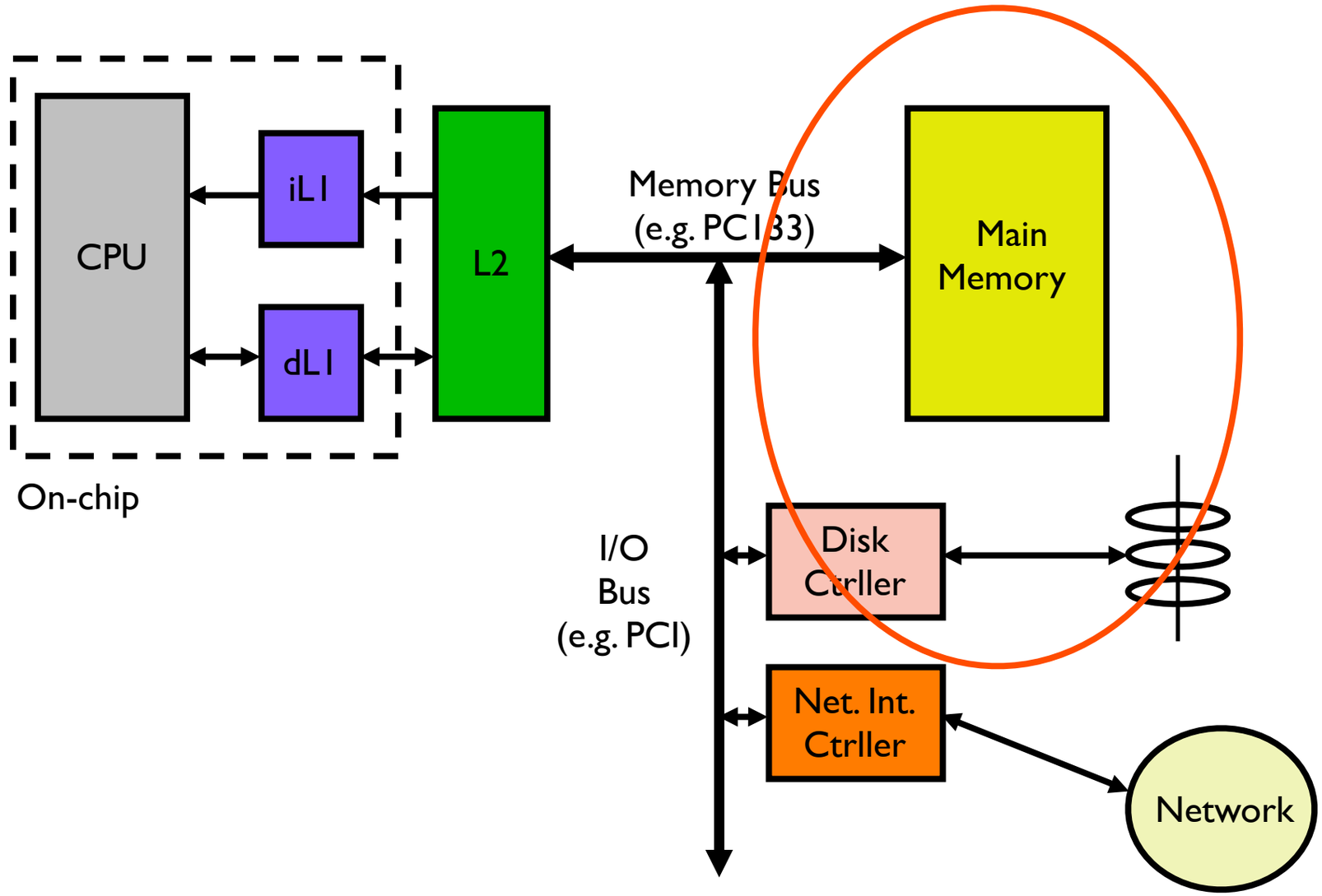
UNIVERSITY OF OREGON

CIS 415:
Operating Systems
Memory Management

Spring 2014
Prof. Kevin Butler


CPU
iL1
dL1
L2
On-chip
Memory Bus
(e.g. PC133)
Main
Memory
I/O
Bus
(e.g. PCI)
Disk
Ctrller
Net. Int.
Ctrller
Network

