

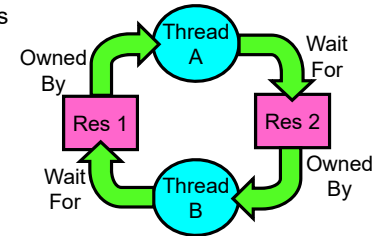
CS162
Operating Systems and
Systems Programming
Lecture 13

Memory 1: Address Translation and Virtual Memory

March 3rd, 2022
Prof. Anthony Joseph and John Kubiawicz
<http://cs162.eecs.Berkeley.edu>

Recall: Deadlock is A Deadly type of Starvation

- 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



3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.2

Recall: Four requirements for occurrence of 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

Recall: Techniques for Preventing Deadlock

- Make all threads request everything they'll need at the beginning.
 - Problem: Predicting future is hard, tend to over-estimate resources
 - Example:
 - » If need 2 chopsticks, request both at same time
 - » Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
 - Thus, preventing deadlock
 - Example $(x.Acquire(), y.Acquire(), z.Acquire(), \dots)$
 - » Make tasks request disk, then memory, then...
 - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.3

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.4

Request Resources Atomically (1)

Rather than:

Thread A:	Thread B:
x.Acquire();	y.Acquire();
y.Acquire();	x.Acquire();
...	...
y.Release();	x.Release();
x.Release();	y.Release();

Consider instead:

Thread A:	Thread B:
Acquire_both(x, y);	Acquire_both(y, x);
...	...
y.Release();	x.Release();
x.Release();	y.Release();

Request Resources Atomically (2)

Or consider this:

Thread A	Thread B
z.Acquire();	z.Acquire();
x.Acquire();	y.Acquire();
y.Acquire();	x.Acquire();
z.Release();	z.Release();
...	...
y.Release();	x.Release();
x.Release();	y.Release();

Acquire Resources in Consistent Order

Rather than:

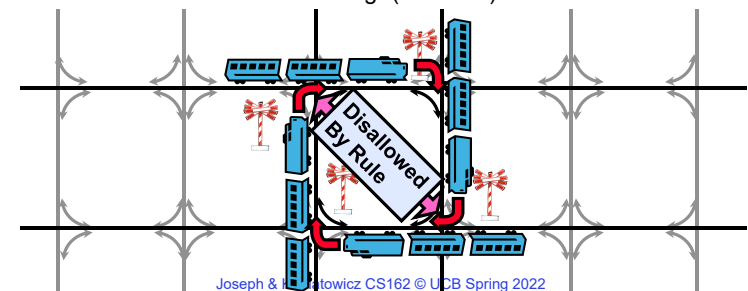
Thread A:	Thread B:
x.Acquire();	y.Acquire();
y.Acquire();	x.Acquire();
...	...
y.Release();	x.Release();
x.Release();	y.Release();

Consider instead:

Thread A:	Thread B:	
x.Acquire();	x.Acquire();	
y.Acquire();	y.Acquire();	
...	...	
y.Release();	x.Release();	
x.Release();	y.Release();	Does it matter in which order the locks are released?

Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
 - Each train wants to turn right
 - Blocked by other trains
 - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
 - Force ordering of channels (tracks)
 - » Protocol: Always go east-west first, then north-south
 - Called “dimension ordering” (X then Y)



Techniques for Recovering from Deadlock

- Terminate thread, force it to give up resources
 - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
 - Hold dining lawyer in contempt and take away in handcuffs
 - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
 - Take away resources from thread temporarily
 - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
 - Hit the rewind button on TiVo, pretend last few minutes never happened
 - For bridge example, make one car roll backwards (may require others behind him)
 - Common technique in databases (transactions)
 - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.9

Another view of virtual memory: Pre-empting Resources

Thread A:

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

Thread B:

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

- Before: With virtual memory we have “infinite” space so everything will just succeed, thus above example won't deadlock
 - Of course, it isn't actually infinite, but certainly larger than 2MB!
- Alternative view: we are “pre-empting” memory when paging out to disk, and giving it back when paging back in
 - This works because thread can't use memory when paged out

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.10

Techniques for Deadlock Avoidance

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

THIS DOES NOT WORK!!!!

- Example:

	<u>Thread A:</u>	<u>Thread B:</u>	
	x.Acquire();	y.Acquire();	
Blocks...	y.Acquire();	x.Acquire();	Wait?
	But it's already too late...
	y.Release();	x.Release();	
	x.Release();	y.Release();	

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.11

Deadlock Avoidance: Three States

- Safe state
 - System can delay resource acquisition to prevent deadlock
- Unsafe state
 - No deadlock yet...
 - But threads can request resources in a pattern that **unavoidably** leads to deadlock
- Deadlocked state
 - There exists a deadlock in the system
 - Also considered “unsafe”

