



# CSCI [4 | 6]730 Operating Systems

## Main Memory



Maria Hybinette, UGA

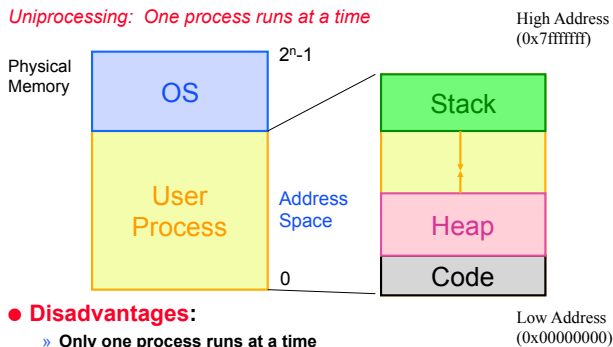
## Memory Questions?

- What is main memory?
- How does *multiple* processes share memory space?
  - » Key is how do they refer to memory addresses?
- What is *static* and *dynamic* allocation?
- What is *segmentation*?

Maria Hybinette, UGA

## Review: Motivation for Multiprogramming

*Uniprocessing: One process runs at a time*



- **Disadvantages:**
  - » Only one process runs at a time
  - » Process can destroy OS

Maria Hybinette, UGA

## Multiprogramming Goals

- **Sharing**
  - » Several processes coexist in main memory
  - » Cooperating processes can share portions of address space
- **Transparency**
  - » Processes are not aware that memory is shared
  - » Works regardless of number and/or location of processes
- **Protection**
  - » Cannot corrupt OS or other processes
  - » Privacy: Cannot read data of other processes
- **Efficiency**
  - » Do not waste CPU or memory resources
  - » Keep fragmentation low (later)

Maria Hybinette, UGA

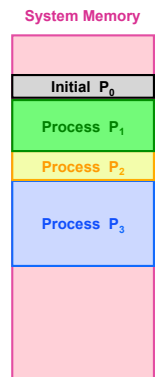
## Memory Addresses

- **Address space**
  - » What we go so far:
    - Physical addresses

Maria Hybinette, UGA

## Static Relocation (after loading)

- **Goal: Allow transparent sharing - Each address space may be placed anywhere in memory**
  - » OS finds free space for new process
  - » **Modify addresses *statically*** (similar to linker) when **loading the process**
  - » **Fixed addresses.**
- **Advantages:**
  - » Allows multiple processes to run
  - » Requires no hardware support

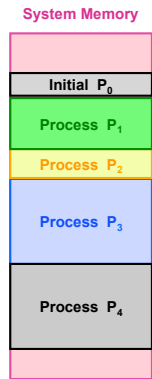


Maria Hybinette, UGA

## Static Reallocation

- **Disadvantages:**

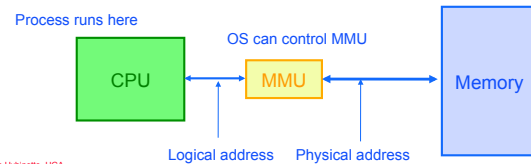
- » No protection
  - Process can destroy OS or other processes
  - No privacy
- » Address space must be allocated contiguously
  - Allocate space for worst-case stack and heap
  - Processes **may not grow**
- » Cannot move process after they are placed or loaded (**static addresses**)
- » Fragmentation (later)



Maria Hyönnelie, UGA

## Dynamic Relocation

- **Goal: Protect** processes from one another
- Requires hardware support
  - » Memory Management Unit (MMU)
- MMU dynamically changes process address *at every* memory reference (compute address *on-the-fly*)
  - » Process generates **logical** or **virtual** addresses
  - » Memory hardware uses **physical** or **real** addresses



