

CSC369: Operating Systems

Intro to Memory Management

Andrew Petersen

Midterm

- We have a midterm next week, in tutorial.
- The midterm covers all material up to and including this week.
- Review the lecture and tutorial material
- Expect to see questions related to the assignments
- Sample tests from previous years are posted on the website.

Group Test

- As you leave the midterm, you'll draw a group number out of a hat.
- Well, maybe not a hat. Maybe a box.
- That group number pairs you up with someone else in class.
- For the first thirty minutes of class on Friday, you and your partner will work together on a group test.

Group Test Mechanics

- The group component of the test is worth a total of 10 marks.
 - The first 5 are part of the test. 10%.
 - The second 5 are extra credit.
- You and your partner must both contribute to the test.
- You and your partner must come to an agreement on each answer you write down.

Any Questions about the Test?

- This week: Memory mechanisms
- Tutorial: Midterm
- Next week: Group test, Intro to A3, Segmentation

Where are we?

- We've decided how to distinguish between different threads of control
 - Processes, threads
- We've discussed methods for synchronizing threads and communicating between them
 - Locks, semaphores, monitors, atomicity
 - Deadlock, starvation
- We've learned how to share the CPU resource between threads
- Now ... memory

Looking Ahead ...

- To impose **logical and physical hierarchy** and enable **sharing, protection, and relocation**, we'll need some tools.
- Modern systems use **virtual memory**, a complicated technique requiring hardware and software support
- Virtual memory is based on **segmentation** and **paging**
- We'll build up to virtual memory by looking at some simpler schemes first
- We've seen **address translation**. Next, partitions

Address Translation

- Programs must be in memory to execute
- Program binary is loaded into a process's address space
- Addresses in the program must be **translated (mapped, bound)** to real (**physical**) addresses
 - Programmers use symbolic addresses (i.e., variable names) to refer to memory locations
 - CPU fetches from, and stores to, real memory addresses
- *Address translation is the process of linking variable names to physical locations.*

Outline

- Introduction to Memory Management
 - Address Translation
- Memory Partitioning
- Relocation
- Paging
- Segmentation
- Virtual Memory

Partitioning

- How do we determine where programs are placed within physical memory?
- This is the **partitioning problem**
 - Two considerations:
 - Efficiency: are we wasting space?
 - Flexibility: are we supporting as many processes as possible?
 - We will consider **static** and **dynamic** solutions

Fixed Partitioning of Physical Addr

- Divide memory into regions with fixed boundaries
 - Can be equal-size or unequal-size
- Operating system occupies one partition
- A single process can be loaded into each remaining partition
 - Memory is wasted if process is smaller than partition (**internal fragmentation**)
 - Programmer must deal with programs that are larger than partition (**overlays**)

