

# EE445M/EE360L.6

## Embedded and Real-Time Systems/ Real-Time Operating Systems

### Lecture 12: Memory Protection, Virtual Memory, Paging

References: T. Anderson, M. Dahlin, "Operating Systems: Principles and Practice"  
R. & A. Arpaci-Dusseau, "Operating Systems: Three Easy Pieces", <http://ostep.org>

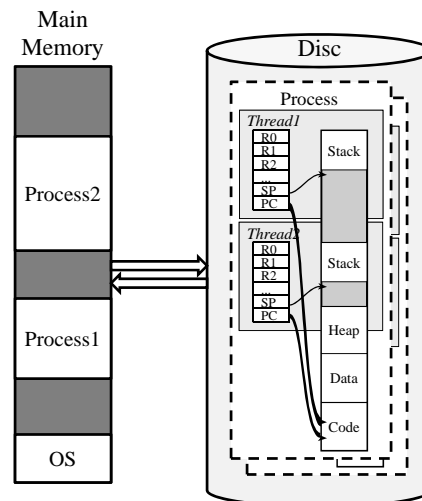
Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

1

## Recap: Memory Management

- Sharing
  - Per-thread: stack
  - Per-program/-process: heap, code, data
- Allocation
  - Static, permanent
    - Globals, OS code
  - Dynamic, temporary
    - Stack, heap, process swapping
- Protection
  - Access control



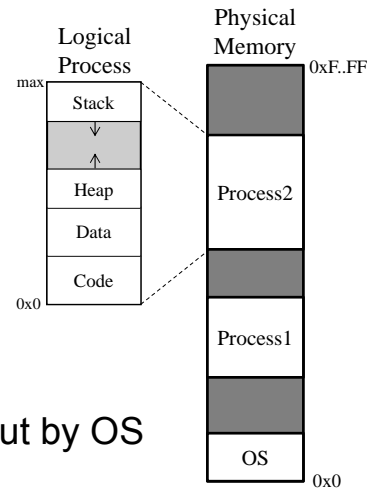
Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

2

## Recap: Processes

- OS manages processes
  - CPU scheduling
  - Code/data memory
- Independent programs
  - Separately compiled
  - Virtual address space
- Brought in/out of memory
  - On load/exit, swapped in/out by OS
  - Address translation



Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

3

## Memory Protection

- Divide memory into regions
  - Allocated heap blocks
  - Thread/process data & code segments
- Define access control per region
  - Read-only/read-write
  - Execute/no-execute
- Enforce access control in hardware
  - On every memory access (load/store)
  - Permission fault exception

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

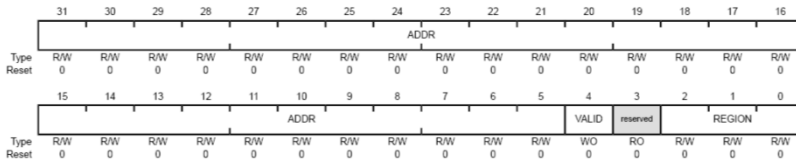
4

# TM4C123 Memory Protection

- Memory Protection Unit (MPU)
  - 8 separate memory regions

### MPU Region Base Address (MPUBASE)

Base 0xE000.E000  
 Offset 0xD9C  
 Type R/W, reset 0x0000.0000



Bit/Field	Name	Type	Reset	Description
31:5	ADDR	R/W	0x0000.000	Base Address Mask Bits 31:N in this field contain the region base address. The value of N depends on the region size, as shown above. The remaining bits (N-1):5 are reserved. Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.

Lecture 12

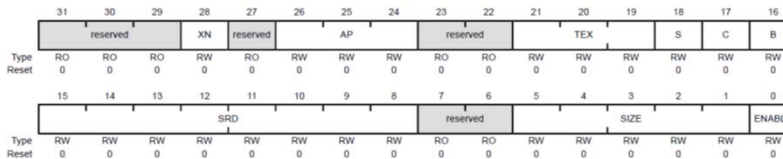
J. Valvano, A. Gerstlauer  
 EE445M/EE380L.6

5

# Memory Region Attributes

### MPU Region Attribute and Size (MPUATTR)

Base 0xE000.E000  
 Offset 0xDA0  
 Type RW, reset 0x0000.0000



28	XN	R/W	0	Instruction Access Disable
				Value Description
				0 Instruction fetches are enabled.
				1 Instruction fetches are disabled.
26:24	AP	R/W	0	Access Privilege
				For information on using this bit field, see Table 3-5 on page 101.