Deadlock avoidance: prevent system from reaching an unsafe state

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.12

Deadlock Avoidance

- 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();
```

Wait until
Thread A
releases
mutex X

Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
 - State maximum (max) resource needs in advance
 - Allow particular thread to proceed if:

$$(\text{available resources} - \# \text{requested}) \geq \text{max remaining that might be needed by any thread}$$
- Banker's algorithm (less conservative):
 - Allocate resources dynamically
 - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
 - Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

$$([Max_{node}] - [Alloc_{node}] \leq [Avail]) \text{ for } ([Request_{node}] \leq [Avail])$$
 Grant request if result is deadlock free (conservative!)



Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
  done = true
  Foreach node in UNFINISHED {
    if ([Requestnode] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```



- » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

$$([Max_{node}] - [Alloc_{node}] \leq [Avail]) \text{ for } ([Request_{node}] \leq [Avail])$$
 Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
  done = true
  Foreach node in UNFINISHED {
    if ([Maxnode] - [Allocnode] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```



- » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

$$([Max_{node}] - [Alloc_{node}] \leq [Avail]) \text{ for } ([Request_{node}] \leq [Avail])$$
 Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
 - State maximum (max) resource needs in advance
 - Allow particular thread to proceed if:
 $(\text{available resources} - \# \text{requested}) \geq \text{max remaining that might be needed by any thread}$
- Banker's algorithm (less conservative):
 - Allocate resources dynamically
 - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
 - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
 $([\text{Max}_{\text{node}}] - [\text{Alloc}_{\text{node}}] \leq [\text{Avail}]) \text{ for } ([\text{Request}_{\text{node}}] \leq [\text{Avail}])$
 Grant request if result is deadlock free (conservative!)
 - Keeps system in a "SAFE" state: 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..



