stack
- **Address space (with or w/o translation)**
 - Set of memory addresses accessible to program (for read or write)
 - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Process: an instance of a running program**
 - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
 - Only the “system” has the ability to access certain resources
 - Combined with translation, isolates programs from each other and the OS from programs

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.9

THE BASICS: Address/Address Space



- What is 2^{10} bytes (where a byte is abbreviated as “B”)?
 - $2^{10} \text{ B} = 1024 \text{ B} = 1 \text{ KB}$ (for memory, $1\text{K} = 1024$, *not* 1000)
- How many bits to address each byte of 4KB page?
 - $4\text{KB} = 4 \times 1\text{KB} = 4 \times 2^{10} = 2^{12} \Rightarrow 12 \text{ bits}$
- How much memory can be addressed with 20 bits? 32 bits? 64 bits?
 - Use 2^k

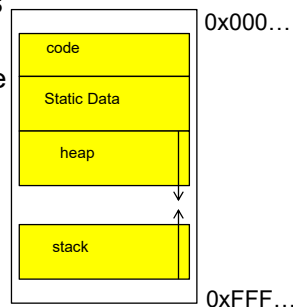
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.10

Address Space, Process Virtual Address Space

- Definition: **Set of accessible addresses and the state associated with them**
 - $2^{32} = \sim 4$ billion **bytes** on a 32-bit machine
- How many 32-bit numbers fit in this address space?
 - 32-bits = 4 bytes, so $2^{32}/4 = 2^{30} = \sim 1$ billion
- What happens when processor reads or writes to an address?
 - Perhaps acts like regular memory
 - Perhaps causes I/O operation
 - » (Memory-mapped I/O)
 - Causes program to abort (segfault)?
 - Communicate with another program
 - ...

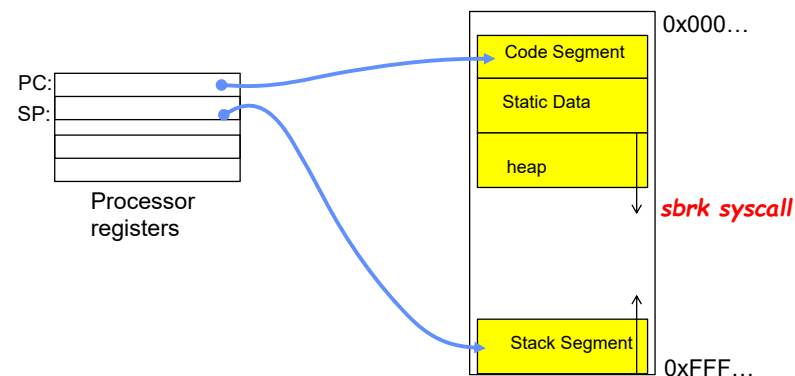


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.11

Recall: Process Address Space: typical structure

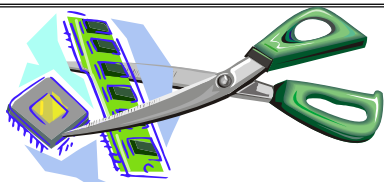


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.12

Virtualizing Resources



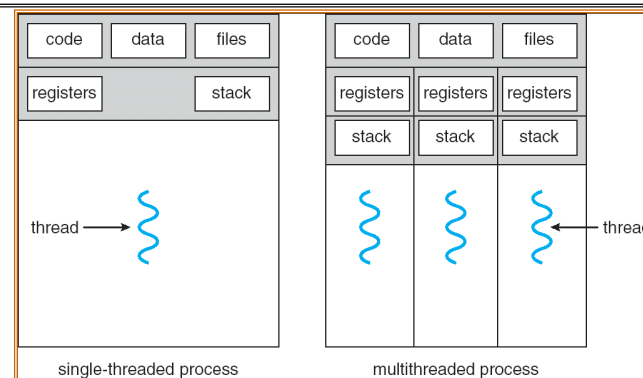
- **Physical Reality:**
Different Processes/Threads share the same hardware
 - Need to multiplex CPU (Just finished: scheduling)
 - Need to multiplex use of Memory (starting today)
 - Need to multiplex disk and devices (later in term)
- **Why worry about memory sharing?**
 - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
 - Consequently, cannot just let different threads of control use the same memory
 - » Physics: two different pieces of data cannot occupy the same locations in memory
 - Probably don't want different threads to even have access to each other's memory if in different processes (protection)

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.13

Recall: Single and Multithreaded Processes



- **Threads encapsulate concurrency**
 - “Active” component of a process
- **Address spaces encapsulate protection**
 - Keeps buggy program from trashing the system
 - “Passive” component of a process

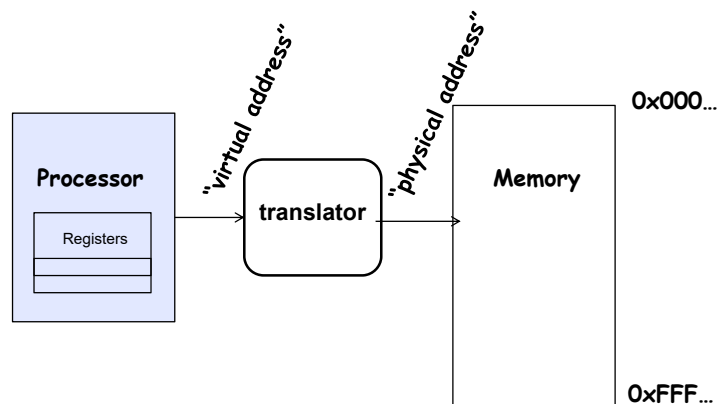
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.14

Recall: Key OS Concept: Address Translation

- Program operates in an address space that is distinct from the physical memory space of the machine



3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.15

Important Aspects of Memory Multiplexing

- **Protection:**
 - Prevent access to private memory of other processes
 - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
 - » Kernel data protected from User programs
 - » Programs protected from themselves
- **Controlled overlap:**
 - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
 - Conversely, would like the ability to overlap when desired (for communication)
- **Translation:**
 - Ability to translate accesses from one address space (virtual) to a different one (physical)
 - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
 - Side effects:
 - » Can be used to avoid overlap
 - » Can be used to give uniform view of memory to programs

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.16

Administrivia

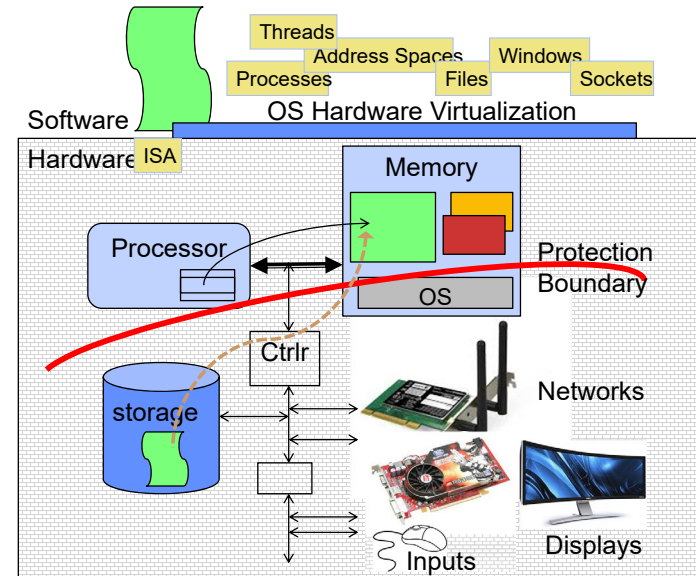
- Midterm 1
 - Regrade requests until next Monday
 - Solutions posted on Resources page (and in piazza)
- Also due: Peer evaluations
 - These are a required mechanism for evaluating group dynamics
 - You get 20 points/partner to distribute as you want:
Example—4 person group, you get $3 \times 20 = 60$ points
 - DO NOT GIVE YOURSELF POINTS!
 - » You are NOT an unbiased evaluator of your group behavior
- Project 2 has been released now
 - Sorry about delay