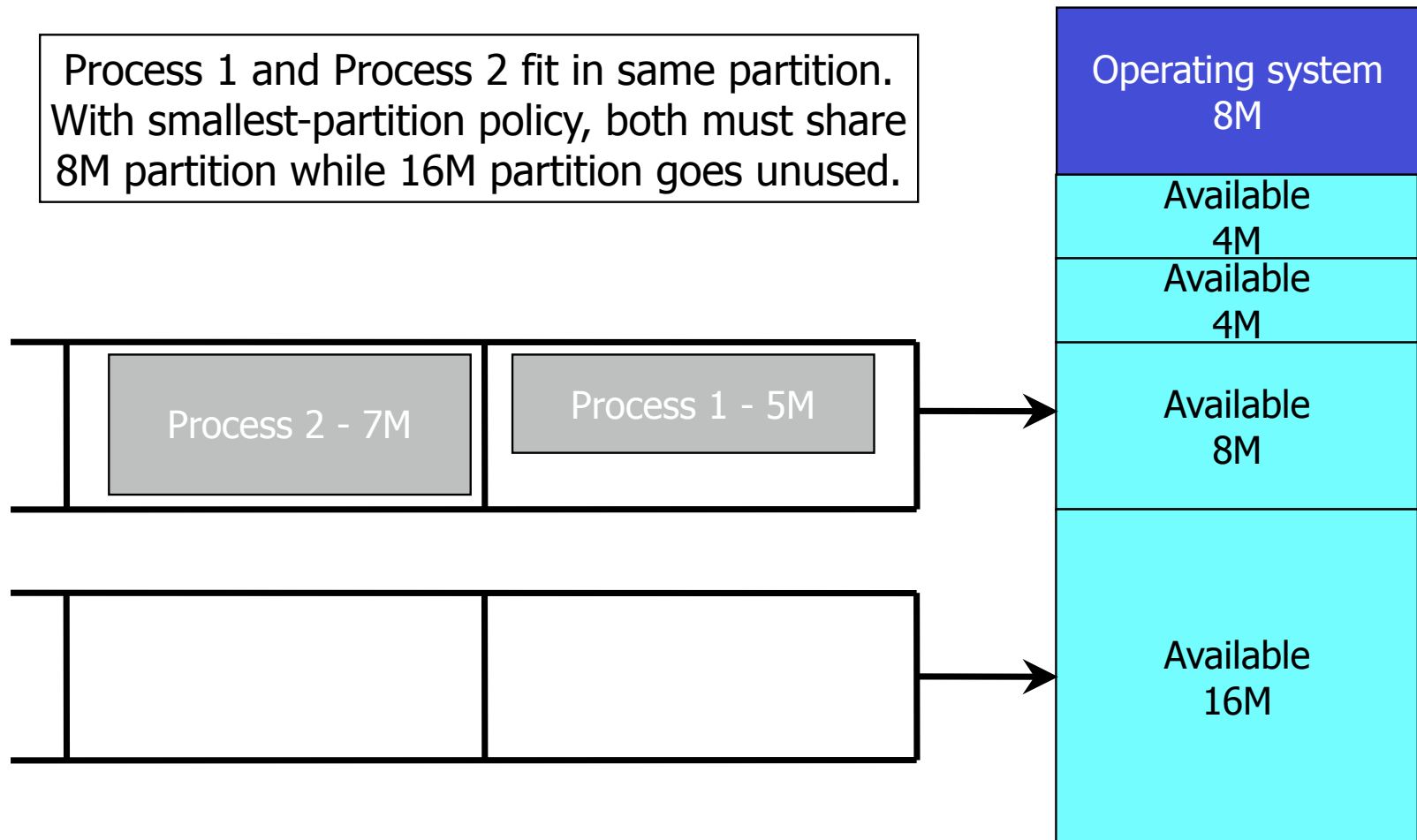


Placement with Fixed Partitions

- Number of partitions determines number of active processes
- If all partitions are occupied by waiting processes, swap some out, bring others in
- Equal-sized partitions:
 - Process can be loaded into any available partition
- Unequal-sized partitions:
 - Queue-per-partition, assign process to smallest partition in which it will fit
 - A process always runs in the same size of partition
 - Or, single queue, assign to smallest available partition in which the process fits

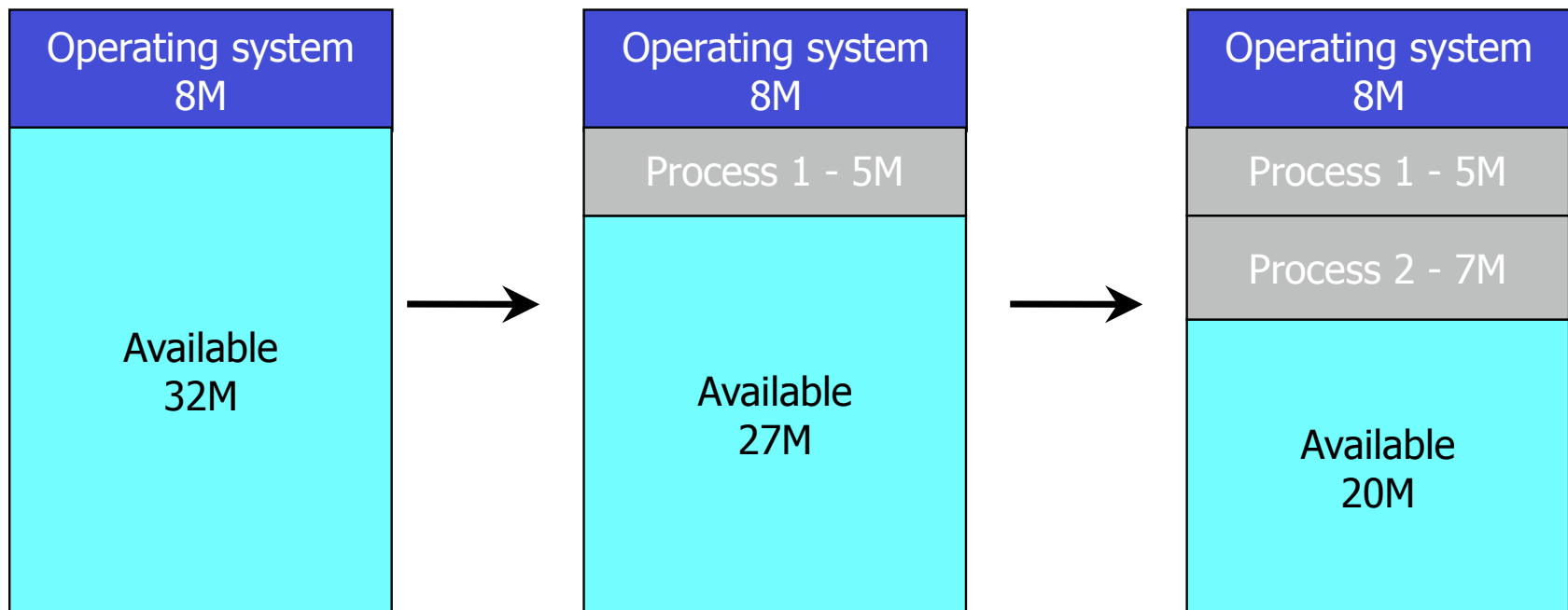
Placement Example

Process 1 and Process 2 fit in same partition.
With smallest-partition policy, both must share
8M partition while 16M partition goes unused.