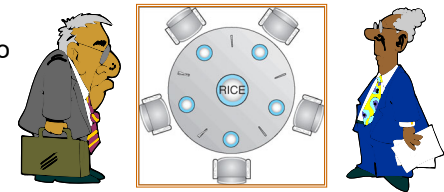
3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.17

Banker's Algorithm Example

- Banker's algorithm with dining lawyers
 - "Safe" (won't cause deadlock) if when try to grab chopstick either:
 - » Not last chopstick
 - » Is last chopstick but someone will have two afterwards



- What if k-handed lawyers? Don't allow if:
 - » It's the last one, no one would have k
 - » It's 2nd to last, and no one would have k-1
 - » It's 3rd to last, and no one would have k-2
 - » ...



3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.18

Deadlock Summary

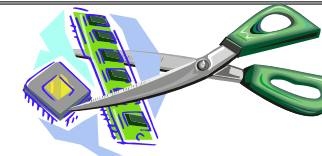
- Four conditions for deadlocks
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait
- Techniques for addressing Deadlock
 - Deadlock prevention:
 - » write your code in a way that it isn't prone to deadlock
 - Deadlock recovery:
 - » let deadlock happen, and then figure out how to recover from it
 - Deadlock avoidance:
 - » dynamically delay resource requests so deadlock doesn't happen
 - » Banker's Algorithm provides an algorithmic way to do this
 - Deadlock denial:
 - » ignore the possibility of deadlock

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.19

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)

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.20

Recall: Four Fundamental OS Concepts

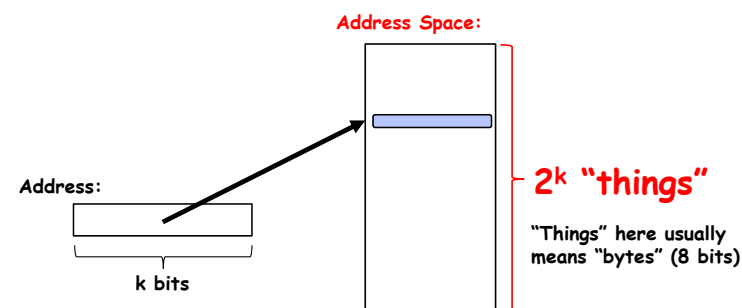
- **Thread: Execution Context**
 - Fully describes program state
 - Program Counter, Registers, Execution Flags, Stack
- **Address space (with or w/o translation)**
 - Set of memory addresses accessible to program (for read or write)
 - May be distinct from memory space of the physical machine (in which 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

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.21

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

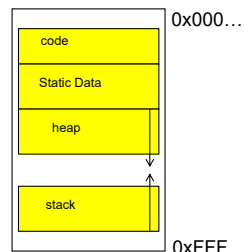
3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.22

Address Space, Process Virtual Address Space

- Definition: **Set of accessible addresses and the state associated with them**
 - $2^{32} = \sim 4 \text{ billion bytes}$ on a 32-bit machine
- How many 32-bit numbers fit in this address space?