Table 3-10. Example SIZE Field Values

SIZE Encoding	Region Size	Value of N <sup>a</sup>	Note
00100b (0x4)	32 B	5	Minimum permitted size
01001b (0x9)	1 KB	10	-
10011b (0x13)	1 MB	20	-
11101b (0x1D)	1 GB	30	-
11111b (0x1F)	4 GB		No valid ADDR field in MPUBASE; the region occupies the complete memory map.

Lecture 12

a. Refers to the N parameter in the MPUBASE register (see page 195).

6

# Access Privileges

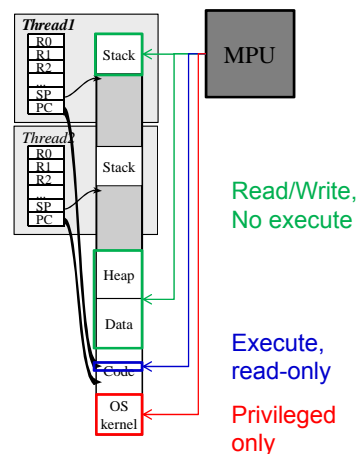
Table 3-5. AP Bit Field Encoding

AP Bit Field	Privileged Permissions	Unprivileged Permissions	Description
000	No access	No access	All accesses generate a permission fault.
001	R/W	No access	Access from privileged software only.
010	R/W	RO	Writes by unprivileged software generate a permission fault.
011	R/W	R/W	Full access.
100	Unpredictable	Unpredictable	Reserved.
101	RO	No access	Reads by privileged software only.
110	RO	RO	Read-only, by privileged or unprivileged software.
111	RO	RO	Read-only, by privileged or unprivileged software.

- Memory management fault on violation
  - Can be caught by OS in an interrupt handler

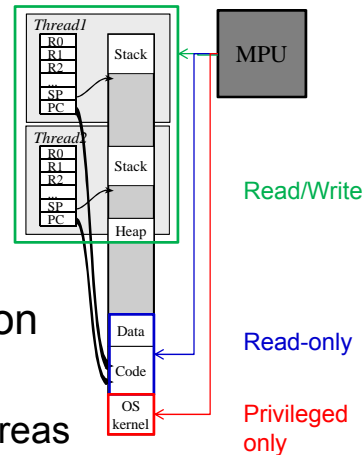
# Thread-Protected Mode

- Only current thread has memory access
  - Code
  - Data/heap, stack
  - OS kernel traps
- On context switch
  - Re-assign MPU permissions
  - Extra overhead



## μCOS Thread Groups

- Group of threads protected jointly
  - Called “process” in μCOS-II
  - Group-local shared memory region
- Inter-group communication
  - Through OS kernel
  - Special shared memory areas



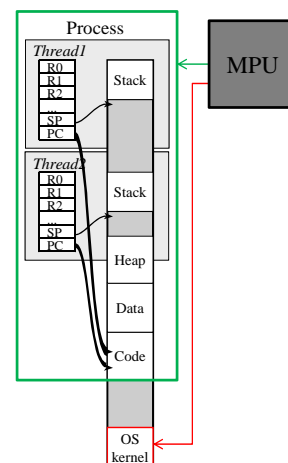
Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

9

## Multi-Programming

- Process
  - Whole program in execution
  - Code, data, heap
  - One or more threads
- Multi-processing
  - Multiple processes/programs in main memory
  - OS schedules processes & threads
  - Process-level protection

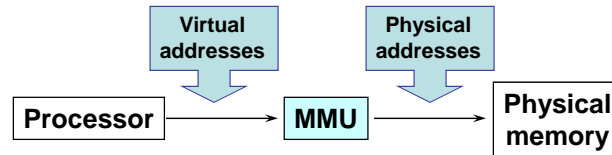


Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

10

## Memory Management Unit (MMU)



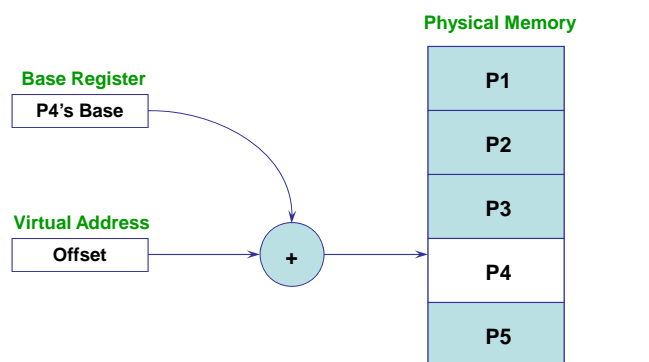
- Fast & efficient address translation
  - Hardware supported, at run-time
  - Start with old, simple ways
  - Progress to current techniques