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.17

Recall: Loading

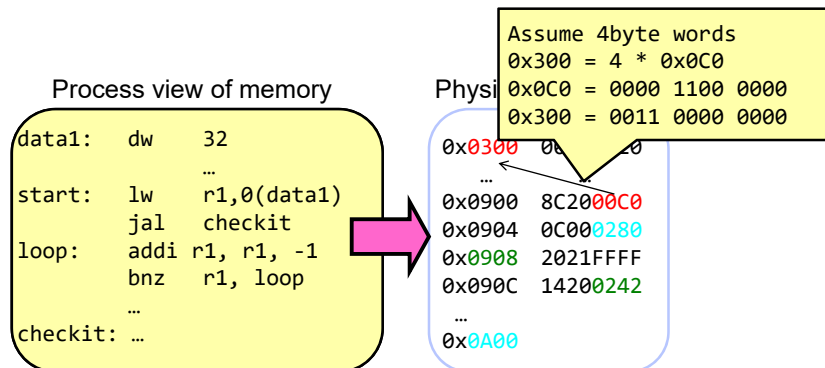


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.18

Binding of Instructions and Data to Memory

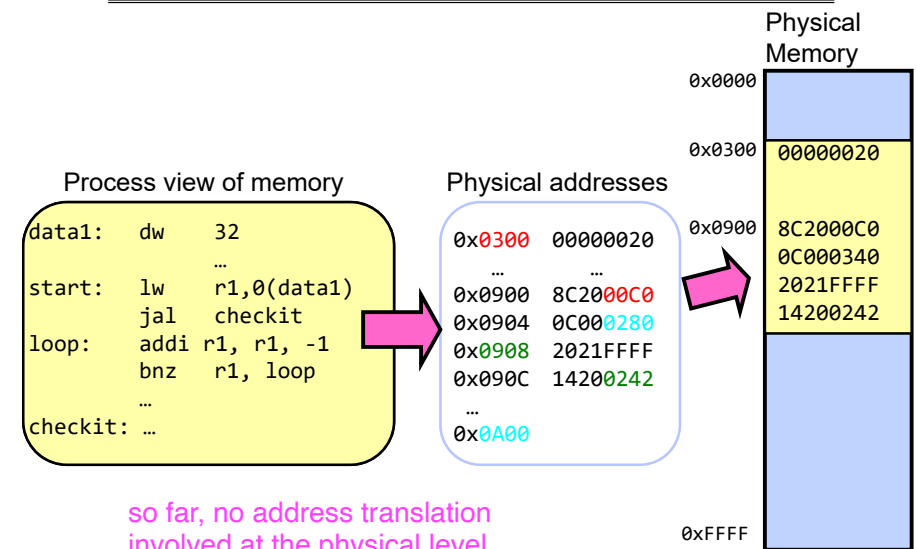


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.19

Binding of Instructions and Data to Memory



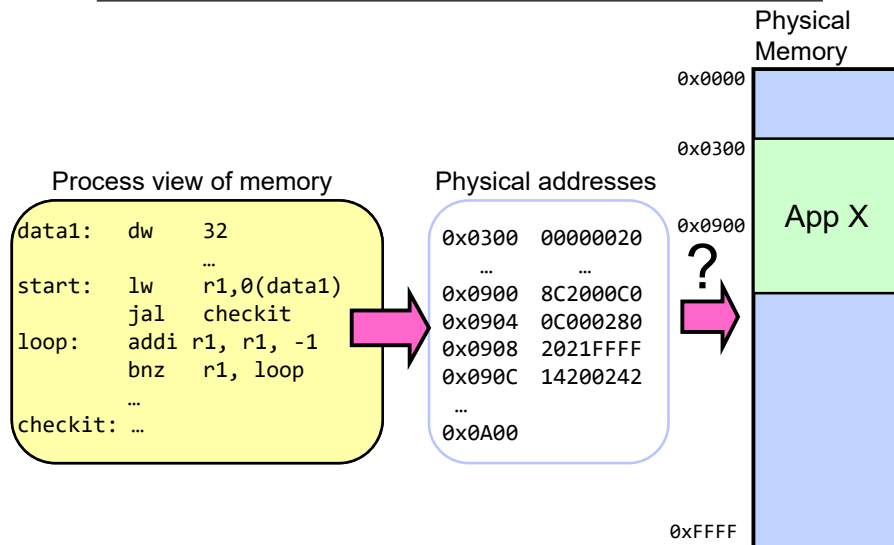
so far, no address translation involved at the physical level.

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.20

Second copy of program from previous example



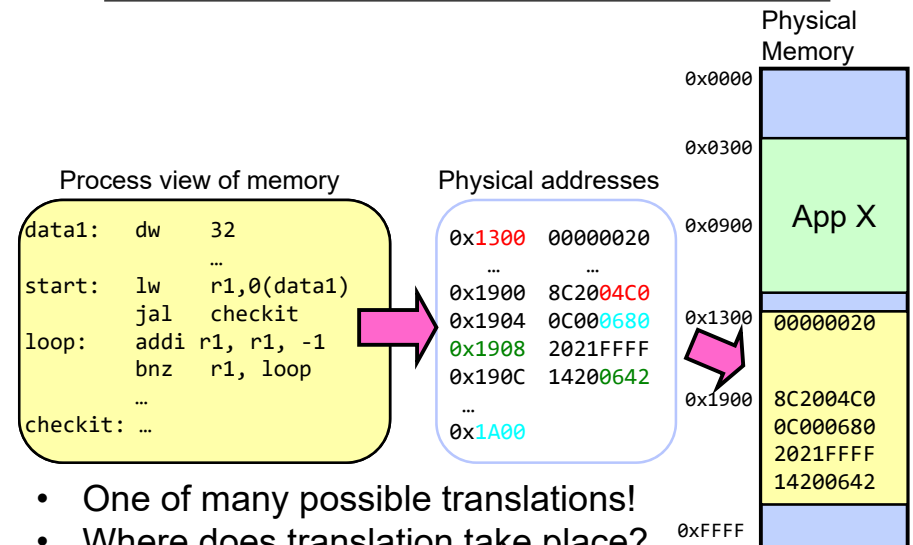
Need address translation!

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.21

Second copy of program from previous example



- One of many possible translations!
- Where does translation take place?
Compile time, Link/Load time, or Execution time?