 - 32-bits = 4 bytes, so $2^{32}/4 = 2^{30} = \sim 1 \text{ billion}$
- 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
 - ...

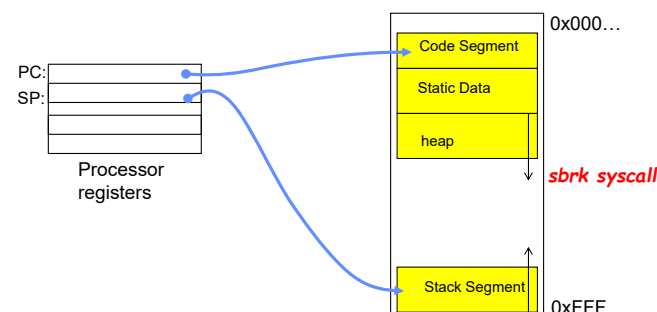


3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.23

Recall: Process Address Space: typical structure

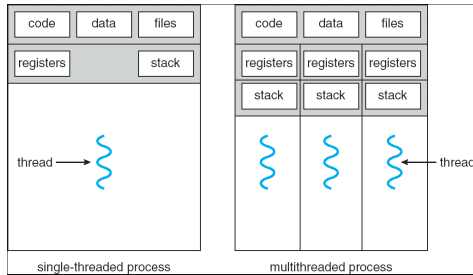


3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.24

Recall: Single and Multithreaded Processes



- **Threads** encapsulate concurrency
 - “Active” component
- **Address space** encapsulate protection:
 - “Passive” component
 - Keeps bugs from crashing the entire system
- Why have multiple threads per address space?

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.25

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
- **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
- **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)

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.26

Alternative View: Interposing on Process Behavior

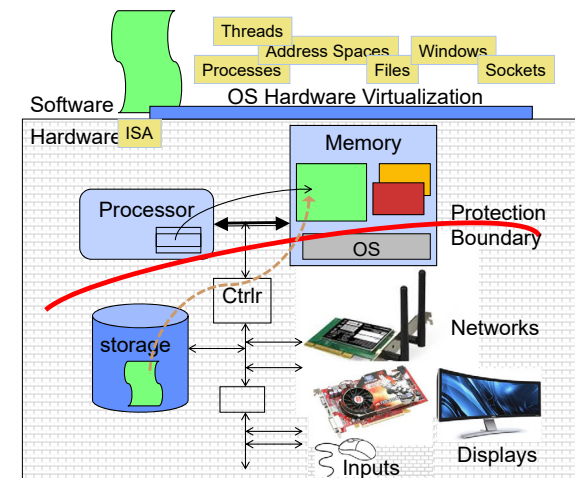
- OS interposes on process' I/O operations
 - How? All I/O happens via syscalls.
- OS interposes on process' CPU usage
 - How? Interrupt lets OS preempt current thread
- **Question: How can the OS interpose on process' memory accesses?**
 - Too slow for the OS to interpose *every* memory access
 - Translation: hardware support to accelerate the common case
 - Page fault: uncommon cases trap to the OS to handle

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.27

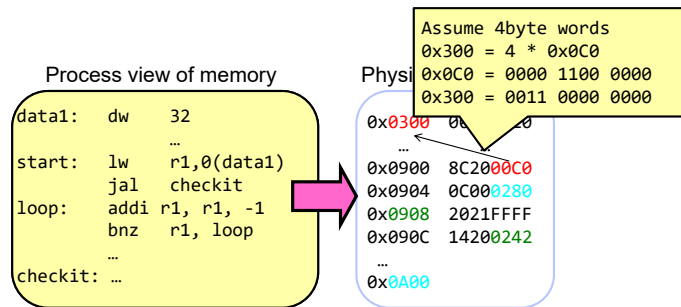
Recall: Loading



Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.28

Binding of Instructions and Data to Memory

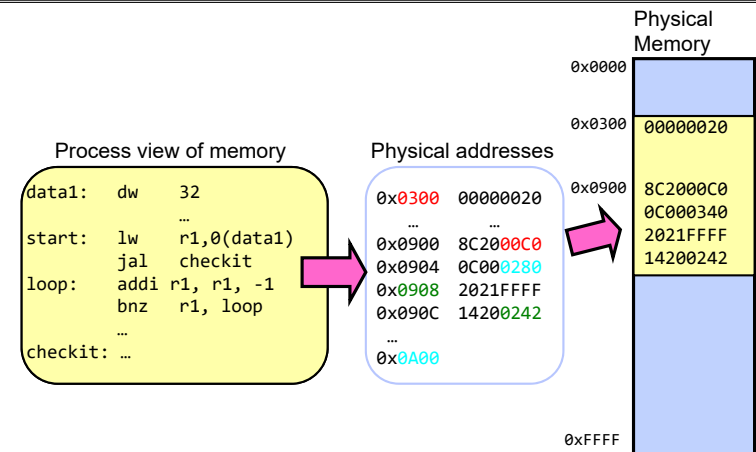


3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.29

Binding of Instructions and Data to Memory

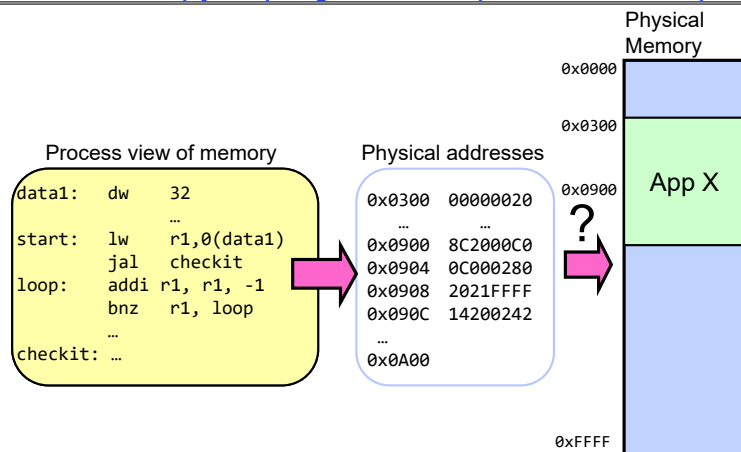


3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.30

Second copy of program from previous example



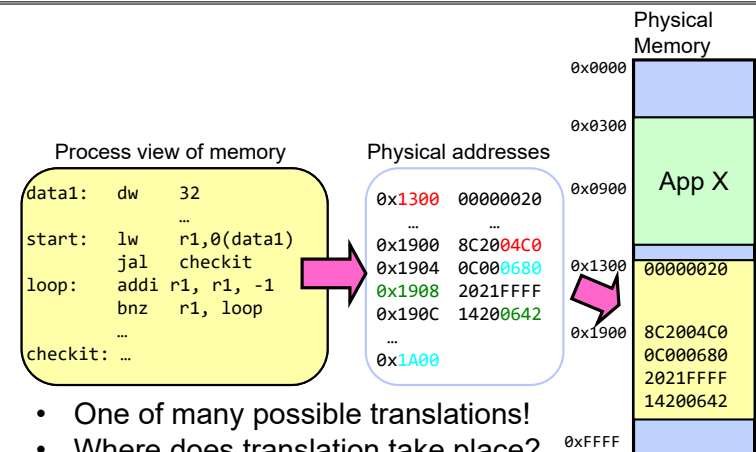
Need address translation!

3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.31

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?

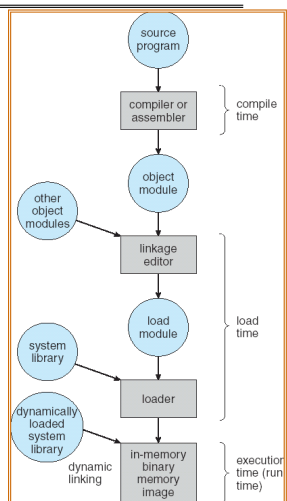
3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.32

From Program to Process