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

11

## Fixed Partitions



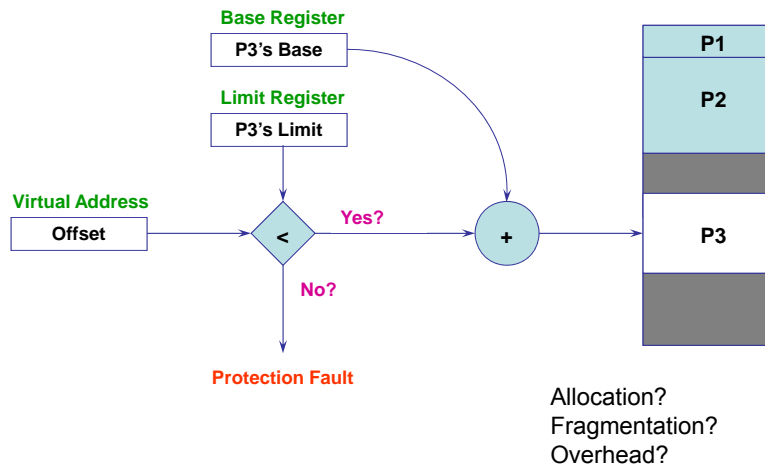
Allocation?  
Fragmentation?  
Overhead?

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

12

## Variable Partitions



Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

13

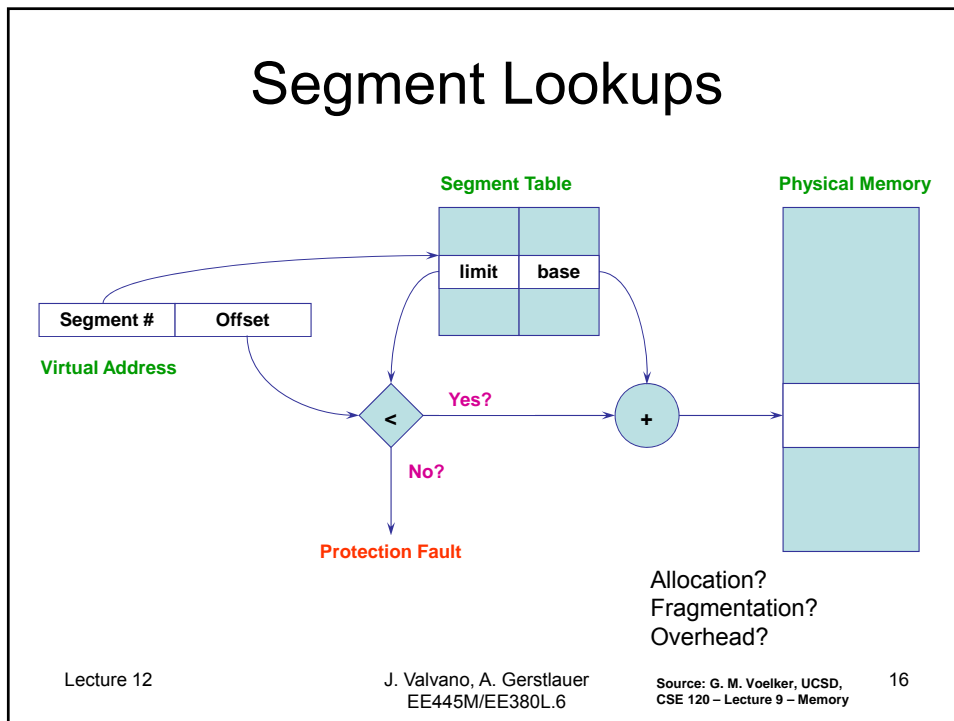
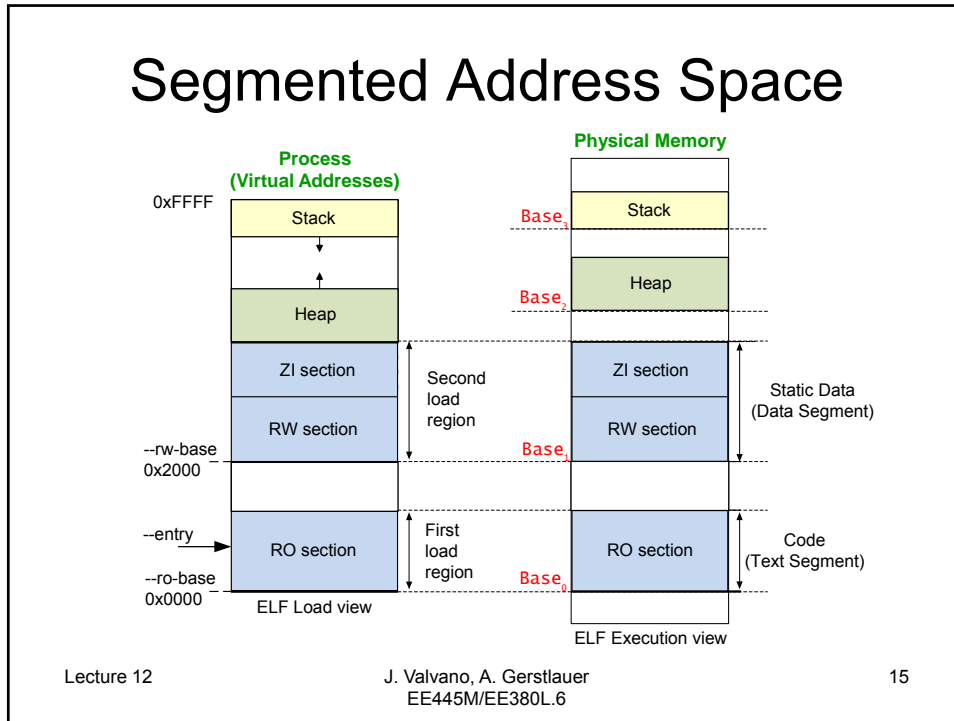
## Segmentation