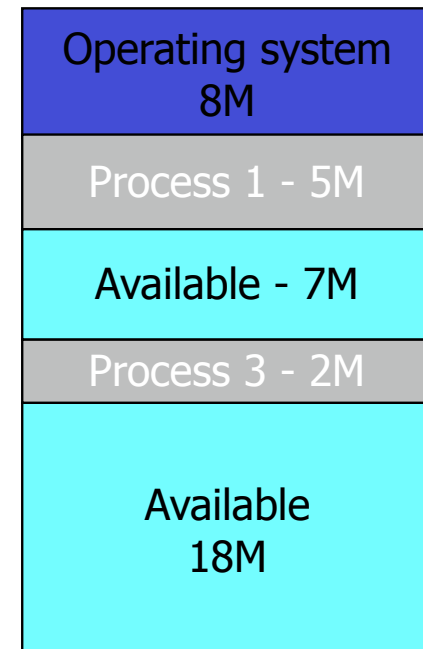
Dynamic Partitioning

- Partitions vary in length and number over time
- When a process is brought in to memory, a partition of exactly the right size is created to hold it



Concerns about Dynamic Partitioning

- As processes come and go, “holes” are created
 - Some blocks may be too small for any process
 - This is called **external fragmentation**
- OS may move processes around to create larger chunks of free space
 - This is called **compaction**
 - Requires processes to be **relocatable**
- We must know, at loadtime, what the maximum size of the process is
 - Else, we must be able to add to its partition, potentially relocating it



Example: Heap Management

- How do malloc() and free() solve the partitioning problem?
 - Dynamic partitioning system, without relocation
- malloc(size) returns a pointer to a block of memory of at least “size” bytes, or NULL
- free(ptr) releases the previously-allocated block pointed to by “ptr”
- Internally, malloc/free manage a contiguous range of logical addresses
 - Starts just after uninitialized data segment
 - Can be extended with brk() system call

Tracking Memory Allocations

- First Implementation: Bitmaps
 - 1 bit per allocation unit
 - “0” == free, “1” == allocated
 - See kern/arch/mips/mips/dumbvm.c
 - Allocation unit is a page of physical memory
 - Allocating a N-unit chunk requires scanning bitmap for sequence of N zero’s
 - Drawback? Slow

Memory:

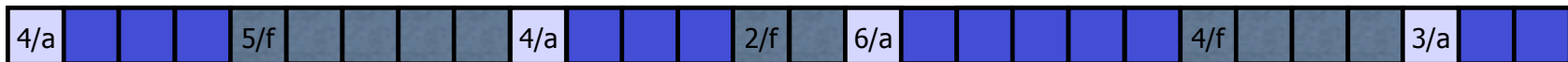


Bitmap:

111000001110011111000011

Tracking Allocations (continued)

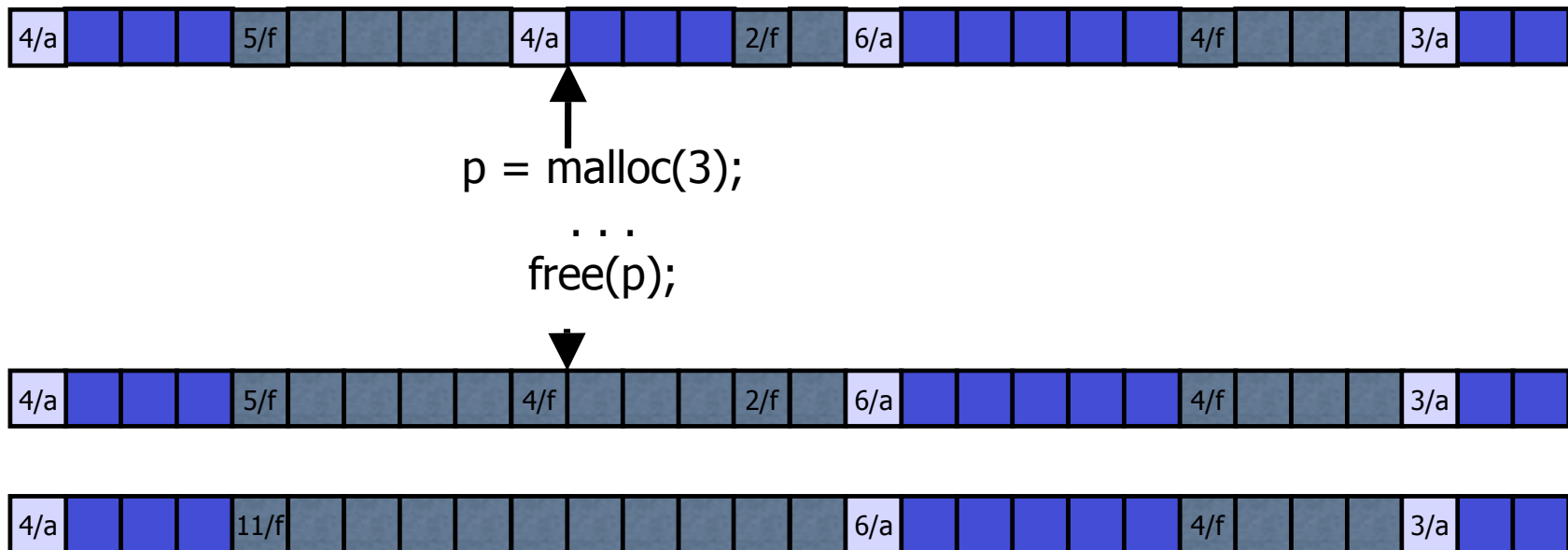
- Free lists: a linked list of allocated and free segments
 - List needs memory too. Where do we store it?
- Implicit list
 - Each block has header that records size and status (allocated or free)
 - Searching for free block is linear in total number of blocks