- 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



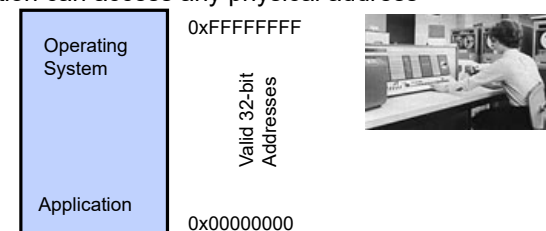
Lec 13.33

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

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



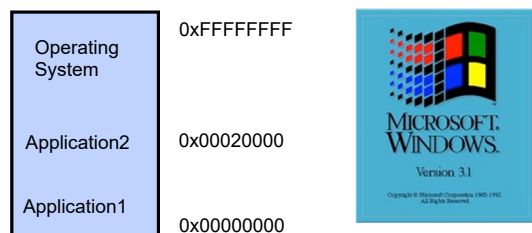
- Application given illusion of dedicated machine by giving it reality of a dedicated machine

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.34

Primitive Multiprogramming

- 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

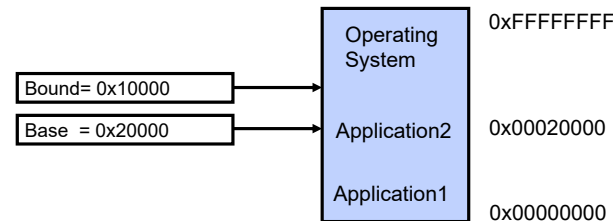
Lec 13.35

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Multiprogramming with Protection

- Can we protect programs from each other without translation?
 - Yes: **Base and Bound!**
 - Used by, e.g., Cray-1 supercomputer



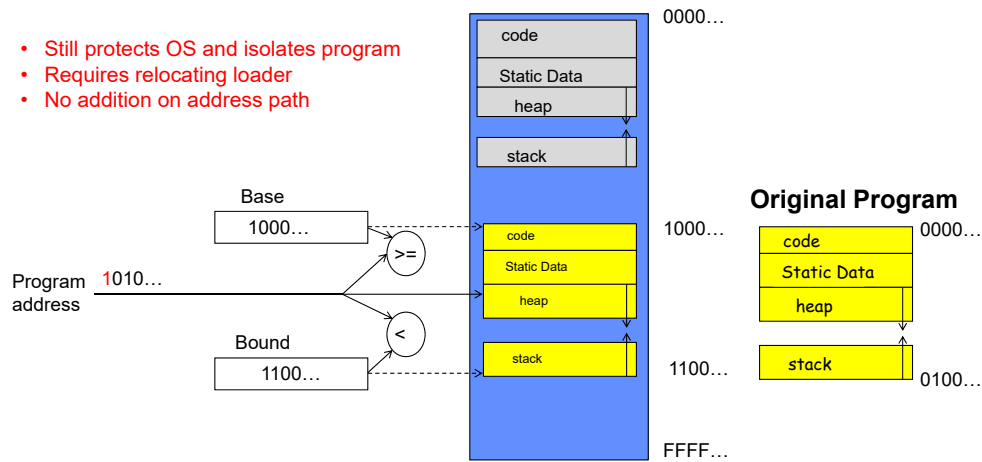
Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.36

3/3/22

Recall: Base and Bound (No Translation)

- Still protects OS and isolates program
- Requires relocating loader
- No addition on address path

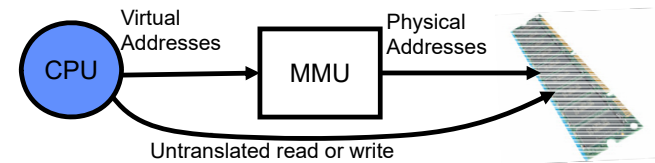


3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.37

Recall: General Address translation



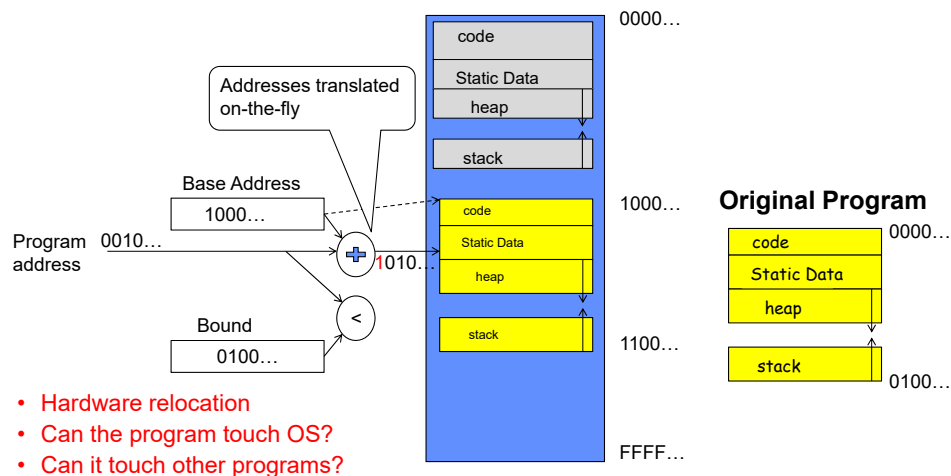
- Consequently, two views of memory:
 - View from the CPU (what program sees, virtual memory)
 - View from memory (physical memory)
- **Translation box** (Memory Management Unit or MMU) converts between the two views
- **Translation \Rightarrow much easier to implement protection!**
 - If task A cannot 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

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.38

Recall: Base and Bound (with Translation)



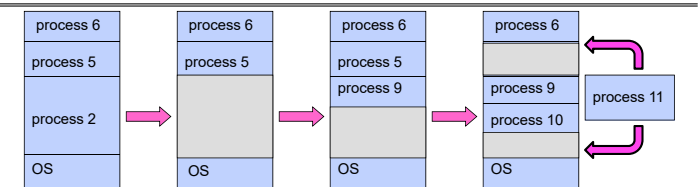
- **Hardware relocation**
- **Can the program touch OS?**
- **Can it touch other programs?**

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.39

Issues with Simple B&B Method



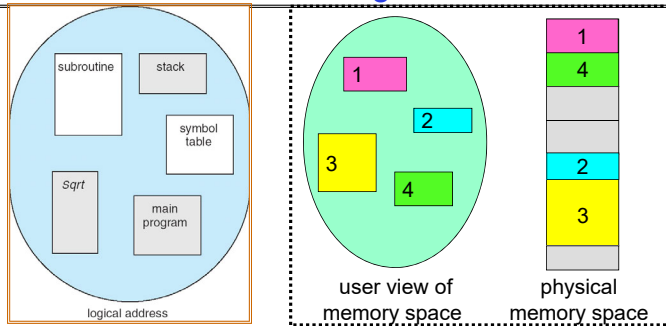
- **Fragmentation problem over time**
 - Not every process is same size \Rightarrow memory becomes fragmented over time
- **Missing support for sparse address space**
 - Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)
- **Hard to do inter-process sharing**
 - Want to share code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