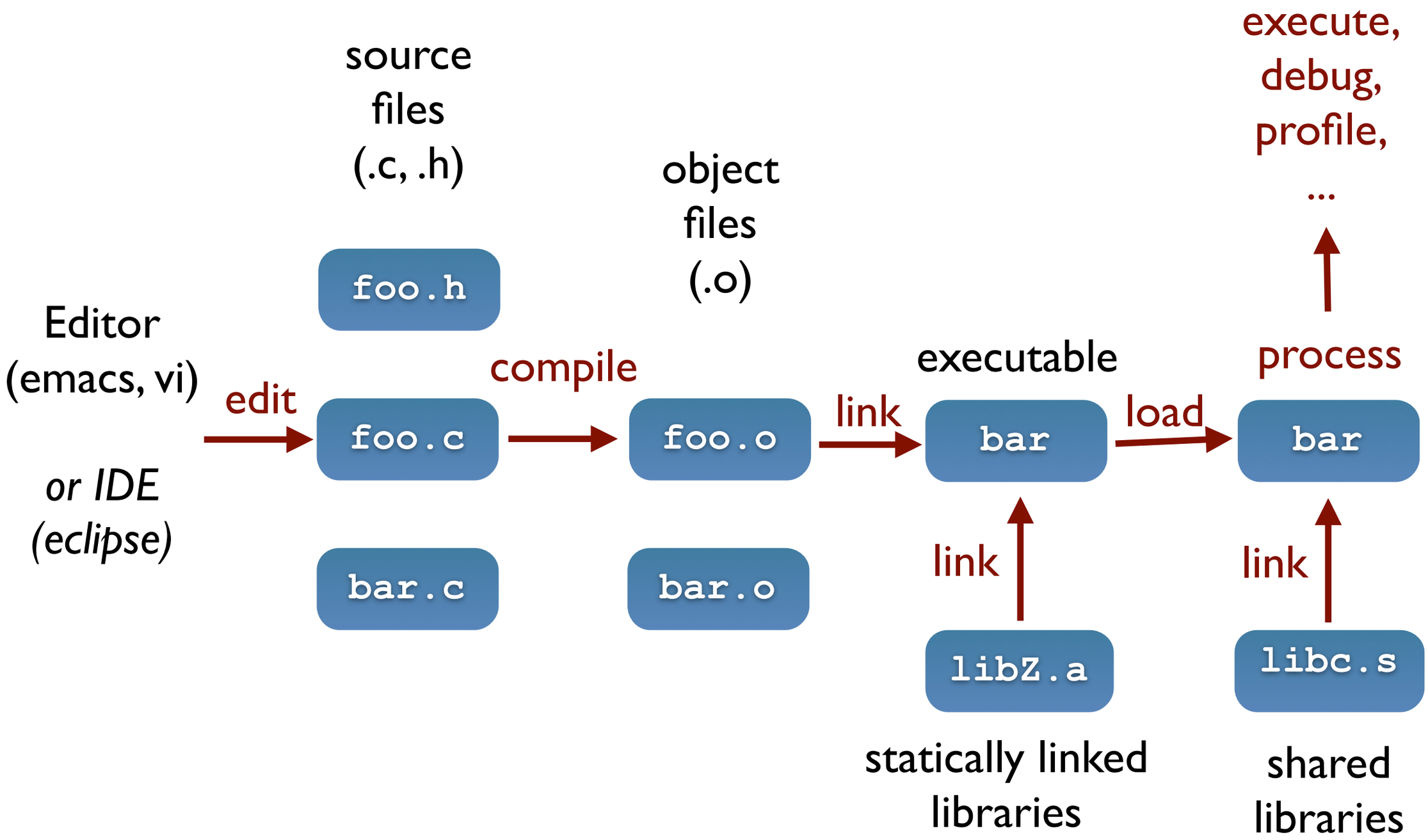
Binding Programs to Addresses

UNIVERSITY OF OREGON

- Address binding of instructions and data to memory addresses can happen at three different stages
  - *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)