- Partition memory into logically related units
  - Module, procedure, stack, data, file, etc.
  - Virtual addresses become <segment #, offset>
  - **Units of memory from programmer's perspective**
- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

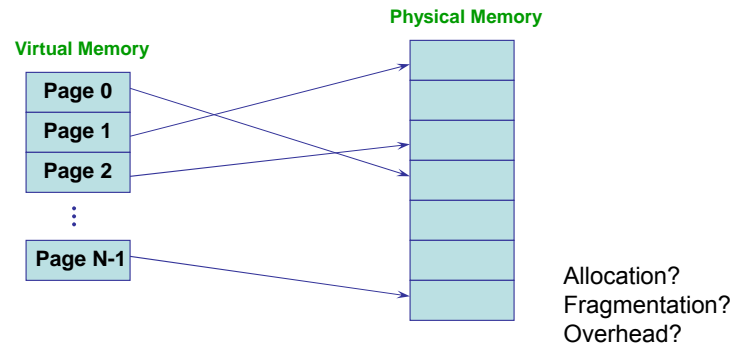
14





## Paging

- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory



Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

17

## Virtual Memory

- Programmers (and processes) view memory as one contiguous address space
  - From 0 through N
  - Virtual address space (VAS)
- In reality, pages are scattered throughout physical storage
- The mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
  - The address “0x1000” maps to different physical addresses in different processes

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

18

## Demand Paging / Swapping

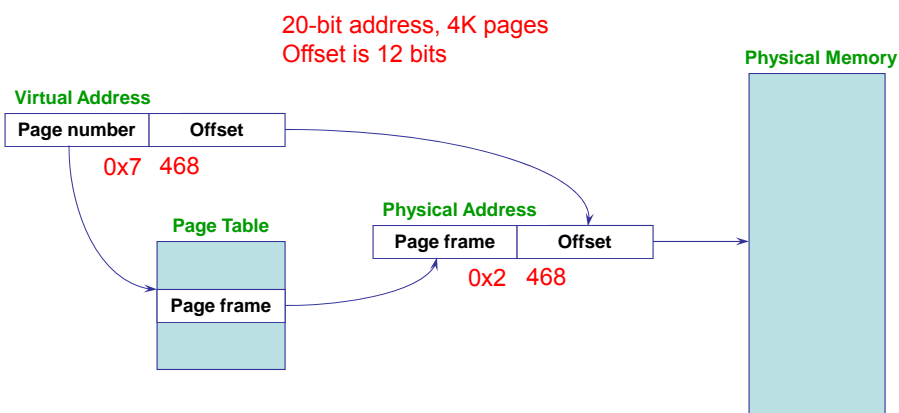
- Pages can be moved between memory and disk
  - Use disk to provide more virtual than physical memory
- OS uses main memory as a page cache of all the data allocated by processes in the system
  - Initially, pages are allocated from memory
  - When memory fills up, allocating a page in memory requires some other page to be evicted from memory
    - Why physical memory pages are called “frames”
  - Evicted pages go to disk
    - Where? The swap file/backing store
  - The movement of pages between memory and disk is done by the OS, and is transparent to the application
    - But: expensive!