- Explicit list
 - Store pointers in free blocks to create doubly-linked list

Freeing Blocks

- Adjacent free blocks can be **coalesced**



- Easier if all blocks end with a footer with size/status info (called the **boundary tag**)

Outline

- Introduction to Memory Management
 - Address Translation
- Memory Partitioning
- Relocation
- Paging
- Segmentation
- Virtual Memory

Placement Heuristics

- Compaction is time-consuming and not always possible
- We can reduce the need for it by being careful about how memory is allocated to processes over time
 - **First-fit** - choose first block that is large enough; search can start at beginning, or where previous search ended (called next-fit)
 - **Best-fit** - choose the block that is closest in size to the request
 - **Worst-fit** – choose the largest block
 - **Quick-fit** – keep multiple free lists for common block sizes

Comparing Placement Algorithms

- **Best-fit**
 - left-over fragments tend to be small (unusable)
 - In practice, similar storage utilization to first-fit
- **First-fit**
 - Simplest, and often fastest and most efficient
 - May leave many small fragments near start of memory that must be searched repeatedly
 - Next-fit variant tends to allocate from end of memory
 - Free space becomes fragmented more rapidly
- **Worst-fit**
 - Not as good as best-fit or first-fit in practice
- **Quick-fit**
 - Great for fast allocation, generally harder to coalesce

Relocation

- Swapping and compaction require a way to change the physical memory addresses a process refers to
- Really, we need **dynamic relocation** (execution-time binding of addresses)
 - Processes refer to relative addresses
 - The hardware translates to physical addresses as instructions are executed
- Let's begin with minimum requirements to relocate fixed or dynamic partitions...
 - Assume: All memory used by a process is contiguous

Hardware for Relocation

- Basic idea: add **relative address** to process starting (**base**) address to form a **real (physical) address**
 - Check that the address generated is within process's space
- 2 registers, “base” and “limit”
 - When process is assigned to CPU (i.e., set to “Running” state), load **base register** with starting address *for that process*
 - Load the **limit register** with last legal address of process
 - On memory reference instruction (load, store) add base to address and compare with limit
 - If compare fails, trap to operating system
 - **if (addr < base || addr >= (base+limit)) then trap**
 - This is an **illegal address exception**

Problems with Partitioning

- With fixed partitioning, internal fragmentation and need for overlays are big problems
 - Scheme is too inflexible
- With dynamic partitioning, external fragmentation and managing the available space are major problems
- Basic problem is that assumption we made that processes must be allocated to contiguous blocks of memory
 - Hard to figure out how to size these blocks given that processes are not all the same
- **Paging** provides the *appearance* of a contiguous allocation