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.40

More Flexible Segmentation

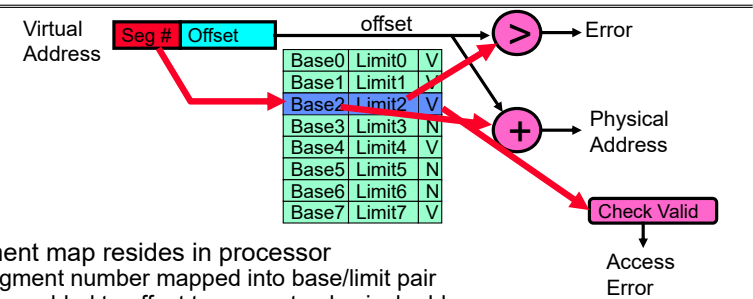


- Logical View: multiple separate segments
 - Typical: Code, Data, Stack
 - Others: memory sharing, etc
- Each segment is given region of contiguous memory
 - Has a base and limit
 - Can reside anywhere in physical memory

3/3/22 Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.41

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

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.42

Intel x86 Special Registers

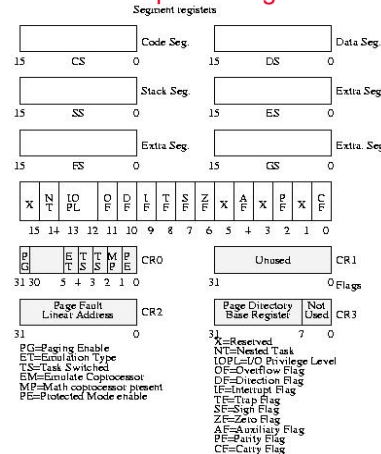


RPL = Requestor Privilege Level
 TL = Table Indicator
 (0 = GDT, 1 = LDT)
 Index = Index into table

Protected Mode segment selector



80386 Special Registers

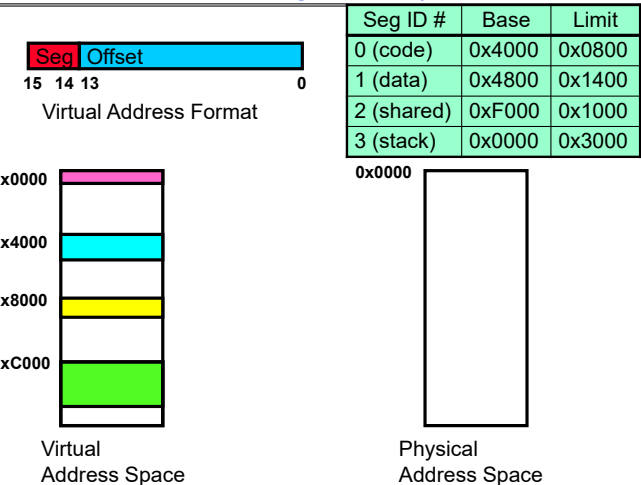


- Typical Segment Register
 - Current Priority is RPL of Code Segment (CS)
- Segmentation can't be just "turned off"
 - What if we just want to use paging?
 - Set base and bound to all of memory, in all segments

3/3/22 Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.43

Example: Four Segments (16 bit addresses)

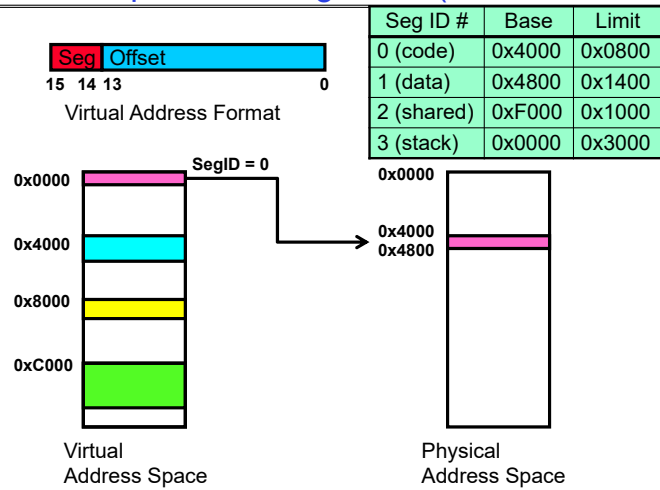


3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.44

Example: Four Segments (16 bit addresses)

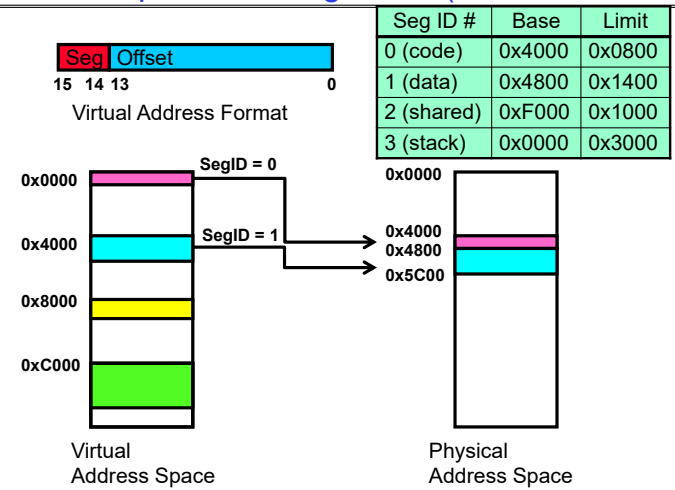


3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.45

Example: Four Segments (16 bit addresses)

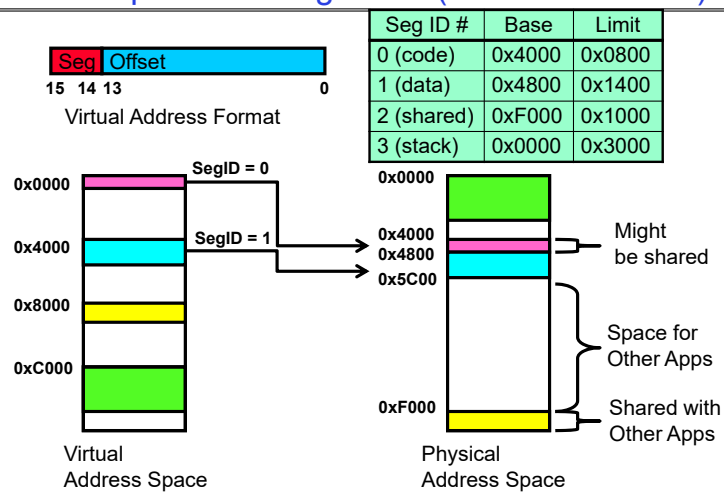


3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.46

Example: Four Segments (16 bit addresses)



3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.47

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):

- 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

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.48

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):

- 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
- 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):

- 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
- Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC
- 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):

- 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
- Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC
- Fetch 0x0360. Translated to Physical=0x4360. Get "li \$v0, 0"
Move 0x0000 → \$v0, Move PC+4→PC
- 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

- Translation on every instruction fetch, load or store
- Virtual address space has holes
 - Segmentation efficient for sparse address spaces
- When it is OK to address outside valid range?
 - This is how the stack (and heap?) 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)
- What must be saved/restored on context switch?
 - Segment table stored in CPU, not in memory (small)
 - Might store all of processes memory onto disk when switched (called "swapping")

Administrivia

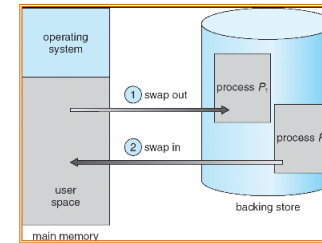
- Prof Joseph's office hours: Tuesdays 1-2pm and Thursdays 12-1 (room TBD)
- Homework 2 is due **TODAY** (Thursday 3/3)
- Midterm 2 conflict requests are due **TOMORROW** (Friday 3/4)

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.53

What if not all segments fit in memory?



- Extreme form of Context Switch: **Swapping**
 - 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

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.54

Problems with Segmentation

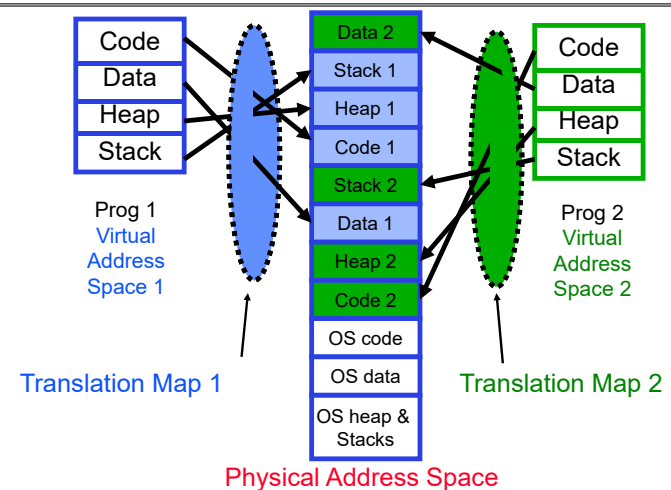
- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- **Fragmentation**: wasted space
 - **External**: free gaps between allocated chunks
 - **Internal**: don't need all memory within allocated chunks

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.55

Recall: General Address Translation



3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.56

Paging: Physical Memory in Fixed Size Chunks

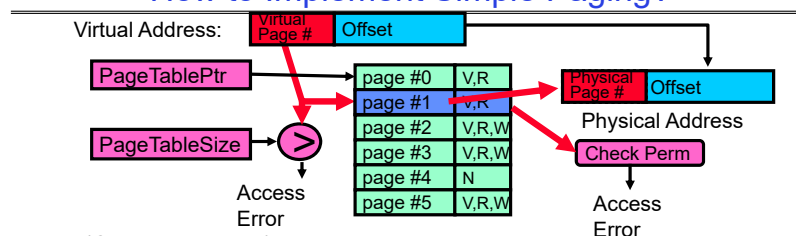
- Solution to fragmentation from segments?
 - Allocate physical memory in **fixed size** chunks (“pages”)
 - Every chunk of physical memory is equivalent
 - » Can use simple vector of bits to handle allocation: 00110001110001101 ... 110010
 - » Each bit represents page of physical memory
1 ⇒ allocated, 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

3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.57

How to Implement Simple Paging?



- Page Table (One per process)
 - Resides in physical memory
 - Contains physical page and permission for each virtual page (e.g. 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