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

19

## Page Lookups



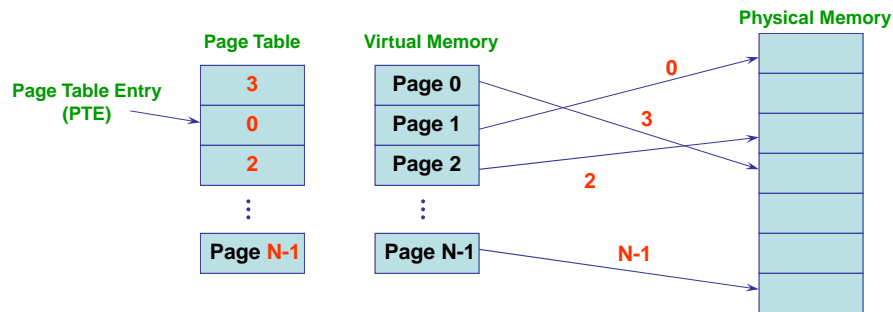
Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

20

## Page Tables

- Page tables completely define the mapping between virtual pages and physical pages for an address space
- Each process has address space & page table
- Page tables are data structures maintained in the OS
- Accessible by MMU for hardware translation



Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

21

## Page Table Size

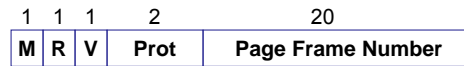
- How big does the page table need to be?
  - 20-bit address
    - 12 bit page offset (4 kibibyte page size)
    - 8 bit page number
  - 256 pages, each entry is 12 bits
  - 32-bit address
    - 14 bit page offset (16 kibibyte page size)
    - 18 bit page number
  - $2^{18}$  pages, each entry is 14 bits

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

22

## Page Table Entries (PTEs)



- Page table entries control mapping
  - The **Modify** bit says whether or not the page has been written
    - It is set when a write to the page occurs
  - The **Reference** bit says whether the page has been accessed
    - It is set when a read or write to the page occurs
  - The **Valid** bit says whether or not the PTE can be used
    - It is checked each time the virtual address is used, set when page is in memory
  - The **Protection** bits say what operations are allowed on page
    - Read, write, execute
  - The **page frame number** (PFN) determines physical page

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

23

## Segmentation and Paging

- Can combine segmentation and paging
  - The x86 supports segments and paging
- Use segments to manage logically related units
  - Module, procedure, stack, file, data, etc.
  - Segments vary in size, but usually large (>1 page)
- Pages to partition segments into fixed size chunks
  - Segments easier to manage within physical memory
    - Segments become “pageable” – rather than moving segments into and out of memory, just move page portions of segment
  - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated
- Tends to be complex...

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

24

## Paging Limitations

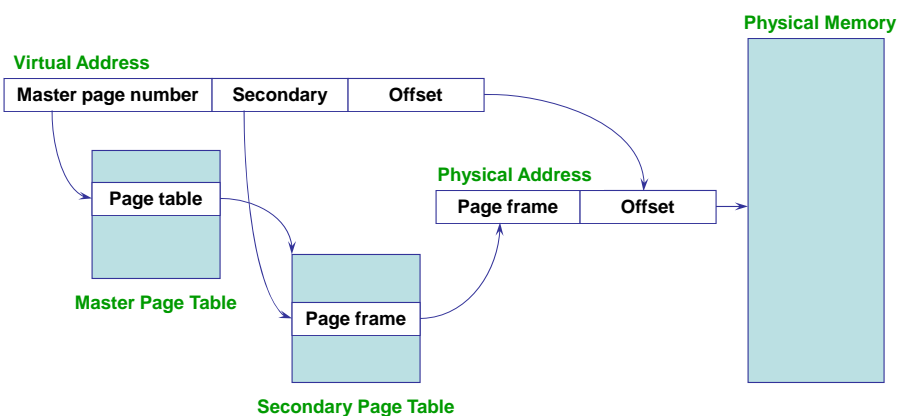
- Can still have internal fragmentation
  - Process may not use memory in multiples of a page
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (more later)
- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages =  $2^{20}$  PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - How to reduce page size?
- How do we only map what is being used?
  - Dynamically extending page table, but fragmentation