Maria Hyönnelie, UGA

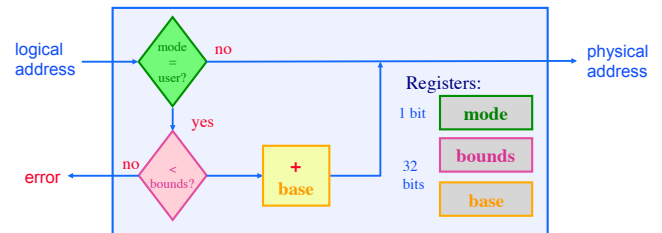
## Hardware Support for Dynamic Relocation

- Two operating modes
  - » **Privileged (protected, kernel) mode:** OS runs
    - When enter OS (trap, system calls, interrupts, exceptions)
    - Allows certain instructions to be executed
      - Can manipulate contents of MMU
    - Allows OS to access all of physical memory
  - » **User mode:** User processes run
    - Perform translation of logical address to physical address
- MMU contains base and bounds registers
  - » **base:** start location for address space (physical address)
  - » **bounds:** size limit of address space (memory span)

Maria Hyönnelie, UGA

## Implementation of Dynamic Relocation

- Translation on every memory access of user process
  - » MMU compares logical address to bounds register
    - if logical address is greater, then generate error
  - » MMU adds base register to logical address to form physical address



Maria Hyönnelie, UGA

## Example of Dynamic Relocation

- What are the **physical addresses** for the following 16-bit **logical addresses (HEX: highest F:1111)?**
- **Process 1:** base: 0x4320, bounds: 0x2220 (in HEX)
  - » 0x0000:
  - » 0x1110:
  - » 0x3000:
- **Process 2:** base: 0x8540, bounds: 0x3330
  - » 0x0000:
  - » 0x1110:
  - » 0x3000:
- **Operating System**
  - » 0x0000:
  - » 0x5FFF:

Maria Hyönnelie, UGA

## Managing Processes with *Base* and *Bounds*

- Context-switch
  - » Add base and bounds registers to PCB
  - » Steps:
    1. Change to privileged mode
    2. Save base and bounds registers of old process
    3. Load base and bounds registers of new process
    4. Change to user mode and jump to new process
- What if don't change base and bounds registers when switch?
- Protection requirement
  - » User process cannot change base and bounds registers
  - » User process cannot change to privileged mode

Maria Hyönnelie, UGA

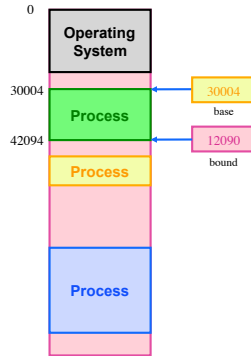
## Base and Bounds Discussion

### Advantages

- » Provides protection (both read and write) across address spaces
- » Supports dynamic relocation
  - Can move address spaces
  - Why might you want to do this?
- » Simple, inexpensive: Few registers, little logic in MMU
- » Fast: Add and compare can be done in parallel

### Disadvantages

- » Each process must be allocated **contiguously** in physical memory
  - Must allocate memory that may not be used by process
- » No partial sharing: Cannot share limited parts of address space



Maria Hyömette, UGA

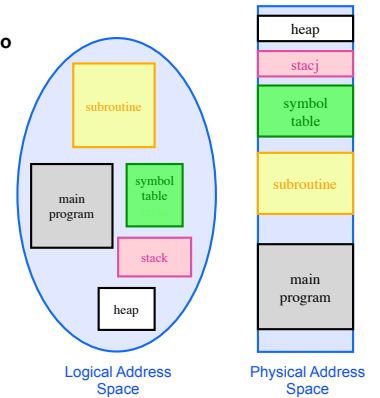
## Segmentation

### Divide address space into logical segments

- » Each segment corresponds to logical entity in address space
  - code, stack, heap

### Each segment can independently:

- » be placed separately in physical memory
- » grow and shrink
- » be protected (separate read/write/execute protection bits)



Maria Hyömette, UGA

## Segmented Addressing

### How does process designate a particular segment?

- » Use part of logical address
  - Top bits of logical address select segment
  - Low bits of logical address select offset within segment

Maria Hyömette, UGA

## Segmentation Implementation

### MMU contains Segment Table (per process)

- » Each segment has own base and bounds, protection bits
- » Example: 14 bit logical address, 4 segments