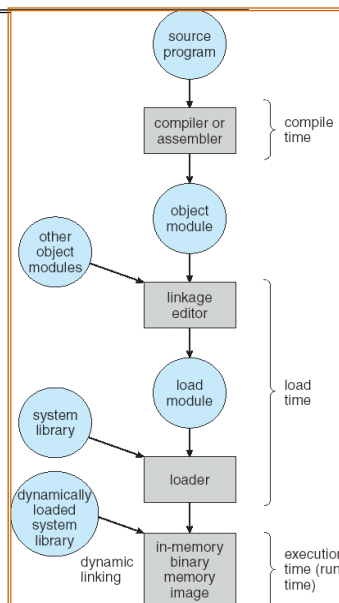
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.22

Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
 - Compile time (i.e., “gcc”)
 - Link/Load time (UNIX “ld” does link)
 - Execution time (e.g., dynamic libs)
- **Addresses can be bound to final values anywhere in this path**
 - Depends on hardware support
 - Also depends on operating system
- **Dynamic Libraries**
 - Linking postponed until execution
 - Small piece of code (i.e. the *stub*), locates appropriate memory-resident library routine
 - Stub replaces itself with the address of the routine, and executes routine



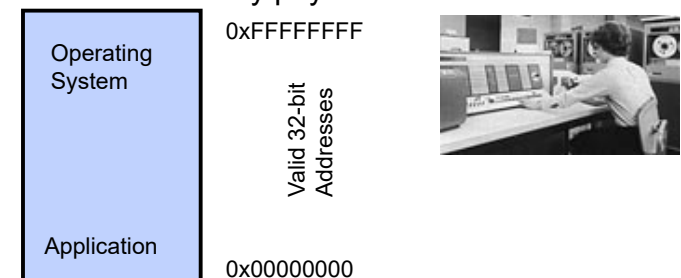
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.23

Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
 - Application always runs at same place in physical memory since only one application at a time
 - Application can access any physical address



- Application given illusion of dedicated machine by giving it reality of a dedicated machine

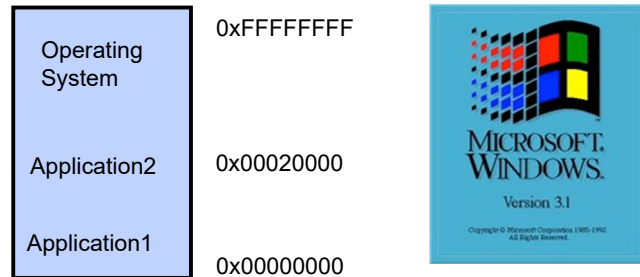
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.24

Multiprogramming (primitive stage)

- Multiprogramming without Translation or Protection
 - Must somehow prevent address overlap between threads



- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
 - Everything adjusted to memory location of program
 - Translation done by a linker-loader (relocation)
 - Common in early days (... till Windows 3.x, 95?)
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

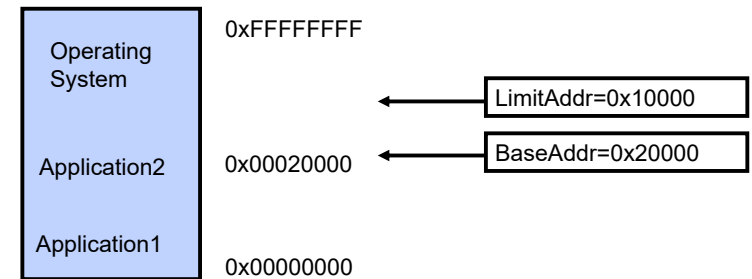
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.25

Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



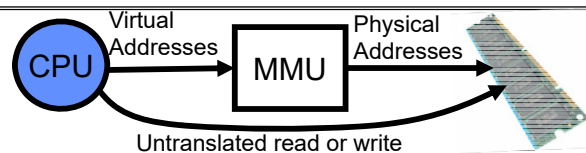
- Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
 - Cause error if user tries to access an illegal address
- During switch, kernel loads new base/limit from PCB (Process Control Block)
 - User not allowed to change base/limit registers

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.26

Recall: General Address translation



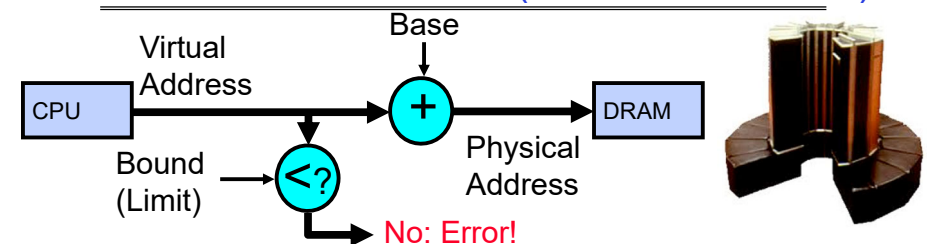
- Recall: Address Space:
 - All the addresses and state a process can touch
 - Each process and kernel has different address space
- Consequently, two views of memory:
 - View from the CPU (what program sees, virtual memory)
 - View from memory (physical memory)
 - Translation box (Memory Management Unit or MMU) converts between the two views
- Translation \Rightarrow much easier to implement protection!
 - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.27

Recall: Base and Bound (was from CRAY-1)



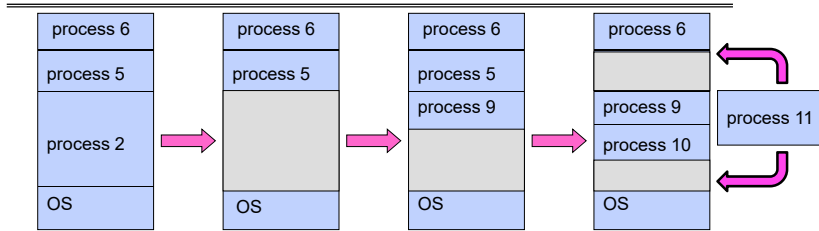
- Could use base/bounds for **dynamic address translation** – translation happens at execution:
 - Alter address of every load/store by adding “base”
 - Generate error if address bigger than limit
- Gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
 - Program gets continuous region of memory
 - Addresses within program do not have to be relocated when program placed in different region of DRAM

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.28

Issues with Simple B&B Method



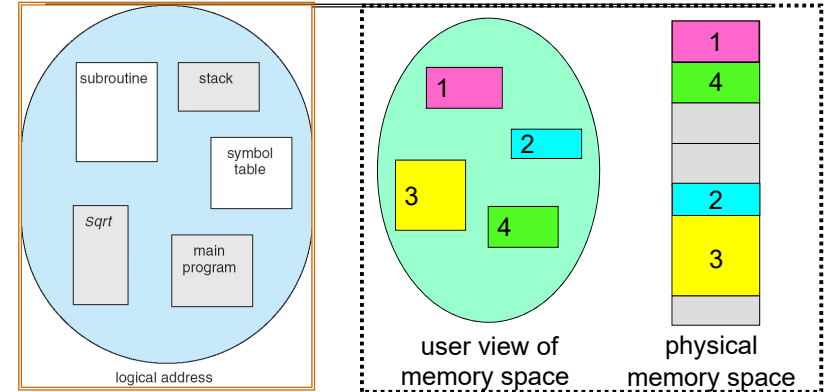
- Fragmentation problem over time
 - Not every process is same size \Rightarrow memory becomes fragmented over time
- Missing support for sparse address space
 - Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)
- Hard to do inter-process sharing
 - Want to share code segments when possible
 - Want to share memory between processes
 - Helped by providing multiple segments per process

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.29

More Flexible Segmentation



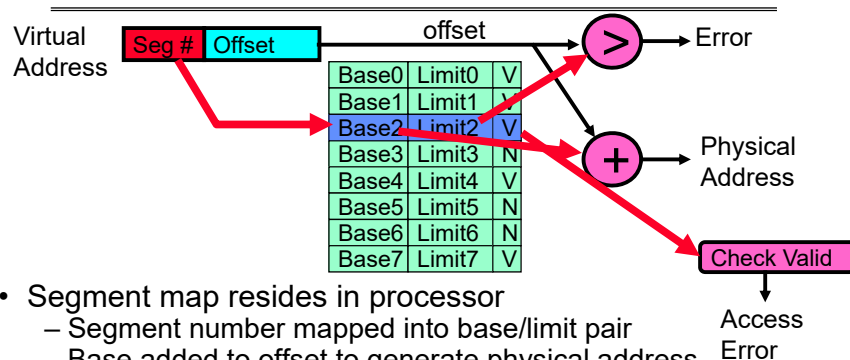
- Logical View: multiple separate segments
 - Typical: Code, Data, Stack
 - Others: memory sharing, etc
- Each segment is given region of contiguous memory
 - Has a base and limit
 - Can reside anywhere in physical memory