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 9 – Memory

25

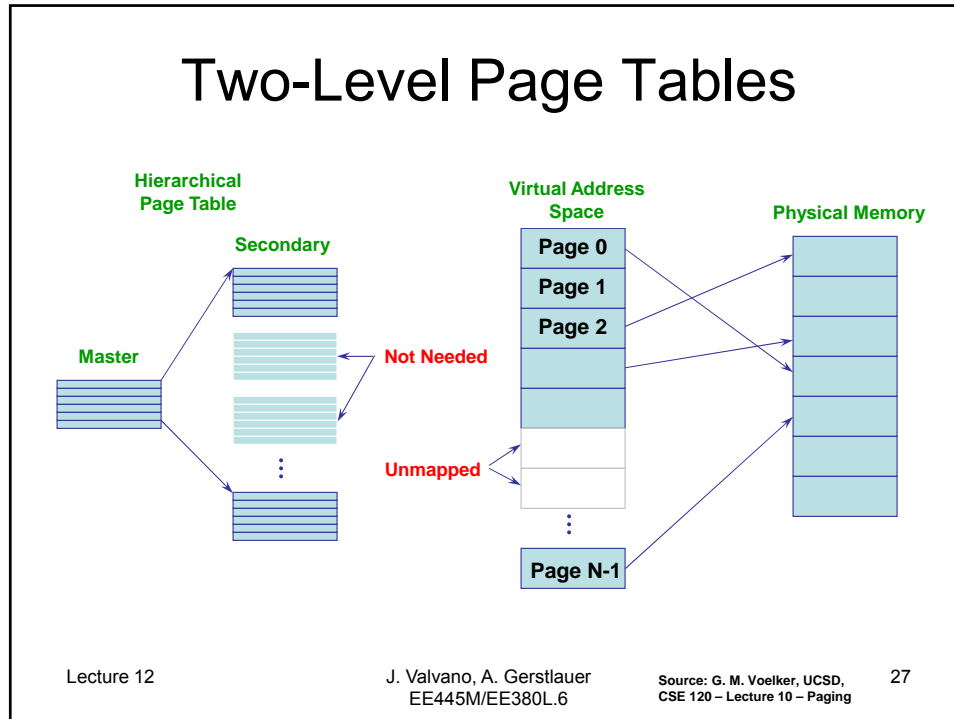
## Two-Level Page Tables



Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

26



## Addressing Page Tables

- Where do we store page tables?
  - Physical memory
    - Easy to address, no translation required
    - But, allocated tables consume memory for lifetime of VAS
  - Virtual memory (OS virtual address space)
    - Cold (unused) page table pages can be paged out to disk
    - But, addressing page tables requires translation
    - How do we stop recursion?
    - Do not page the outer page table (called **wiring**)
  - If we're going to page the page tables, might as well page the entire OS address space, too
    - Need to wire special code and data (fault, int handlers)

Lecture 12      J. Valvano, A. Gerstlauer      Source: G. M. Voelker, UCSD, CSE 120 – Lecture 10 – Paging      28  
EE445M/EE380L.6

## Efficient Translations

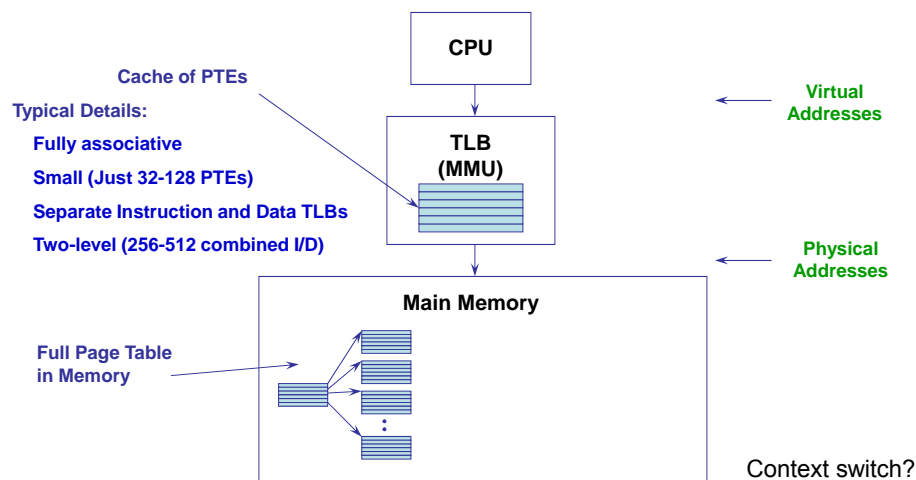
- Original page table scheme already doubled the cost of doing memory lookups
  - Lookup into page table + fetch the data
- Two-level page tables triple the cost!
  - 2x lookups into page tables, a third to fetch the data
  - And this assumes the page table is in memory
- How can we use paging but also have lookups cost about the same as fetching from memory?
  - Cache translations in hardware
  - Translation Lookaside Buffer (TLB)
  - TLB managed by Memory Management Unit (MMU)