Segment	Base	Bounds	R	W
0	0x2000	0x06ff	1	0
1	0x0000	0x04ff	1	0
2	0x3000	0x0fff	1	1
3	0x1000	0x0fff	0	0

### Translate logical addresses ⇒ physical addresses:

- » 0x0240: 0<sup>th</sup> segment 240 internal address within segment ⇒ what address?
- » 0x1108:
- » 0x265c:
- » 0x3002:

Maria Hyömette, UGA

## Discussion of Segmentation

### Advantages

- » Enables sparse allocation of address space
  - Stack and heap can grow independently
  - Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
  - Stack: OS recognizes reference outside legal segment, extends stack implicitly
- » Different protection for different segments
  - Read-only status for code
- » Enables sharing of selected segments
- » Supports dynamic relocation of each segment

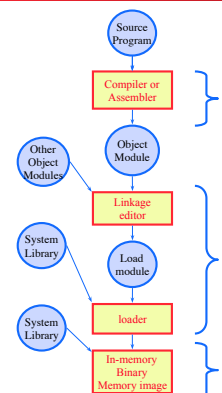
### Disadvantages

- » Each segment must be allocated contiguously
  - May not have sufficient physical memory for large segments

Maria Hyömette, UGA

## When to Bind Physical & Logical Addresses

- **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
- **Load time:** Must generate relocatable code if memory location is not known at compile time
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)



Maria Hyömette, UGA

# Motivation for Dynamic Memory

- **Why do processes need dynamic allocation of memory?**
  - » Do not know amount of memory needed at compile time
  - » Must be pessimistic when allocate memory statically
    - Allocate enough for worst possible case
    - Storage is used inefficiently
- **Recursive procedures**
  - » Do not know how many times procedure will be nested
- **Complex data structures: lists and trees**
  - » `struct my_t`

```
*p = (struct my_t *)malloc(sizeof(struct my_t));
```
- **Two types of dynamic allocation**
  - » Stack
  - » Heap

Maria Hyömette, UGA

# Stack Organization

- **Definition:** Memory is freed in opposite order from allocation
 

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```
- **Implementation:** Pointer separates allocated and freed space
  - » Allocate: Increment pointer
  - » Free: Decrement pointer

Maria Hyömette, UGA

# Stack Discussion

OS uses stack for procedure call frames (local variables)

```
main()
{
  int A = 0;
  maria(A);
  printf("A: %d\n", A);
}

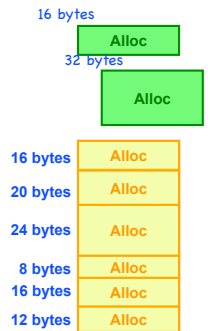
void maria( int Z )
{
  int A = 2;
  Z = 5;
  printf("A: %d Z: %d\n", A, Z);
}
```

- **Advantages**
  - » Keeps all free space contiguous (and keep order of calls)
  - » Simple to implement
  - » Efficient at run time
- **Disadvantages**
  - » Not appropriate for all data structures

Maria Hyömette, UGA

# Heap Organization

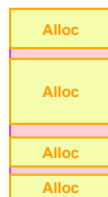
- **Definition:** Allocate from any random location
  - » Memory consists of allocated areas and free areas (holes)
  - » Order of allocation and free is unpredictable
- **Advantage**
  - » Works for all data structures
- **Disadvantages**
  - » Allocation can be slow
  - » End up with small chunks of free space
    - fragmentation



Maria Hyömette, UGA

# Fragmentation

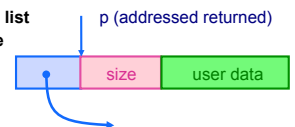
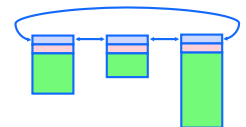
- **Definition:** Free memory that is too small to be usefully allocated
  - » **External:** Visible to allocator
  - » **Internal:** Visible to requester (e.g., if must allocate at some granularity)
- **Goal:** Minimize fragmentation
  - » Few holes, each hole is large
  - » Free space is contiguous
- **Stack**
  - » All free space is contiguous
  - » No fragmentation
- **Heap**
  - » How to allocate to minimize fragmentation?