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.30

Implementation of Multi-Segment Model



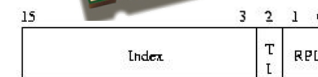
- Segment map resides in processor
 - Segment number mapped into base/limit pair
 - Base added to offset to generate physical address
 - Error check catches offset out of range
- As many chunks of physical memory as entries
 - Segment addressed by portion of virtual address
 - However, could be included in instruction instead:
 - x86 Example: `mov [es:bx], ax`.
- What is "V/N" (valid / not valid)?
 - Can mark segments as invalid; requires check as well

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.31

Intel x86 Special Registers



RPL = Requestor Privilege Level

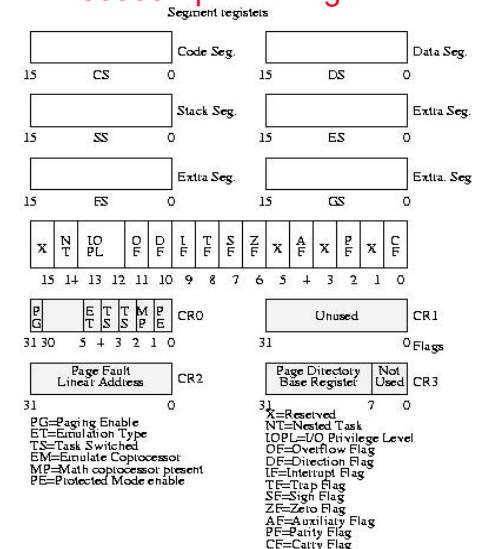
TI = Table Indicator

(0 = GDT, 1 = LDT)

Index = Index into table

Protected Mode segment selector

80386 Special Registers



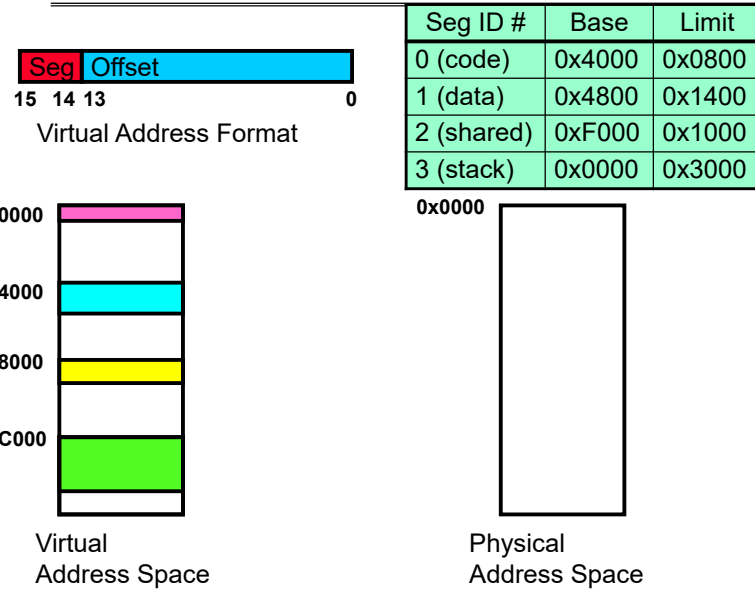
Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.32

Example: Four Segments (16 bit addresses)

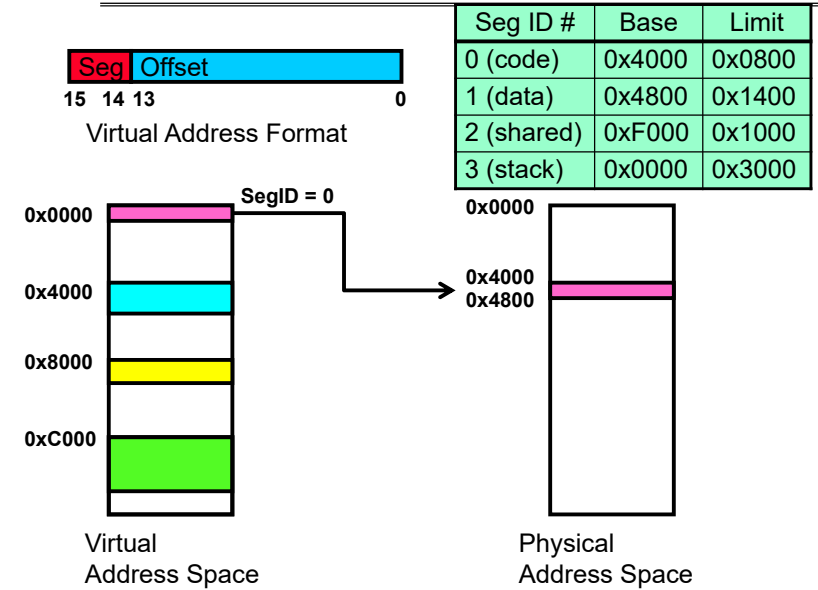


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.33

Example: Four Segments (16 bit addresses)

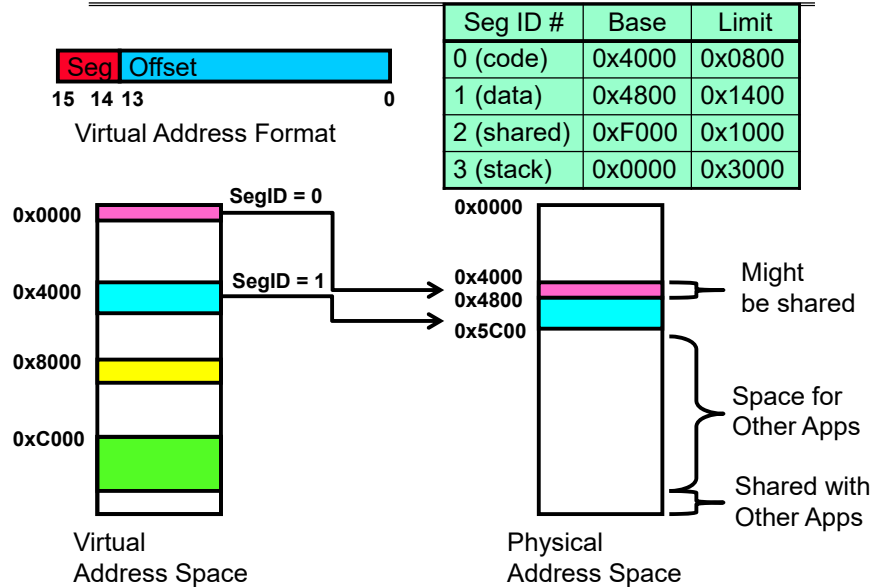


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.34

Example: Four Segments (16 bit addresses)

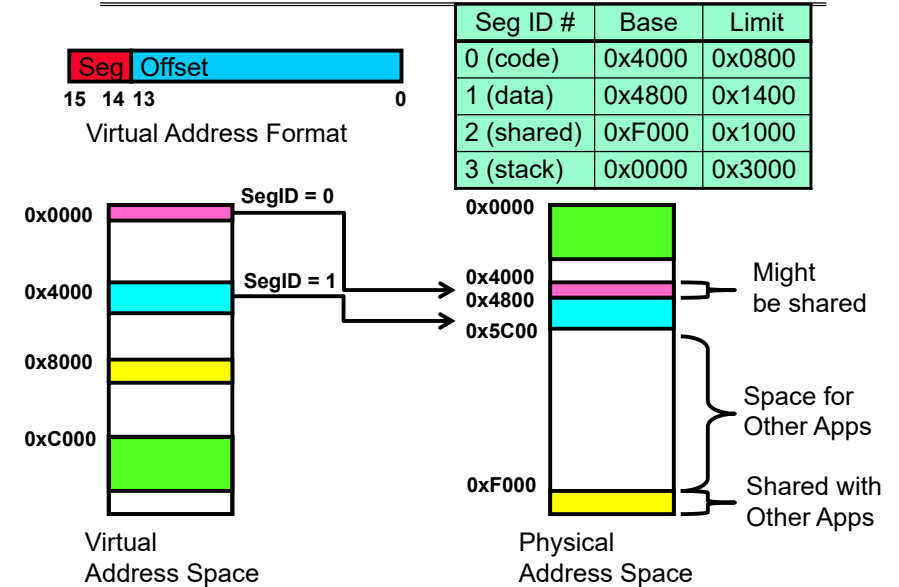


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.35

Example: Four Segments (16 bit addresses)