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

29

## Translation Lookaside Buffer (TLB)



Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

30

## Memory Access Example

- Process is executing on CPU, issues a read to an address
  - What kind of address is it? Virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the **page number** of the address
  2. Common case is that the page number matches, returning a **page table entry (PTE)** for the mapping for this address
  3. TLB validates that the **PTE protection** allows reads (in this case)
  4. PTE specifies which **physical frame** holds the page
  5. MMU combines physical frame and offset into a **physical address**
  6. MMU then reads from that physical address, returns value to CPU
- Note: **This is all done by the hardware**

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

31

## TLB Miss

- If the TLB does not have mapping:
  1. MMU loads PTE from page table in memory
    - **Hardware managed TLB [x86]**
    - OS has already set up the page tables so that the hardware can access it directly, otherwise not involved
  2. Trap to the OS
    - **Software/OS managed TLB [MIPS, Alpha, Sparc, PowerPC]**
    - OS does lookup in page table, loads PTE into TLB
    - OS returns from exception, TLB continues
- **Replace existing PTE in TLB**
  - Done in hardware, e.g. least recently used
  - At this point, PTE for the address in the TLB

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

32



## Page Fault

- PTE can indicate a protection fault
  - Read/write/execute – operation not permitted
  - Invalid – virtual page not allocated/not in memory
- TLB traps to the OS (OS takes over)
  - R/W/E violation
    - OS sends fault back up to process, or intervenes
  - Invalid
    - Virtual page not allocated in address space
      - OS sends fault to process (e.g., segmentation fault)
    - Page not in physical memory
      - OS allocates frame, reads from disk (swap space)
      - Maps PTE to physical frame, update TLB

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

33

## Page Replacement

- Which page to evict on invalid page fault?
  - Page replacement policies
  - Avoid thrashing (if possible)
- Exploit locality
  - Temporal and spatial locality
  - Working set (pages most recently referenced)
  - FIFO, Least Recently Used (LRU), ...
- Dirty vs. clean pages (marked in PTE)
  - Only write back dirty pages to disk

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 11 – Page Replacement

34

## Advanced Functionality

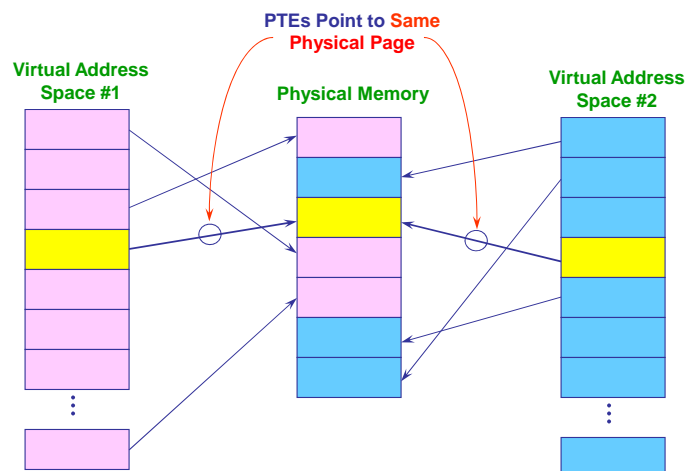
- Shared memory
  - PTEs of two processes point to same page
- Copy on Write (`fork()` a process)
  - Copy only page table to clone process
  - Copy memory frame only on first write
- Mapped files
  - Map pages from file on disk into memory