Maria Hyömette, UGA

# Heap Implementation: Free List

- **Data structure: free list**
  - » A circular linked list of free blocks, tracks memory not in use
  - » Header in each block
    - size of block
    - ptr to next block in list
- `void *Allocate( x bytes )`
  - » Choose block large enough for request ( $\geq x$  bytes)
  - » Keep remainder of free block on free list
  - » Update list pointers and size variable
  - » Return pointer to allocated memory
- `Free( ptr )`
  - » Add block back to free list
  - » Merge (coalesce) adjacent blocks in free list, update ptrs and size variables



Maria Hyömette, UGA

## Heap Allocation Policies

- **Best fit**
  - » Search entire list for each allocation
  - » Choose free block that **most closely matches** size of request
  - » Optimization: Stop searching if see exact (close) match
- **First fit**
  - » **Version 1:**
    - Allocate first block that is large enough
  - » **Version 2:**
    - Rotating first fit (or “Next fit”):
      - Variant of first fit, remember place in list
      - Start with next free block each time
- **Worst fit**
  - » Allocate largest block to request (**most leftover space**)

Maria Hyönnelie, UGA

## Heap Allocation Examples

Scenario: Two free blocks of size 20 and 15 bytes

- Allocation stream: 10, 20
  - » Best
  - » First
  - » Worst
- Allocation stream: 8, 12, 12
  - » Best
  - » First
  - » Worst

Maria Hyönnelie, UGA

## Comparison of Allocation Strategies

- **No optimal algorithm**
  - » Fragmentation highly dependent on workload
- **Best fit**
  - » Tends to leave some (**very large holes**) and some **very small holes**
    - Can’t use very small holes easily
- **First fit**
  - » Tends to leave “average” sized holes
  - » **Advantage:** Faster than best fit
  - » Next fit used often in practice
- **Uses a ‘Modified’ Buddy allocation Scheme (Linux)**
  - » **Minimizes external fragmentation**
  - » **Disadvantage:** *Internal fragmentation* when not  $2^n$  request

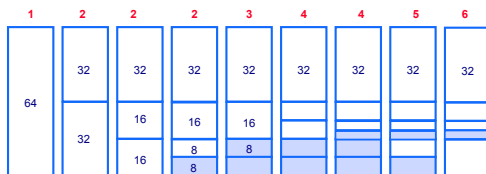
Maria Hyönnelie, UGA

## Simple Buddy Allocation

- **Fast, simple allocation for blocks of  $2^n$  bytes [Knuth68]**
- **void \*Allocate ( k bytes )**
  - » Raise allocation request to nearest (next highest)  $s = 2^n$ 
    - 63K allocates a 64K block
    - 65K allocates a 128K block
    - 31K allocates a 32K block
  - » Search free list for appropriate size (near  $s$ )
    - **Recursively divide larger free blocks** until find block of size  $s$
    - “Buddy” block remains free
- **Free( ptr )**
  - » Mark blocks as free
  - » **Recursively coalesce** block with buddy, if buddy is free
    - May coalesce lazily (later, in background) to avoid overhead

Maria Hyönnelie, UGA

## Buddy Algorithm



**Toy Example:** Assume there is initially 64K bytes of memory and the **first** request is for 5K bytes

1. Round up request to nearest  $s=2^n K$  so we need a  $s=8K$  bytes and search for a block of that size
  - Divide 64K block chunk into half (again, again and again) until desired block size and return to caller (shaded area)
2. Suppose **second** request is for 8 then return remaining free chunk to be used
3. **Third** request is for 4 -- split block again and again and return to caller
4. Fourth and last allocated 8 chunk is **released** and **returned**
5. Finally the other is released and coalesced