Outline

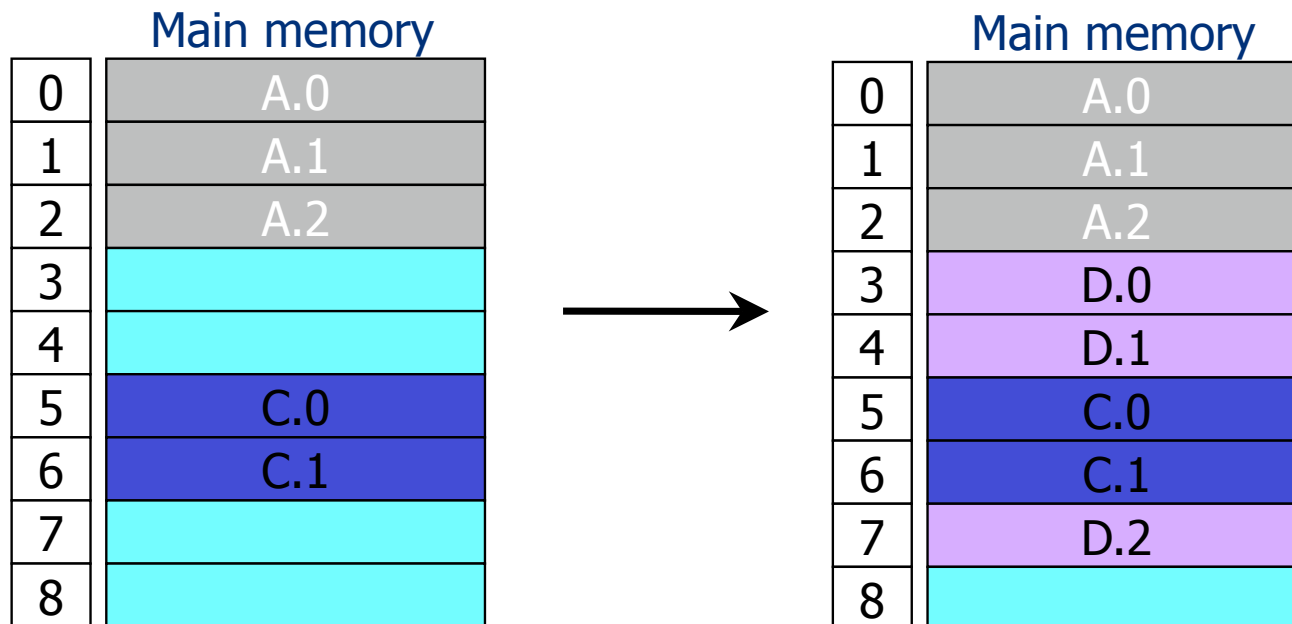
- Introduction to Memory Management
 - Address Translation
- Memory Partitioning
- Relocation
- Paging
- Segmentation
- Virtual Memory

Paging

- Partition memory into equal, fixed-size chunks
 - These are called **page frames** or simply **frames**
- Divide processes' memory into chunks of the same size
 - These are called **pages**
- Any page can be assigned to any free page frame
 - External fragmentation is eliminated
 - Internal fragmentation is at most a part of a page
- Possible page frame sizes are restricted to powers of 2 to simplify translation