C workflow

UNIVERSITY OF OREGON



From C to machine code

UNIVERSITY OF OREGON

C source file (dosum.c)

```
int dosum(int i, int j) {
  return i+j;
}
```

C compiler (gcc -S)

assembly source file (dosum.s)

```
dosum:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %eax
    addl    8(%ebp), %eax
    popl    %ebp
    ret
```

assembler (as)

machine code (dosum.o)

```
80483b0: 55
89 e5 8b 45
0c 03 45 08
   5d c3
```

Multi-file C programs

UNIVERSITY OF OREGON

C source file (dosum.c)

```
int dosum(int i, int j) {
  return i+j;
}
```

this "prototype" of dosum( ) tells gcc about the types of dosum's arguments and its return value

C source file (sumnum.c)

```
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

dosum( ) is implemented in sumnum.c

Multi-file C programs

C source file
(dosum.c)

```
int dosum(int i, int j) {
  return i+j;
}
```

C source file
(sumnum.c)

```
#include <stdio.h>

int dosum(int i, int j);

int main(int argc, char **argv) {
  printf("%d\n", dosum(1,2));
  return 0;
}
```

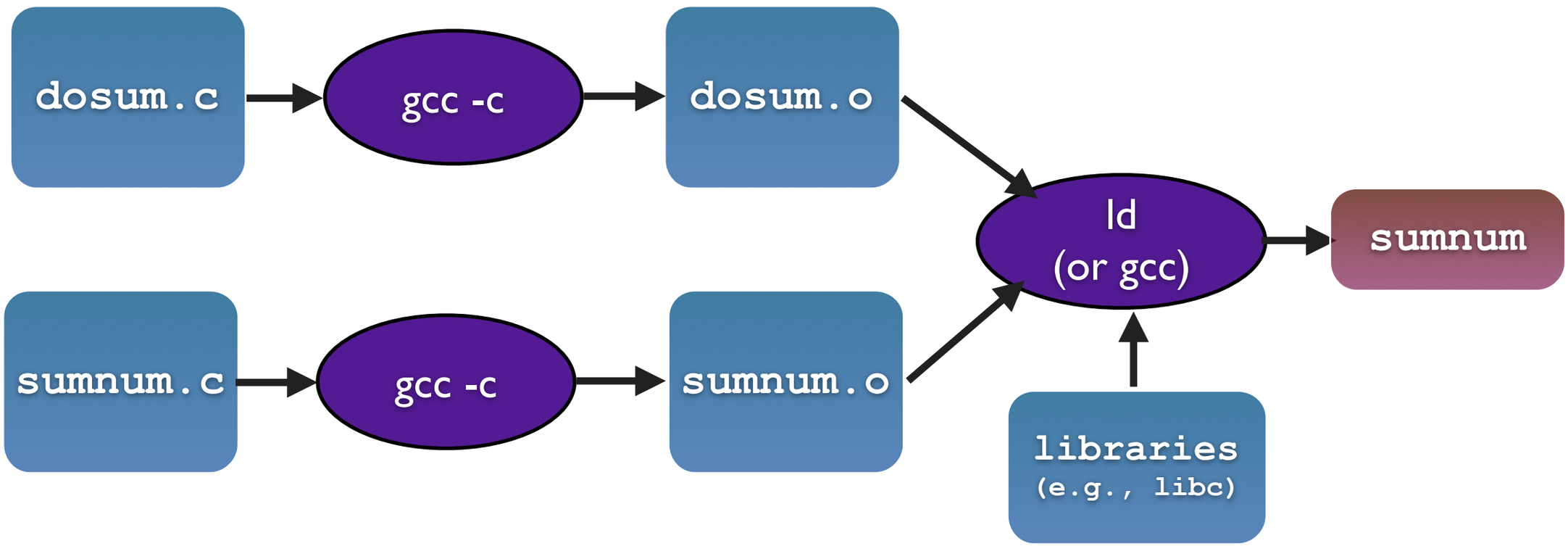
why do we need this #include?

where is the implementation of printf?

Compiling multi-file programs

UNIVERSITY OF OREGON

- Multiple object files are ***linked*** to produce an executable
  - standard libraries (libc, crt1, ...) are usually also linked in
  - a library is just a pre-assembled collection of .o files



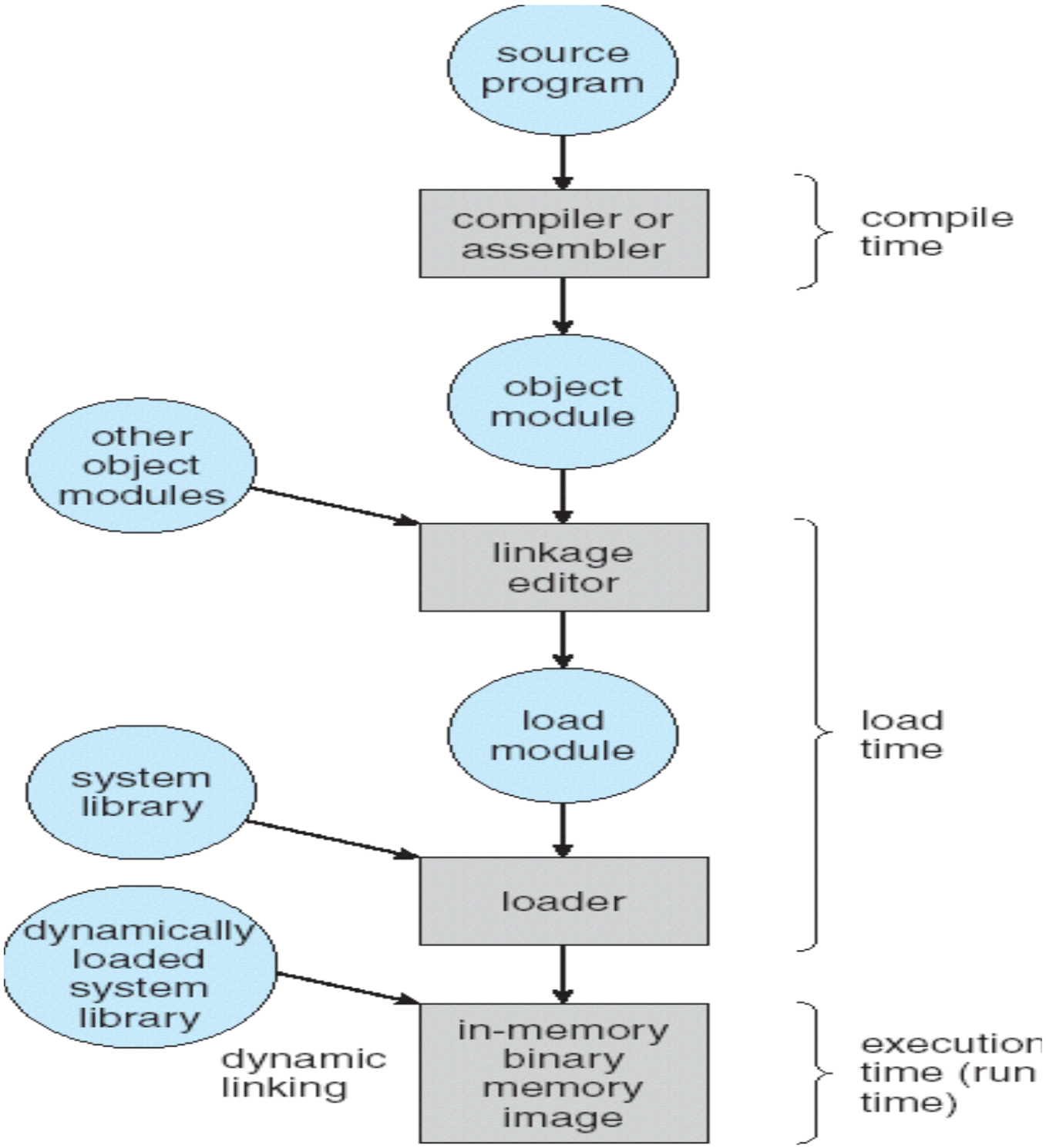
Object files

UNIVERSITY OF OREGON

- sumnums, dosum.o are **object files**