Maria Hyönnelie, UGA

## Buddy Implementation

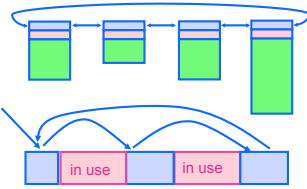
- **IF holes in free list is of power of 2 in size then very easy to implement**
  - » A buddy’s hole is the exclusive OR of the hole size and starting address of hole.
- **Example:**
  - » **Blocks of size 4 could start at addresses:**
    - 0, 4, 8, 12, 16, 20,

Starting &		Old	New	
0	0 0 4	0000000	0000100	4
4	4 0 4	0000100	0000000	0
8	8 0 4	0001000	0001100	12
12	12 0 4	0001100	0001000	8
16	16 0 4	0010000		
20	20 0 4	0010100		

Maria Hyönnelie, UGA

## Memory Allocation (K&R)

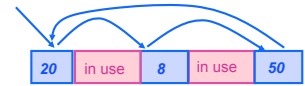
- How are `malloc()`, `free()` implemented?
- **Data structure:** Circular list of free chunks
  - » Header for each element of free list
    - pointer to next free block
    - size of block
- **Malloc:** first-fit (next-fit) with **splitting** (large chunks)
- **Free:** coalescing with adjacent chunks if they are free
- **Disadvantage:**
  - » Fragmentation of memory due to first-fit (next-fit) strategy
  - » Linear time to scan list during `malloc` and `free`



Maria Hyönnelä, UGA

## Improvements

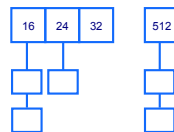
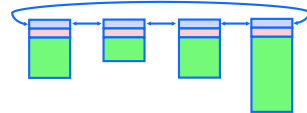
- **Placement:** reducing fragmentation
  - » Deciding which free chunk to use
  - » Use best fit or *good* fit
    - Example: `malloc(8)` returns 8 byte block instead of 20 byte block
- **Splitting:** only split when saving is big enough: `malloc(14)` allocate the entire block.
- **Coalescing:** defer coalescing
- **Performance:**
  - » Doubly-linked list



Maria Hyönnelä, UGA

## Memory Allocation in Practice (improved)

- How are `malloc()`, `free()` implemented?
- **Data structure:** Free lists
  - » Header for each element of free list
    - pointer to next free block
    - size of block
    - magic number
      - consistency checking
- **Two free lists**
  - » One organized by size (binning)
    - Separate list for each popular, small size (e.g., 1 KB) -- range of sizes -- fewer bins
    - Allocation is fast, no external fragmentation
  - » Second is sorted by address
    - Use next fit to search appropriately
    - Free blocks shuffled between two lists



Maria Hyönnelä, UGA

## Modified Buddy Algorithm

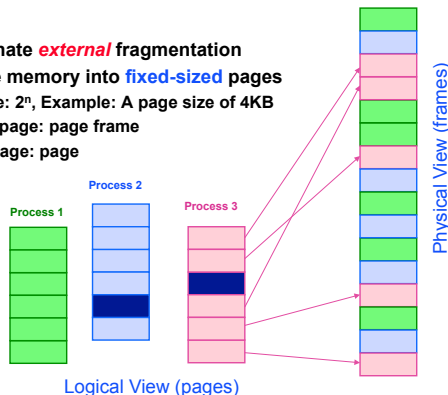
- Linux uses buddy system with the additional of having a cache of pointers to free memory (a slab index array):
  - » the first element is the head of a list of blocks of size 'unit 1',
  - » the second element is a list of blocks of size 'unit 2'
  - » the third element is a list of blocks of size 'unit 3', ...
- Each index contains only slabs of a specific size
  - » And they are linked together as linked list (which in turn links to the next free element (so the slabs themselves may not be contiguous))