Example of Paging

Suppose a new process, D, arrives needing 3 frames of memory

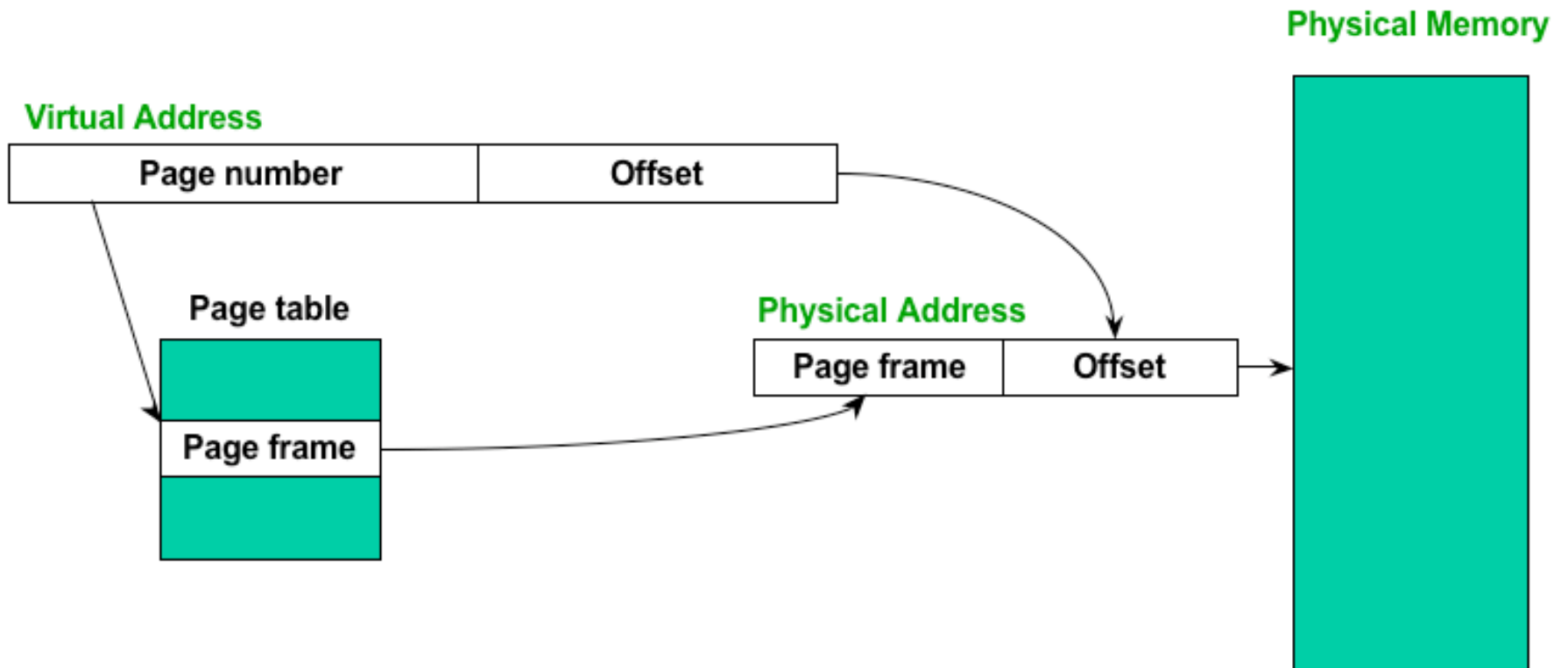


- We can fit Process D into memory, even though we don't have 3 contiguous frames available!

Support for Paging

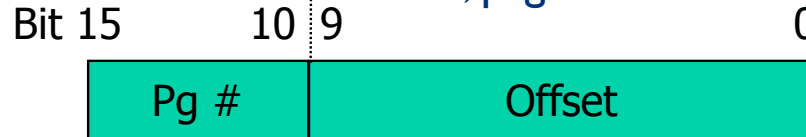
- Need more than base & limit registers now
- Operating system maintains a **page table** for each process
 - Page table records which physical frame holds each page
 - virtual addresses are now **page number + page offset**
 - $\text{page number} = \text{vaddr} / \text{page_size}$
 - $\text{page offset} = \text{vaddr} \% \text{page_size}$
 - On each memory reference, processor translates the page number to its frame number and adds the offset to generate a physical address

Paged Address Translation

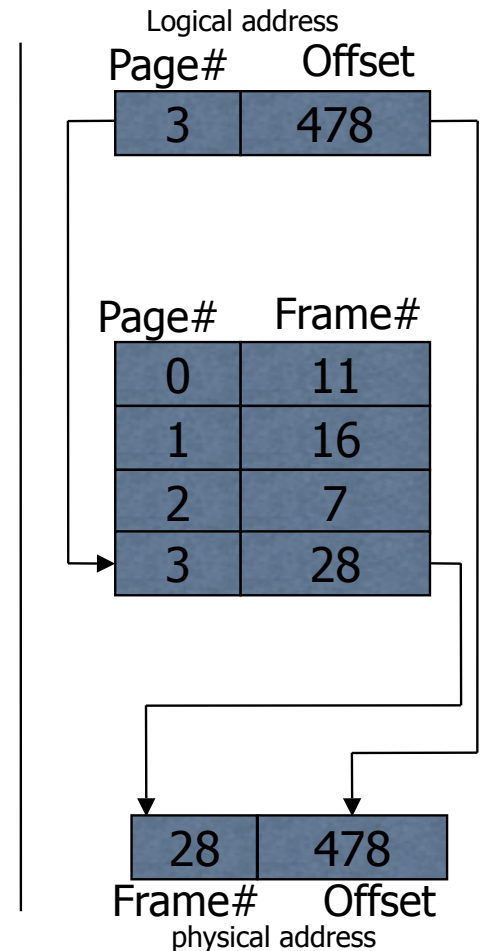


Example of Address Translation

- Suppose addresses are 16 bits, pages are 1024 bytes



- Least significant 10 bits of address provide offset within a page ($2^{10} = 1024$)
- Most significant 6 bits provide page number
- Maximum number of pages = $2^6 = 64$
- To translate virtual address: 0xDDE
 - Extract page number (high-order 6 bits)
-> $pg = vaddr \gg 10$ (== $vaddr / 1024$)
 - Get frame number from page table
 - Combine frame number with page offset
 - $offset = vaddr \& 0x3FF$ (== $vaddr \% 1024$)
 - $paddr = (frame \ll 10) | offset$



Where are Page Tables Stored?