  - each contains machine code produced by the compiler
  - each might contain references to external symbols
    - variables and functions not defined in the associated .c file
      - e.g., sumnum.o contains code that relies on printf( ) and dosum( ), but these are defined in libc.a and dosum.o, respectively
  - linking resolves these external symbols while smooshing together object files and libraries

Loading User Programs

UNIVERSITY OF OREGON

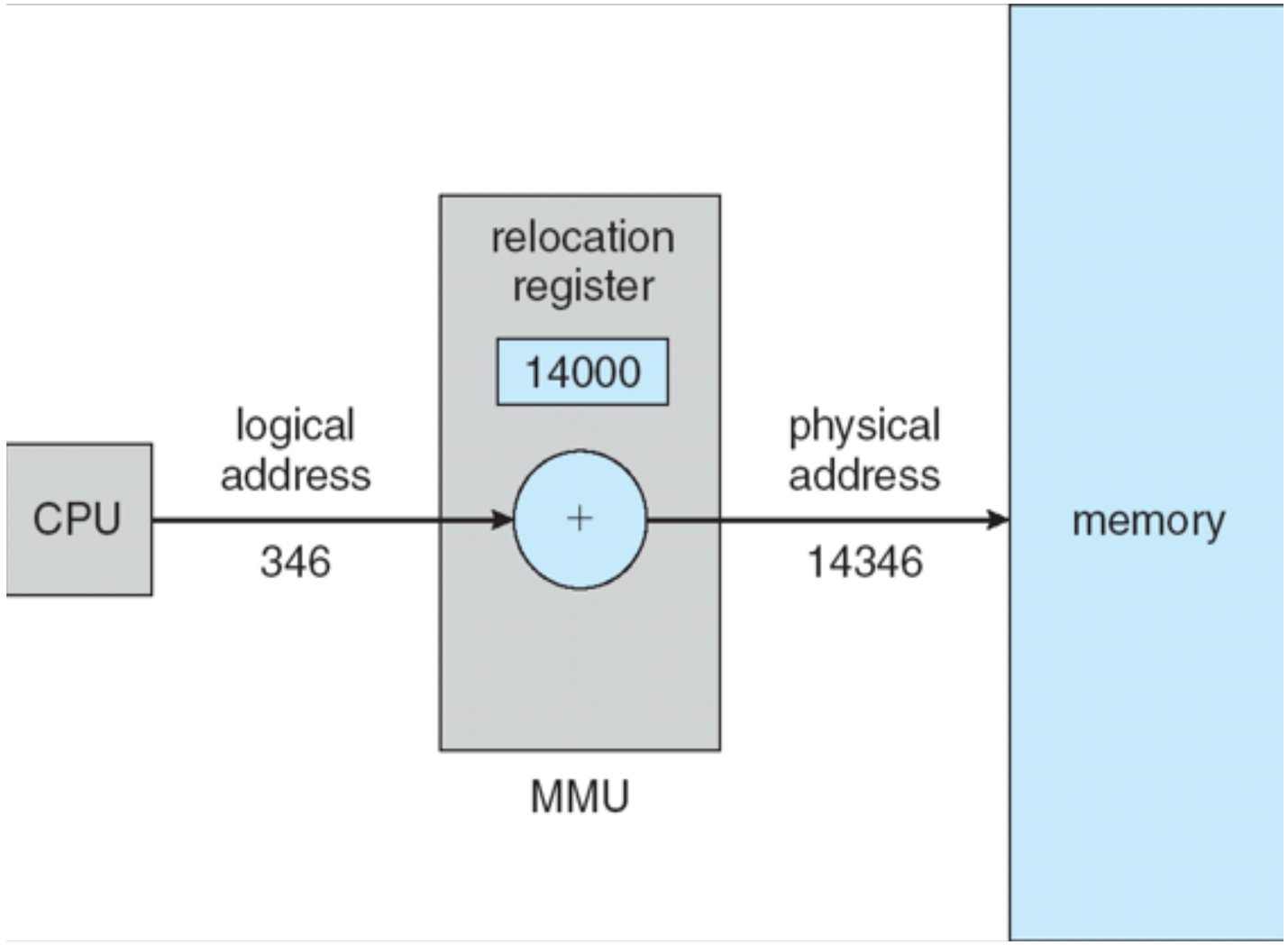


Logical & Physical Addresses