Maria Hyönnelä, UGA

## Paging

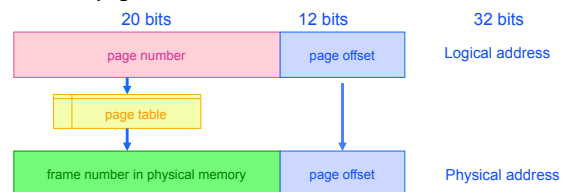
- **Goal:** Eliminate *external* fragmentation
- **Idea:** Divide memory into **fixed-sized** pages
  - » Page Size:  $2^n$ , Example: A page size of 4KB
  - » Physical page: page frame
  - » Logical page: page



Maria Hyönnelä, UGA

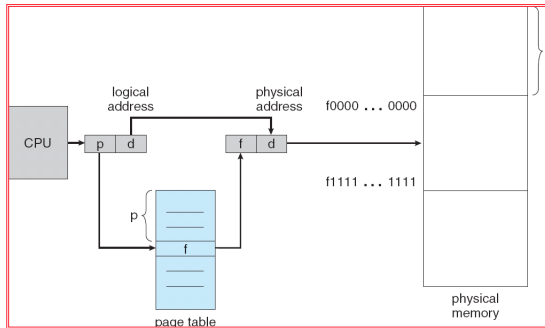
## Translation of Page Addresses

- How to translate logical address to physical address:
  - » High-order bits of address designate page number
  - » Low-order bits of address designate offset within page



Maria Hyönnelä, UGA

## Paging Hardware



Maria Hyömette, UGA

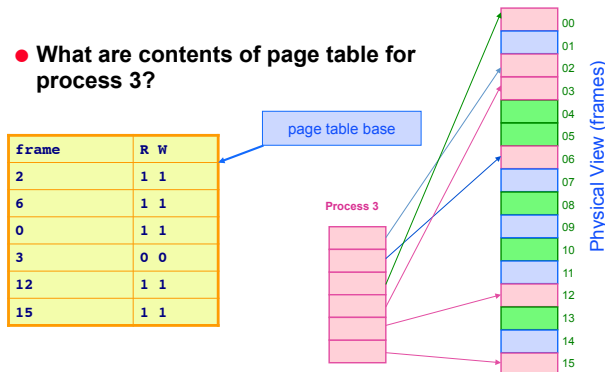
## Page Table Implementation

- Page table per process
  - » Page table entry (PTE) for each virtual page number (vpn)
    - frame number or physical page number (ppn)
    - R/W protection bits
- Simple vpn  $\Rightarrow$  ppn mapping:
  - » No bounds checking, no addition
  - » Simply **table lookup** and bit substitution
- How many entries in table?
- Track page table base in PCB, change on context-switch

Maria Hyömette, UGA

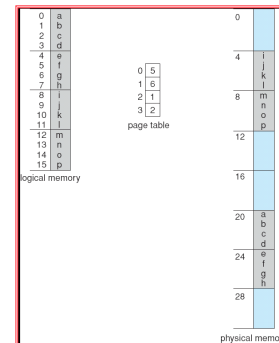
## Page Table Example

- What are contents of page table for process 3?



Maria Hyömette, UGA

## Page Table: Example 2



32-byte (8 pages) addressable memory and 4-byte pages

Maria Hyömette, UGA

## Advantages of Paging

- No external fragmentation
  - » Any page can be placed in any frame in physical memory
  - » Fast to allocate and free
    - Alloc: No searching for suitable free space
    - Free: Doesn't have to coalesce with adjacent free space
    - Just use bitmap to show free/allocated page frames
- Simple to swap-out portions of memory to disk
  - » Page size matches disk block size
  - » Can run process when some pages are on disk
  - » Add "present" bit to page table entry (PTE)
- Enables sharing of portions of address space
  - » To share a page, have PTE point to same frame

Maria Hyömette, UGA

## Disadvantages of Paging

- Internal fragmentation: Page size may not match size needed by process
  - » Wasted memory grows with larger pages
  - » large vs small page size
- Additional memory reference to look up in page table --> Very inefficient
  - » Page table must be stored in memory
  - » MMU stores only base address of page table
- Storage for page tables may be substantial
  - » Simple page table: Requires PTE for all pages in address space
    - Entry needed even if page not allocated
  - » Problematic with dynamic stack and heap within address space