3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.36

Example of Segment Translation (16bit address)

0x240	main:	la \$a0, varx
0x244		jal strlen
...		...
0x360	strlen:	li \$v0, 0 ;count
0x364	loop:	lb \$t0, (\$a0)
0x368		beq \$r0,\$t0, done
...		...
0x4050	varx	dw 0x314159

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
Physical address? Base=0x4000, so physical addr=0x4240
Fetch instruction at 0x4240. Get "la \$a0, varx"
Move 0x4050 → \$a0, Move PC+4→PC

Example of Segment Translation (16bit address)

0x240	main:	la \$a0, varx
0x244		jal strlen
...		...
0x360	strlen:	li \$v0, 0 ;count
0x364	loop:	lb \$t0, (\$a0)
0x368		beq \$r0,\$t0, done
...		...
0x4050	varx	dw 0x314159

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
Physical address? Base=0x4000, so physical addr=0x4240
Fetch instruction at 0x4240. Get "la \$a0, varx"
Move 0x4050 → \$a0, Move PC+4→PC
2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC

Example of Segment Translation (16bit address)

0x240	main:	la \$a0, varx
0x244		jal strlen
...		...
0x360	strlen:	li \$v0, 0 ;count
0x364	loop:	lb \$t0, (\$a0)
0x368		beq \$r0,\$t0, done
...		...
0x4050	varx	dw 0x314159

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
Physical address? Base=0x4000, so physical addr=0x4240
Fetch instruction at 0x4240. Get "la \$a0, varx"
Move 0x4050 → \$a0, Move PC+4→PC
2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get "li \$v0, 0"
Move 0x0000 → \$v0, Move PC+4→PC

Example of Segment Translation (16bit address)

0x0240	main:	la \$a0, varx
0x0244		jal strlen
...		...
0x0360	strlen:	li \$v0, 0 ;count
0x0364	loop:	lb \$t0, (\$a0)
0x0368		beq \$r0,\$t0, done
...		...
0x4050	varx	dw 0x314159

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
Physical address? Base=0x4000, so physical addr=0x4240
Fetch instruction at 0x4240. Get "la \$a0, varx"
Move 0x4050 → \$a0, Move PC+4→PC
2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC
3. Fetch 0x0360. Translated to Physical=0x4360. Get "li \$v0, 0"
Move 0x0000 → \$v0, Move PC+4→PC
4. Fetch 0x0364. Translated to Physical=0x4364. Get "lb \$t0, (\$a0)"
Since \$a0 is 0x4050, try to load byte from 0x4050
Translate 0x4050 (0100 0000 0101 0000). Virtual segment #? 1; Offset? 0x50
Physical address? Base=0x4800, Physical addr = 0x4850,
Load Byte from 0x4850→\$t0, Move PC+4→PC

Observations about Segmentation

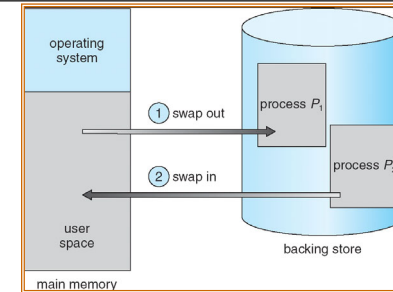
- Virtual address space has holes
 - Segmentation efficient for sparse address spaces
 - A correct program should never address gaps (except as mentioned in moment)
 - » If it does, trap to kernel and dump core
- When it is OK to address outside valid range?
 - This is how the stack and heap are allowed to grow
 - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
 - For example, code segment would be read-only
 - Data and stack would be read-write (stores allowed)
 - Shared segment could be read-only or read-write
- What must be saved/restored on context switch?
 - Segment table stored in CPU, not in memory (small)
 - Might store all of process' memory onto disk when switched (called "swapping")

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.41

What if not all segments fit into memory?



- Extreme form of Context Switch: Swapping
 - In order to make room for next process, some or all of the previous process is moved to disk
 - » Likely need to send out complete segments
 - This greatly increases the cost of context-switching
- What might be a desirable alternative?
 - Some way to keep only active portions of a process in memory at any one time
 - Need finer granularity control over physical memory

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.42

Problems with Segmentation

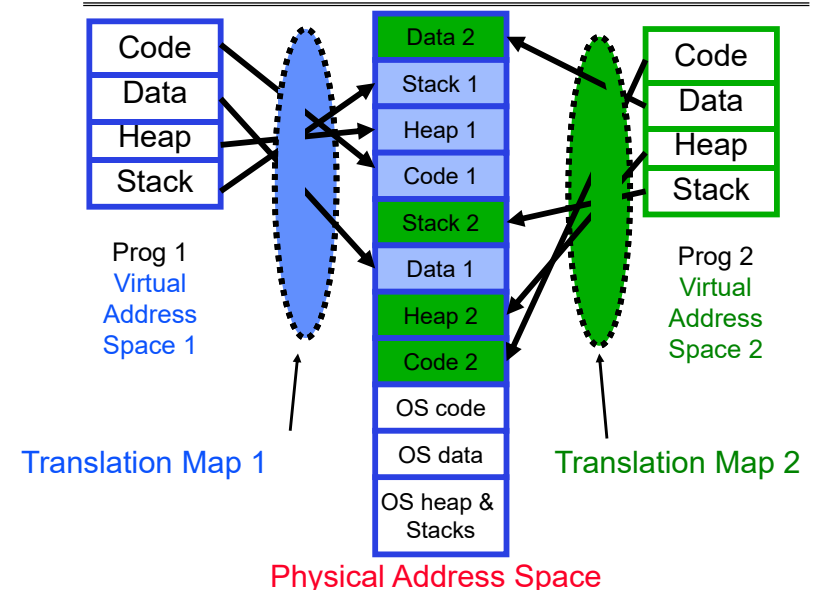
- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation**: wasted space
 - External: free gaps between allocated chunks
 - Internal: don't need all memory within allocated chunks

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.43

Recall: General Address Translation



3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.44

Paging: Physical Memory in Fixed Size Chunks

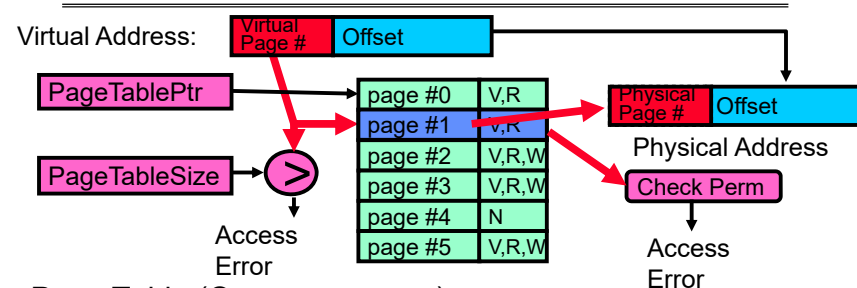
- Solution to fragmentation from segments?
 - Allocate physical memory in **fixed size** chunks (“pages”)
 - Every chunk of physical memory is equivalent
 - » Can use simple vector of bits to handle allocation: 00110001110001101 ... 110010
 - » Each bit represents page of physical memory
1 ⇒ allocated, 0 ⇒ free
- Should pages be as big as our previous segments?
 - No: Can lead to lots of internal fragmentation
 - » Typically have small pages (1K-16K)
 - Consequently: need multiple pages/segment

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.45

How to Implement Simple Paging?