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

35

## Shared Memory

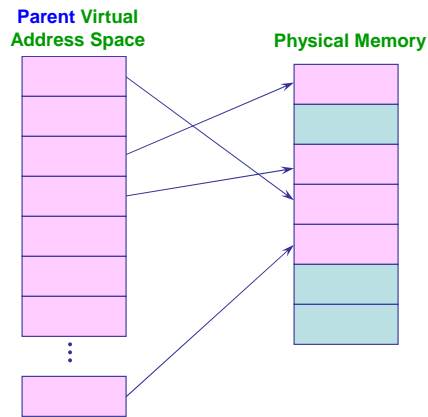


Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

36

# Copy on Write: Before Fork



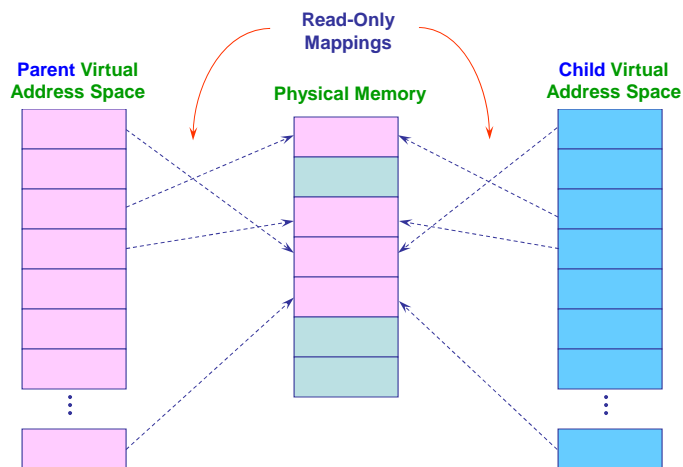
Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

37

# Copy on Write: Fork



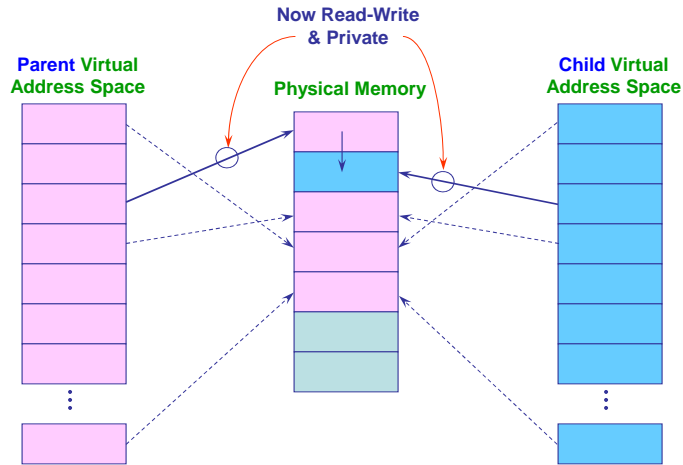
Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

38

## Copy on Write: On A Write



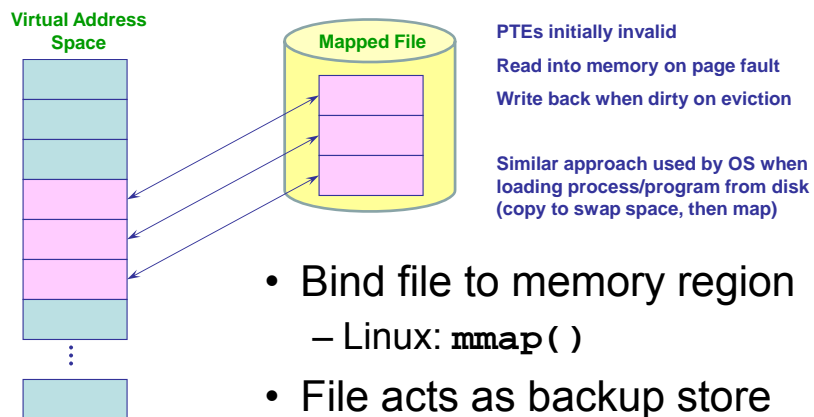
Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

39

## Mapped Files



- Bind file to memory region
  - Linux: `mmap ( )`
- File acts as backup store
  - Instead of swap space

Lecture 12

J. Valvano, A. Gerstlauer  
EE445M/EE380L.6

Source: G. M. Voelker, UCSD,  
CSE 120 – Lecture 10 – Paging

40

## Memory Management Summary

- Often not used in embedded devices
  - Overhead
    - Page table storage, Context switching
  - Unpredictable timing
    - TLB misses, Page faults
- Static memory management
  - Static data allocation, no heap
  - No MMU/paging
    - Compile/load time relocation (optionally segmented)
    - Hardware support for protection & translation
    - Swapping under program control (overlays)