Maria Hyömette, UGA

## Combine Paging and Segmentation

- **Goal:** More efficient support for sparse address spaces
- **Idea:**
  - » Divide address space into segments (code, heap, stack)
    - Segments can be variable length
  - » Divide each segment into fixed-sized pages
- Logical address divided into three portions: System 370

seg # (4 bits)	page number (18 bits)	page offset (12 bits)
-------------------	-----------------------	-----------------------

- **Implementation**
  - » Each segment has a page table
  - » Each segment track base (physical address) and bounds of page table (number of PTEs)

Maria Hyönnelie, UGA

## Example of Paging and Segmentation

### Example of Paging and Segmentation

seg	base	bounds	R	W
0	1400	5	1	0
1	6300	400	0	0
2	4300	1100	1	1
3	1100	5	1	1

...
1100 0x01f
0x011
0x003
0x02a
0x013
...
1400 0x00c
0x007
0x004
0x00b
0x006
...

Maria Hyönnelie, UGA

## Advantages of Paging and Segmentation

- **Advantages of Segments**
  - » Supports sparse address spaces
    - Decreases size of page tables
    - If segment not used, not need for page table
- **Advantages of Pages**
  - » No external fragmentation
  - » Segments can grow without any reshuffling
  - » Can run process when some pages are swapped to disk
- **Advantages of Both**
  - » Increases flexibility of sharing
    - Share either single page or entire segment

Maria Hyönnelie, UGA

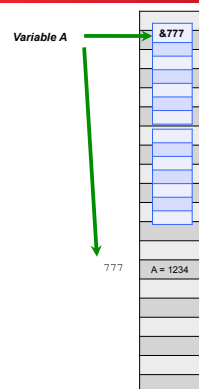
## Disadvantages of Paging and Segmentation

- **Overhead of accessing memory**
  - » Page tables reside in main memory
  - » Overhead reference for every real memory reference
- **Large page tables**
  - » Must allocate page tables contiguously
  - » More problematic with more address bits
  - » **Page table size**
    - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Maria Hyönnelie, UGA

## Disadvantages of Paging and Segmentation

- **Overhead of accessing memory**
  - » Page tables reside in main memory
  - » Overhead reference for every real memory reference
- **Large page tables**
  - » Must allocate page tables contiguously
  - » More problematic with more address bits
  - » **Page table size (32 bit address):**
    - Logical address space:  $2^{32}$
    - Assume page size is 4 KB,  $4,096 \rightarrow 2^{12}$
    - Page table has  $2^{32}/2^{12}$  entries =  $2^{20}$ 
      - 1,048,576 Entries ! Each entry is 4 bytes
      - » 4MB for EACH page table



Maria Hyönnelie, UGA

- **4 MB page tables**
  - » Contiguous in memory?
    - Divide the page tables into smaller pieces
    - Idea is to page the page table hierarchically
      - Assume 2 levels for a start.

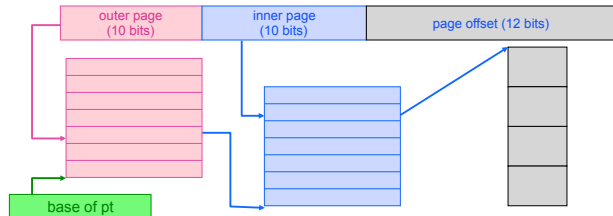
Maria Hyönnelie, UGA



## Hierarchical Paging: Page the Page Tables

- **Problem:** Large logical address space  $2^{32} - 2^{64}$
- **Goal:** Allow page tables to be allocated non-contiguously
- **Approach:** Page the page tables (4K page size  $4,096$  is  $2^{12}$ )
  - » Creates multiple levels of page tables