- Page Table (One per process)
 - Resides in physical memory
 - Contains physical page and permission for each virtual page
 - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
 - Offset from Virtual address copied to Physical Address
 - » Example: 10 bit offset ⇒ 1024-byte pages
 - Virtual page # is all remaining bits
 - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
 - » Physical page # copied from table into physical address
 - Check Page Table bounds and permissions

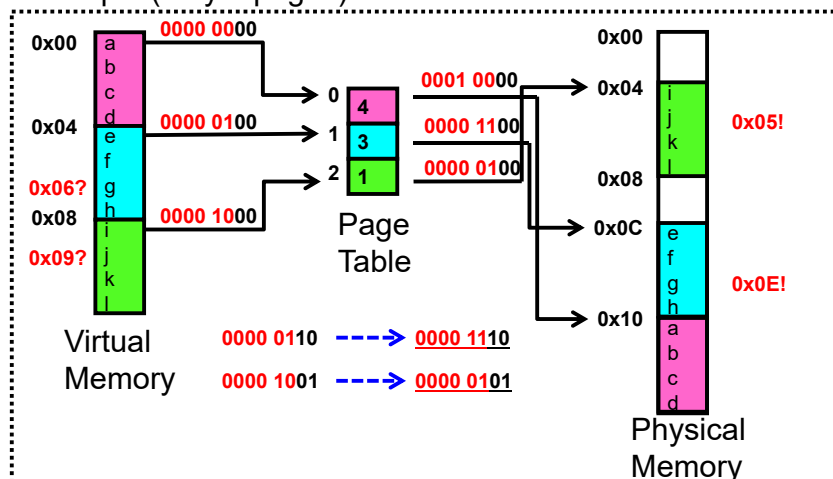
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.46

Simple Page Table Example

Example (4 byte pages)

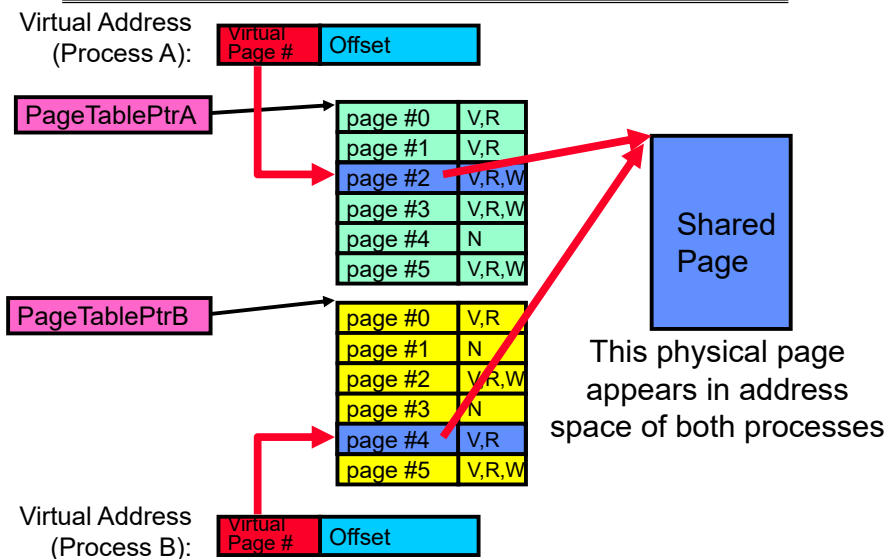


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.47

What about Sharing?



3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.48

Where is page sharing used ?

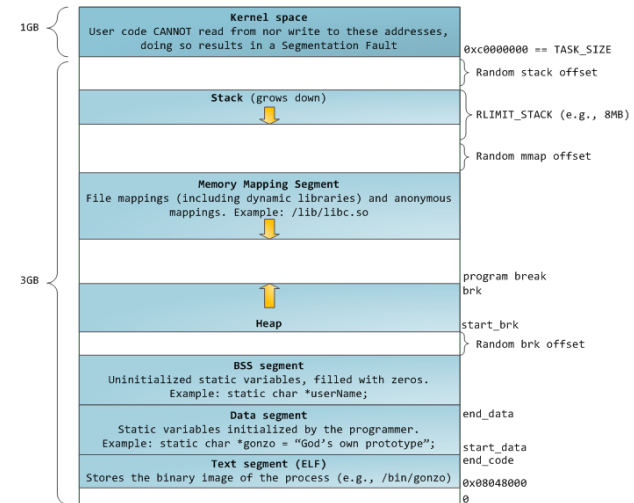
- The “kernel region” of every process has the same page table entries
 - The process cannot access it at user level
 - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
 - » What does the kernel need to do to access other user processes?
- Different processes running same binary!
 - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
 - Can actually share objects directly between processes
 - » Must map page into same place in address space!
 - This is a limited form of the sharing that threads have within a single process

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.49

Example: Memory Layout for Linux 32-bit (Pre-Meltdown patch!)



<http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png>

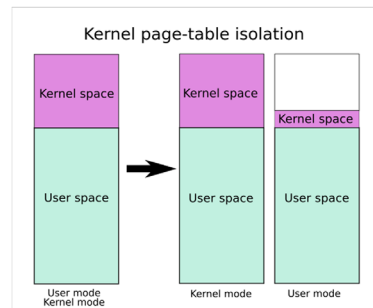
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.50

Some simple security measures

- Address Space Randomization
 - Position-Independent Code => can place user code region anywhere in the address space
 - » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
 - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
 - Don't map whole kernel space into each process, switch to kernel page table
 - Meltdown=>map none of kernel into user mode!