- Simplest version
 - a linear array of entries, 1 entry per page
 - Stored in memory, attached to process
 - Virtual page number (VPN) is array index

```
struct addrspace {  
    paddr_t pgtbl;  
    ...  
}
```

```
struct addrspace *  
as_create(void) {  
    struct addrspace *as =  
        kmalloc(sizeof(struct addrspace));  
    int nentries = (unsigned)(-1) >> 13;  
    int npages = DIVROUNDUP(nentries*  
                             sizeof(pte_t), PAGE_SIZE);  
    as->pgtbl = getppages(npages);  
    ...  
}
```

Example: MIPS Page Table Entries

20	1	1	1	1	8
Page Frame Number	N	D	V	G	unused

- N == not cached
- D == dirty (meaning “writable”, not set by hardware)
- V == valid
- G == global (can be used by all processes)

- Maximum 2^{20} physical pages, each 4 kB
- Maximum 4GB of physical RAM

Limitation of Paging: Access Time

- Memory reference overhead is large
 - 2 references per address lookup (first page table, then actual memory)
 - Solution: use a hardware cache of lookups
- Translation Lookaside Buffer (TLB)
 - Small, fully-associative hardware cache of recently used translations
 - Part of the memory management unit (MMU)

Outline

- Introduction to Memory Management
 - Address Translation
- Memory Partitioning
- Relocation
- Paging
- Segmentation
- Virtual Memory

Segmentation

- Alternate means of dividing user program
- Divisions reflect the **logical organization** of the program
 - Text segment - read-only
 - Data segment - read/write, may be subdivided further
- Segments are variable-sized
 - A lot like dynamic partitioning, but a process may occupy multiple, non-contiguous segments
 - Suffers from external fragmentation
 - No simple mapping from logical to physical addresses

Address Translation

- Operating system maintains a **segment table**
 - Like the page table, but records start address and length for each segment
 - Physical start of segment need not be power-of-2
- Logical addresses consist of a segment # and an offset within that segment
 - For translation, may reserve a fixed number of high-order bits for segment number
 - Maximum segment size is determined by the number of bits left for the offset.
 - E.g., 16 bit address, 4 bit segment number = 16 segments of max size 4096 bytes ($2^{12} = 4096$)

The Main Drawback: Time

- 2 loads are required per address lookup (first into the page table, then for the requested address)
 - Loads are the most expensive operations!
 - Solution: use a hardware cache of lookups to remove the first load
- **Translation Lookaside Buffer (TLB)**
 - Small, fully-associative hardware cache of recently used translations
 - Part of the MMU