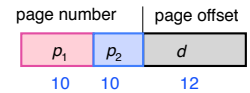
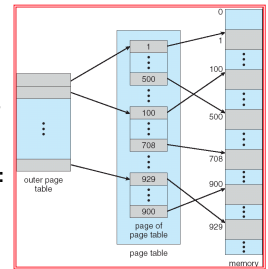
32-bit address: » Only allocate page tables for pages in use (allows)



Maria Hyömette, UGA

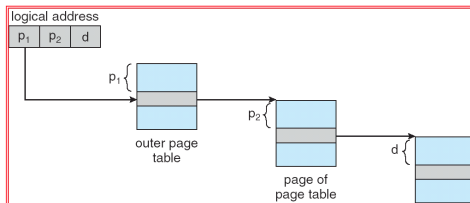
## Example: Two Level Page Table

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - » a page number consisting of 20 bits
  - » a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - » a 10-bit page number
  - » a 10-bit page offset
- Thus, a logical address is as follows:
  - » where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the outer page table



Maria Hyömette, UGA

## Address-Translation Scheme



Maria Hyömette, UGA

## Page the Page Tables (Homework)

- How should logical address be structured?
  - » How many bits for each paging level?
- Calculate such that page table fits within a page (A Page Table Entry = PTE)
  - » **Goal:** PTE size \* number PTE = page size
  - » Assume PTE size = 4 bytes; page size = 4KB
  - $2^2 * \text{number PTE} = 2^{12}$
  - $\rightarrow \text{number PTE} = 2^{10}$
  - $\rightarrow \# \text{ bits for selecting inner page} = 10$  (see earlier slides)
- Apply recursively throughout logical address
  - Will assign homework through different layers of addressing all the way to disk

Maria Hyömette, UGA

## Other Observation

- Accessing a memory location requires two accesses in main memory.
  - One to access the page table (which is in main memory)
    - A contiguous lookup table.
  - Another one that access the memory location
    - Anywhere in memory
- **Problem:** Expensive! Can we do better?

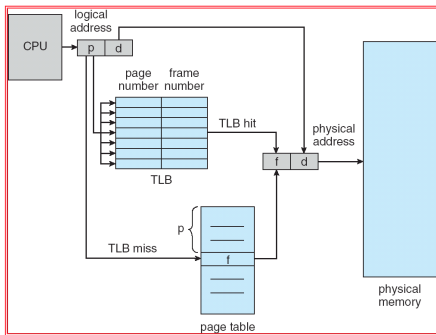
Maria Hyömette, UGA

## Translation Look-Aside Buffer (TLB)

- **Goal:** Avoid page table lookups in main memory (i.e., a total of two memory accesses)
- **Idea:** Hardware cache of recent page translations
  - » Typical size: 64 - 2K entries
  - » Index by segment + vpn  $\rightarrow$  ppn
- Why does this work?
  - » process references few unique pages in time interval
  - » spatial, temporal locality
- On each memory reference, check TLB for translation
  - » If present (hit): use ppn and append page offset
  - » Else (miss): Use segment and page tables to get ppn
    - Update TLB for next access (replace some entry)
- How does page size impact TLB performance? (food for thought).

Maria Hyömette, UGA

## Paging Hardware With TLB



Maria Hyömette, UGA

## Effective Access Time

- **Associative Lookup (TLB) =  $\epsilon$  time unit (small fraction of the time to go to main memory)**
  - » Assume memory cycle time is 1 microsecond
  - » Hit ratio – percentage of times that a page number is found in the *associative registers*; ratio related to number of associative registers
  - » Hit ratio =  $\alpha$  (alpha)
  - » Effective Access Time (EAT)
$$\text{EAT} = (1 + \epsilon) \alpha + (2 + \epsilon)(1 - \alpha)$$
$$= 2 + \epsilon - \alpha$$

Maria Hyömette, UGA

## What Page Size? Page Size Trade-offs

- **Internal Fragmentation**
  - » Smaller the page size the less the internal fragmentation
- **Number of pages**
  - » The smaller the pages the greater the **number** of pages
  - » **Larger Page tables**
- **Page size and page faults**
  - » Larger page size implies (less or more) page faults.

Maria Hyömette, UGA