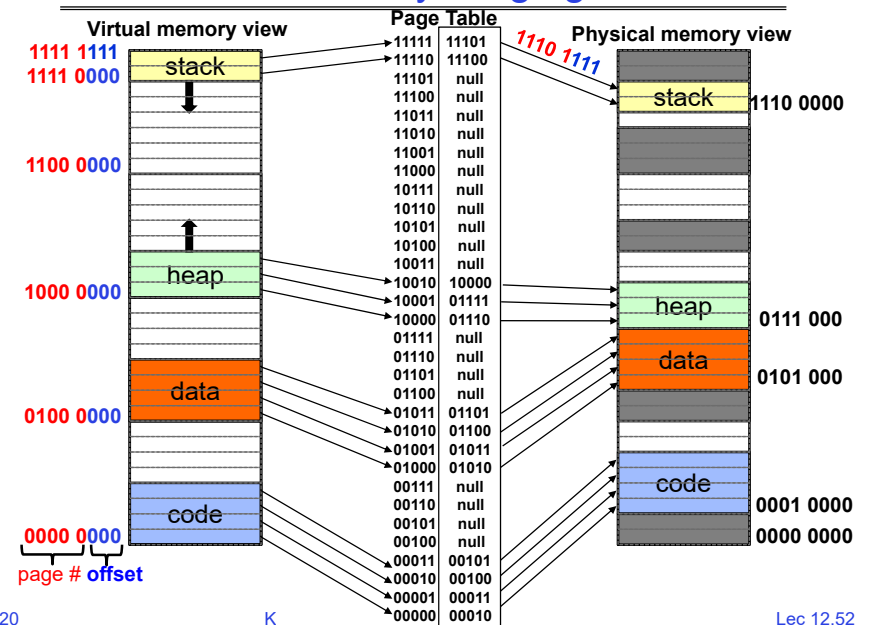


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.51

Summary: Paging

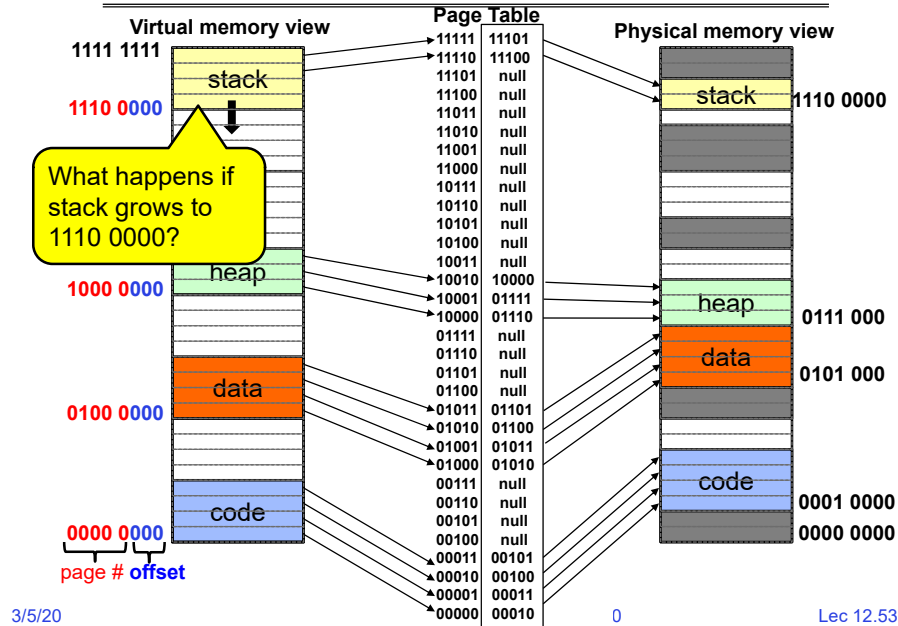


3/5/20

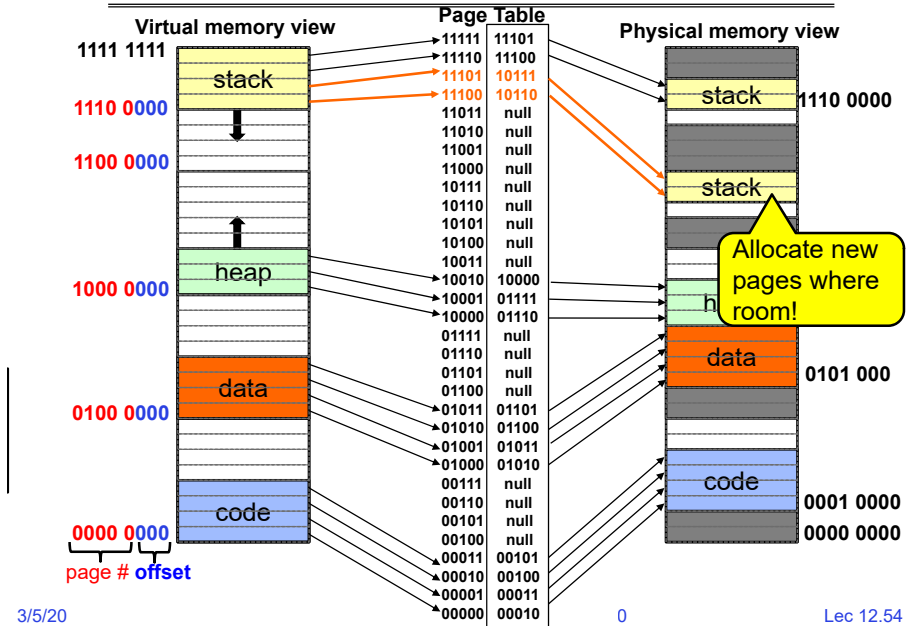
K

Lec 12.52

Summary: Paging



Summary: Paging



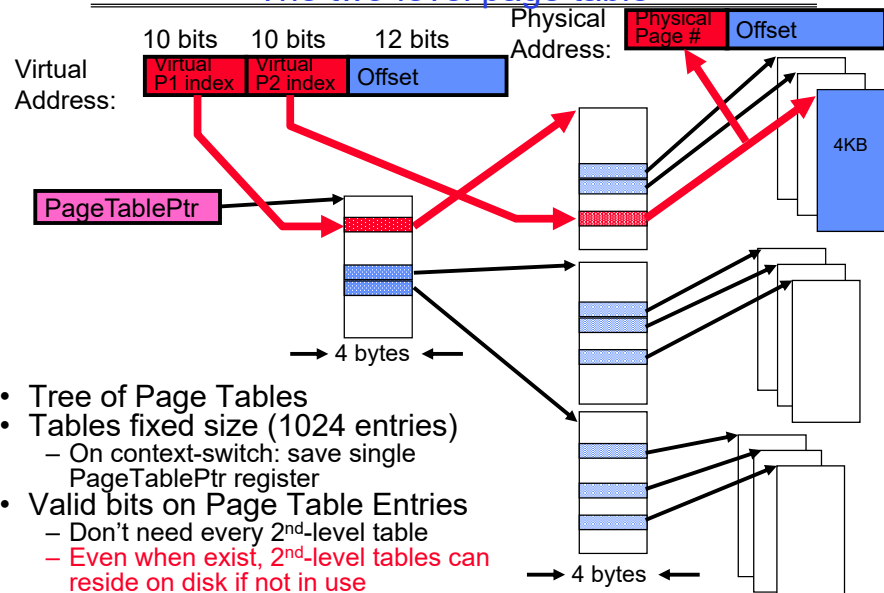
How big do things get?

- 32-bit address space $\Rightarrow 2^{32}$ bytes (4 GB)
 - Note: "b" = bit, and "B" = byte
 - And *for memory*:
 - "K"(kilo) = $2^{10} = 1024 \approx 10^3$ (But not quite!)
 - "M"(mega) = $2^{20} = (1024)^2 = 1,048,576 \approx 10^6$ (But not quite!)
 - "G"(giga) = $2^{30} = (1024)^3 = 1,073,741,824 \approx 10^9$ (But not quite!)
- Typical page size: 4 KB
 - how many bits of the address is that? (remember $2^{10} = 1024$)
 - Ans – 4KB = $4 \times 2^{10} = 2^{12} \Rightarrow 12$ bits of the address
- So how big is the simple page table for each process?
 - $2^{32}/2^{12} = 2^{20}$ (that's about a million entries) $\times 4$ bytes each \Rightarrow 4 MB
 - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory
- How big is a simple page table on a 64-bit processor (x86_64)?
 - $2^{64}/2^{12} = 2^{52}$ (that's 4.5×10^{15} or 4.5 exa-entries) $\times 8$ bytes each = 36×10^{15} bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
 - This is really a lot of space – for only the page table!!!
- Mostly, the address space is *sparse*, i.e. has holes in it that are not mapped to physical memory
 - So, most of this space is taken up by page tables mapped to nothing

Page Table Discussion

- What needs to be switched on a context switch?
 - Page table pointer and limit
- What provides protection here?
 - Translation (per process) and dual-mode!
 - Can't let process alter its own page table!
- Analysis
 - Pros
 - Simple memory allocation
 - Easy to share
 - Con: What if address space is sparse?
 - E.g., on UNIX, code starts at 0, stack starts at $(2^{31}-1)$
 - With 1K pages, need 2 million page table entries!
 - Con: What if table really big?
 - Not all pages used all the time \Rightarrow would be nice to have working set of page table in memory
- Simple Page table is way too big!
 - Does it all need to be in memory?
 - How about multi-level paging?
 - or combining paging and segmentation