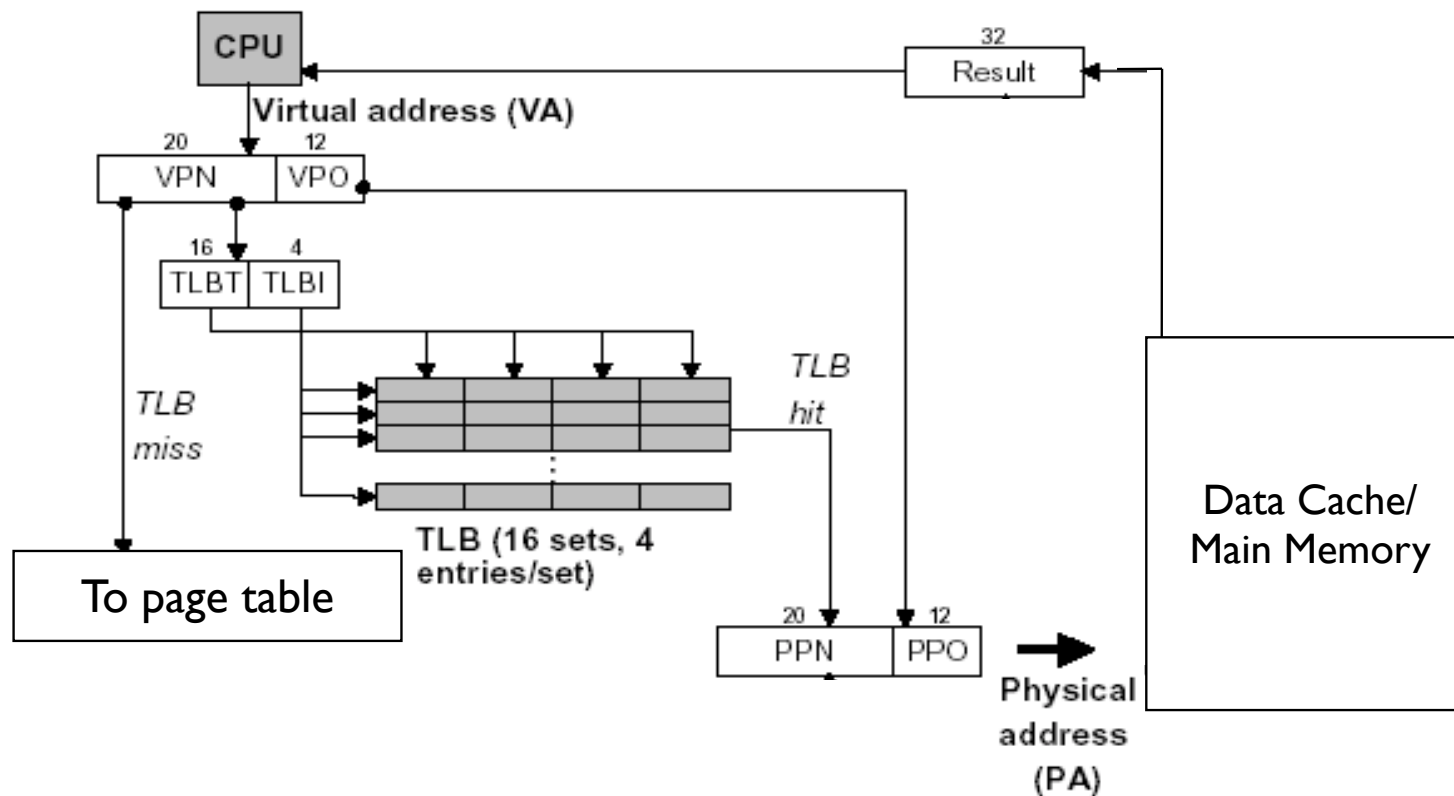
TLBs

- Given a virtual page number, returns the page's page table entry
 - The **tags** (indices) into the cache are virtual page numbers
 - With PTE + offset, can directly calculate physical address with existing hardware
- TLBs must return a value within a single machine cycle
 - This implies that the structure is **fully associative** (all entries are looked up in parallel)
 - Fully associative structures are very space-expensive
 - The TLB cannot be very large

Why do TLBs work?

- TLBs (like all caches) exploit **locality**
 - Locality is the idea that programmers tend to:
 - Access an item multiple times in a short period of time (**temporal locality**)
 - Access items close to an already accessed item (**spatial locality**)
- Processes only use a handful of pages at a time
 - 16-48 entries/pages (64-192K)
 - Only need those pages to be “mapped”
 - **Hit rates are therefore very important**
 - If you have a large, sprawling structure in the heap, you get what you deserve

Example: Pentium



TLBs, Common Case

- Situation: Process is executing on the CPU, and it issues a **read** to an address
- The address 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
 4. PTE specifies which **physical frame** holds the page
 5. MMU combines physical frame & offset into a **physical address**
 6. MMU reads from that physical addr, returns value to CPU

TLB Misses

- At this point, two other things can happen
 1. TLB does not have a PTE mapping this virtual address
 2. PTE exists, but memory access violates PTE protection bits
- Translations for most instructions are handled using the TLB
 - >99% of translations hit, but there are misses (TLB miss or TLB fault)...
 - On a miss, the missed entry needs to be loaded into the TLB
- The big question: Who loads (places translations into) the TLB?

Reloading the TLB

- If the TLB does not have mapping, two possibilities:
 - 1. MMU loads PTE from page table in memory
 - Hardware managed TLB, OS not involved in this step
 - OS has already set up the page tables so that the hardware can access it directly
 - 2. Trap to the OS
 - Software managed TLB, OS intervenes at this point
 - OS does lookup in page table, loads PTE into TLB
 - OS returns from exception, TLB continues
- Most machines will only support one method or the other
- At this point, there is a PTE for the address in the TLB

Software-Managed TLB

```
struct region {
    vaddr_t vbase;
    int len;
    pte_t *table;
}

struct addrspace {
    array of region
}
```

```
int
as_define_region(vaddr_t vbase, int len,...)
{
    struct region *r = (struct region *)
        kmalloc(sizeof(struct region));
    r->vbase = vbase;
    r->len = len;
    int nentries = DIVROUNDUP(len, PAGE_SIZE);
    r->table = (pte_t *)kmalloc(nentries *
        sizeof(pte_t));
    add r to array of regions in addrspace
}
```

- TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
- Must be implemented to be fast (but still 20-200 cycles)
- CPU ISA has instructions for manipulating TLB
- Tables can be in any format convenient for OS (flexible)

Software Management

- OS ensures that TLB and page tables are consistent
 - When a PTE is modified, if it is in the TLB, it must be invalidated there (valid is a bit flag in the PTE)
- On a context switch, the TLB must be reloaded
 - Why?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
 - Choosing PTE to evict is called the **TLB replacement policy**
 - Sometimes, the eviction policy is implemented in hardware

One Last Problem ...

- Page table lookup (by HW or OS) can cause a recursive fault if the page table is paged out
 - This assumes page tables are in OS virtual address space
- These (and issues with the protection bits in PTEs) become **page faults**
 - Page faults trap to the OS, where the fault is handled in software
 - Example: Virtual page not allocated in address space
 - OS sends fault to process (e.g., segmentation fault)
 - Example: Page not in physical memory
 - OS allocates frame and reads it in
 - Sometimes, the OS will use the page fault mechanism for other services (copy on write or mapped files, for example)

Summary

- The major drawback to paging was the increased cost of a lookup
- We use a small hardware cache called the TLB to remove much of the overhead of a page table lookup
 - TLBs can be managed by hardware or software
 - Software management of the TLB is a major portion of A2
- Occasionally, the OS must be invoked to handle cases where the PTE does not have a valid entry or the page table itself is paged out