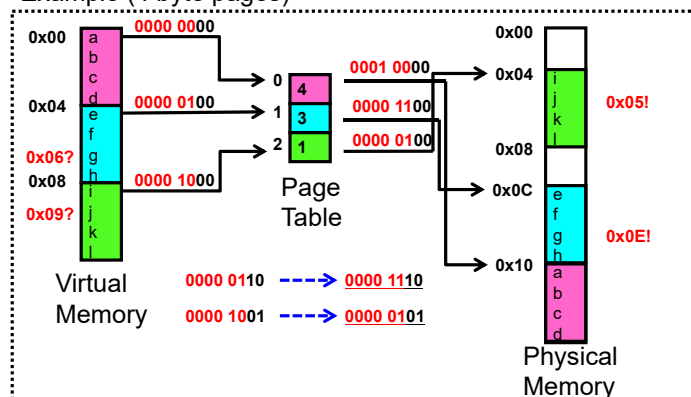
3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.58

Simple Page Table Example

Example (4 byte pages)

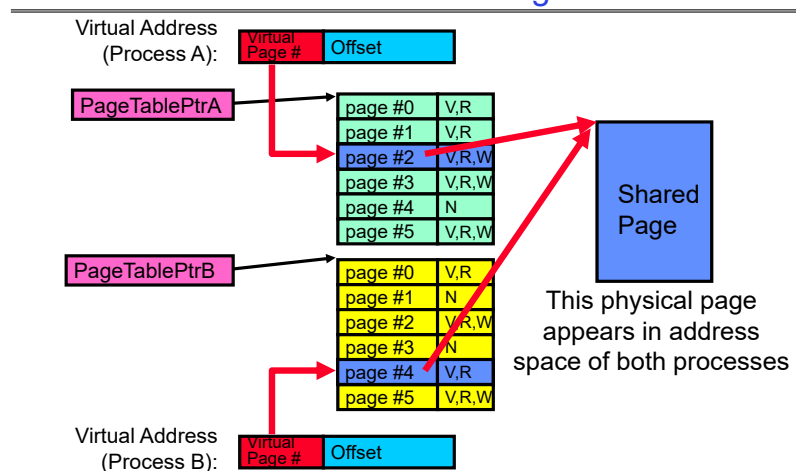


3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.59

What about Sharing?



3/3/22

Joseph & Kubiatowicz CS162 © UCB Spring 2022

Lec 13.60

Where is page sharing used ?

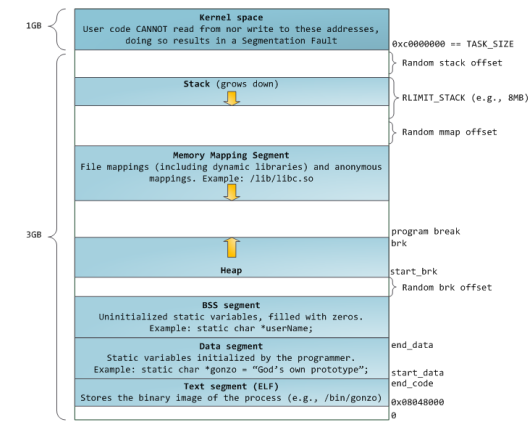
- The “kernel region” of every process has the same page table entries
 - The process cannot access it at user level
 - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
 - » What does the kernel need to do to access other user processes?
- Different processes running 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

3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.61

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



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

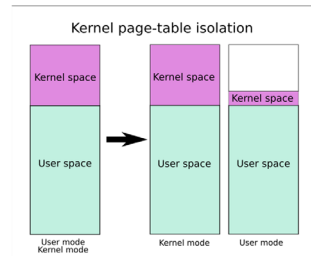
3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.62

Some simple security measures

- Address Space Randomization
 - Position-Independent Code \Rightarrow can place user code anywhere in 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 \Rightarrow map none of kernel into user mode!

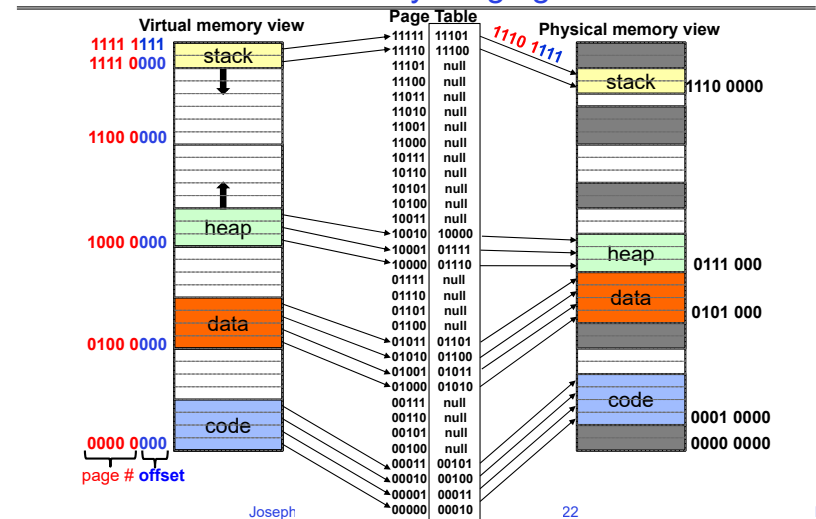


3/3/22

Joseph & Kubiawicz CS162 © UCB Spring 2022

Lec 13.63

Summary: Paging



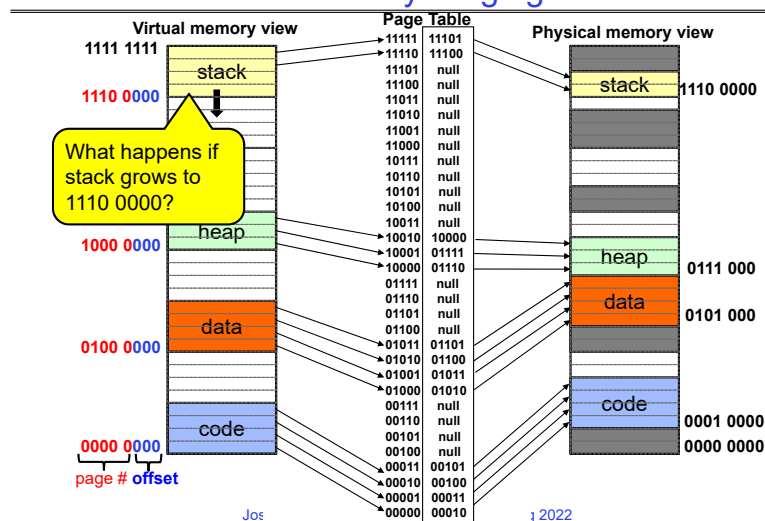
3/3/22

Joseph

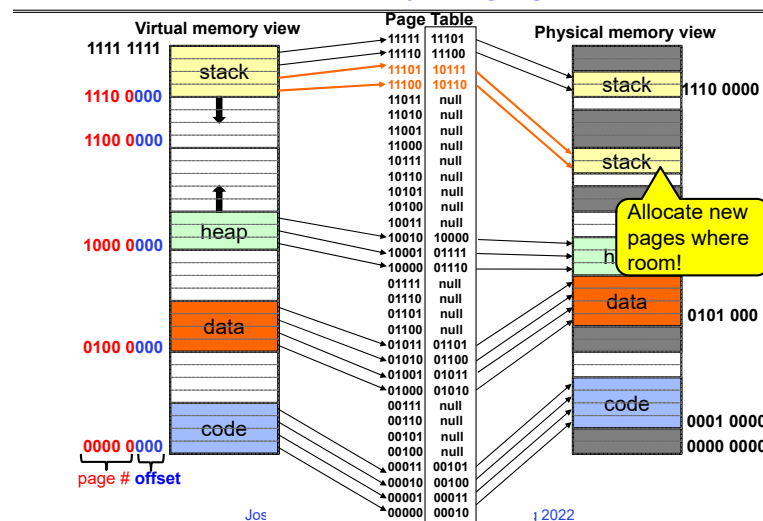
22

Lec 13.64

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!): Sometimes called "Ki" (Kibi)
 - "M"(mega) = $2^{20} = (1024)^2 = 1,048,576 \approx 10^6$ (But not quite!): Sometimes called "Mi" (Mibi)
 - "G"(giga) = $2^{30} = (1024)^3 = 1,073,741,824 \approx 10^9$ (But not quite!): Sometimes called "Gi" (Gibi)
- 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) x 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) x 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!!!
- The address space is sparse, i.e. has holes 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

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
- Next Time: Multi-Level Tables
 - Virtual address mapped to series of tables
 - Permit sparse population of address space