Fix for sparse address space: The two-level page table

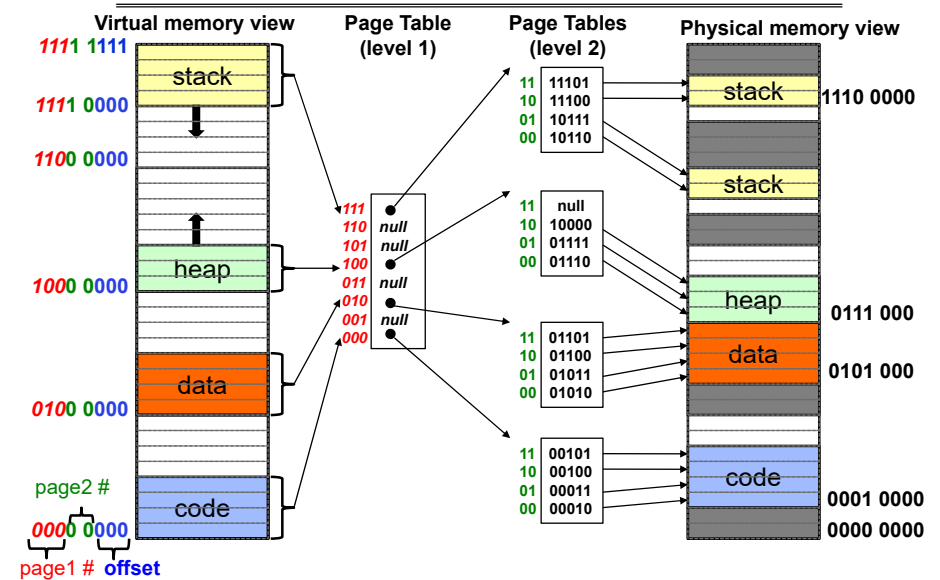


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.57

Summary: Two-Level Paging

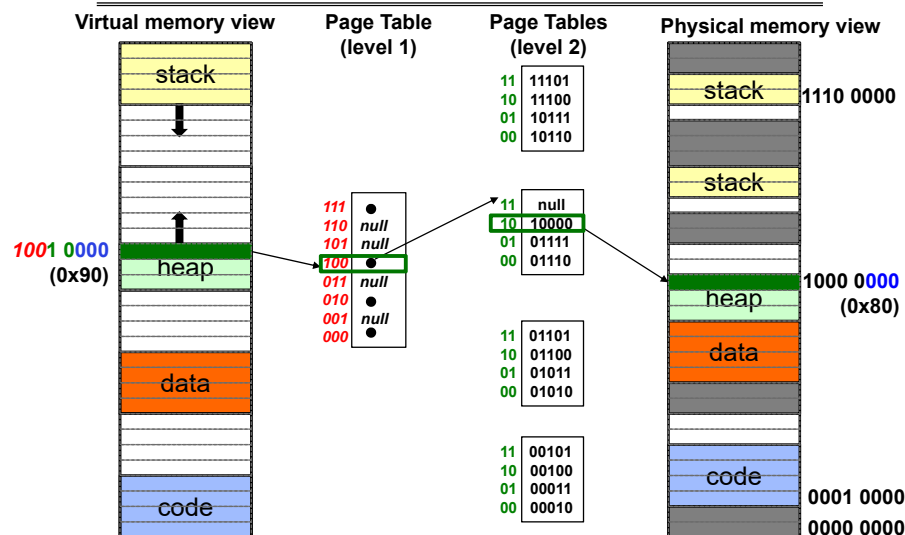


3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.58

Summary: Two-Level Paging



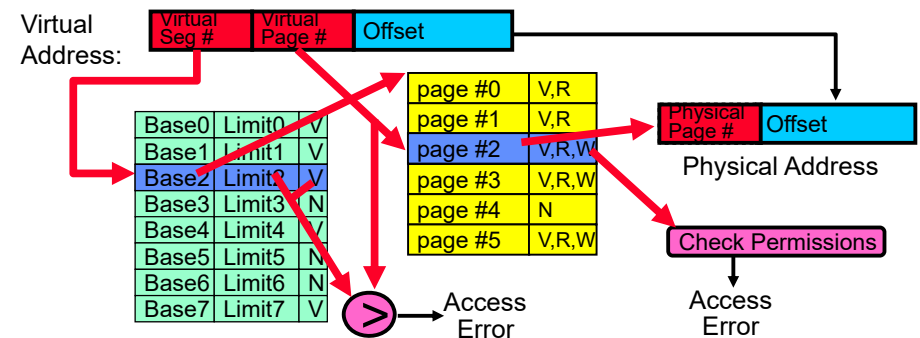
3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.59

Multi-level Translation: Segments + Pages

- What about a tree of tables?
 - Lowest level page table \Rightarrow memory still allocated with bitmap
 - Higher levels often segmented
- Could have any number of levels. Example (top segment):



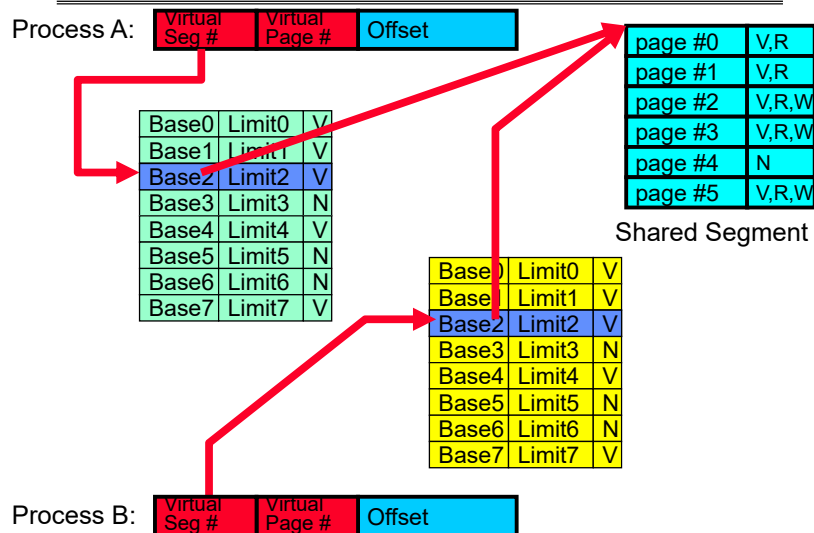
- What must be saved/restored on context switch?
 - Contents of top-level segment registers (for this example)
 - Pointer to top-level table (page table)

3/5/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 12.60

What about Sharing (Complete Segment)?



Multi-level Translation Analysis

- Pros:
 - Only need to allocate as many page table entries as we need for application
 - » In other words, sparse address spaces are easy
 - Easy memory allocation
 - Easy Sharing
 - » Share at segment or page level (need additional reference counting)
- Cons:
 - One pointer per page (typically 4K – 16K pages today)
 - Page tables need to be contiguous
 - » However, previous example keeps tables to exactly one page in size
 - Two (or more, if >2 levels) lookups per reference
 - » Seems very expensive!

Summary

- Segment Mapping
 - Segment registers within processor
 - Segment ID associated with each access
 - » Often comes from portion of virtual address
 - » Can come from bits in instruction instead (x86)
 - Each segment contains base and limit information
 - » Offset (rest of address) adjusted by adding base
- Page Tables
 - Memory divided into fixed-sized chunks of memory
 - Virtual page number from virtual address mapped through page table to physical page number
 - Offset of virtual address same as physical address
 - Large page tables can be placed into virtual memory
- Multi-Level Tables
 - Virtual address mapped to series of tables
 - Permit sparse population of address space