UNIVERSITY OF OREGON

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as virtual address
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

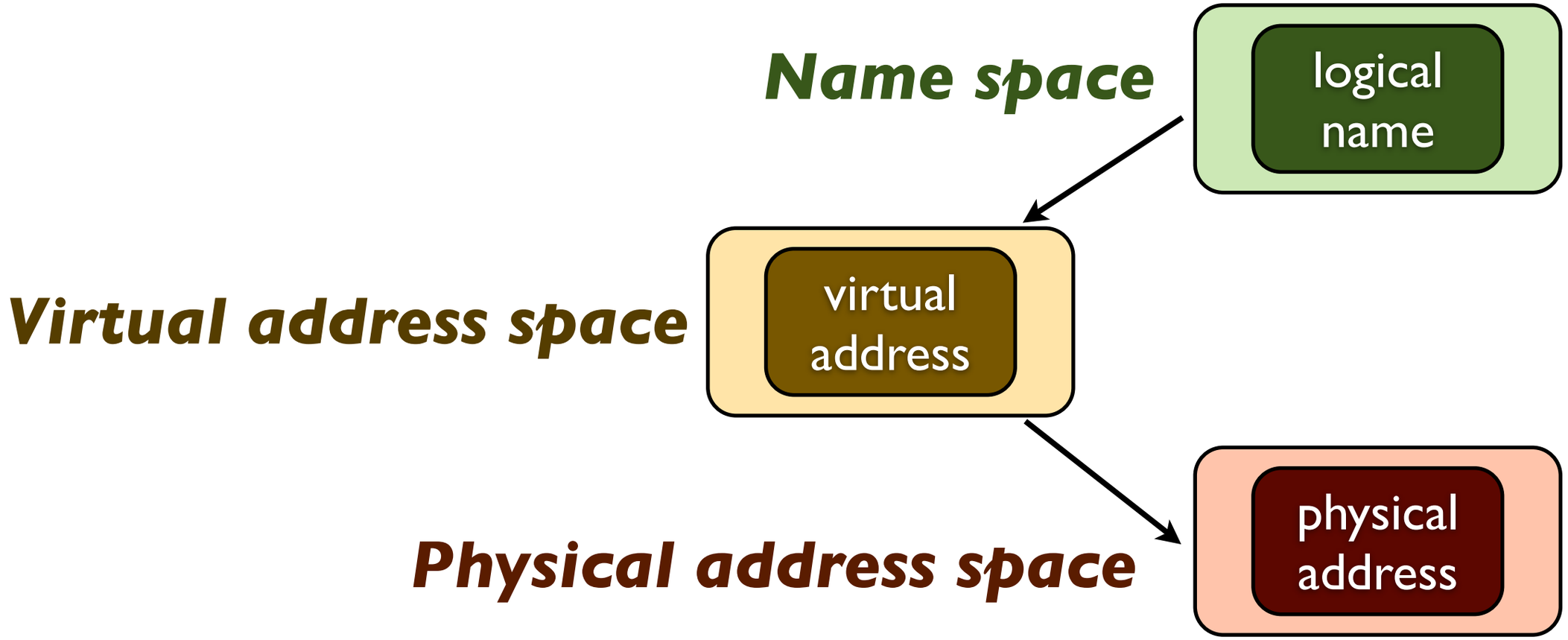
Memory Management Unit



Need for Memory Management

UNIVERSITY OF OREGON

- Physical memory (DRAM) is limited
  - A single process may not all fit in memory
  - Several processes (their address spaces) also need to fit at the same time



Swapping

UNIVERSITY OF OREGON

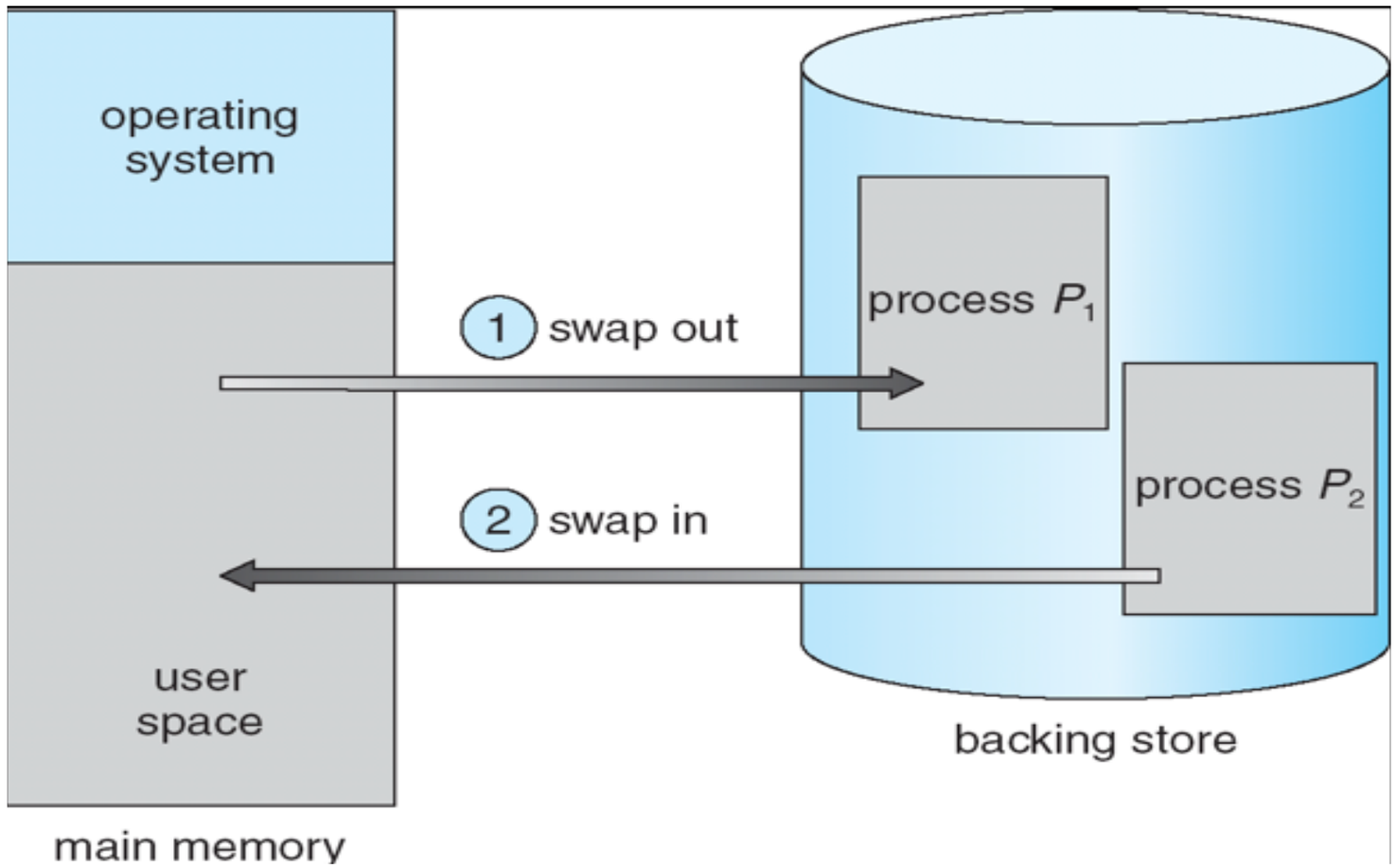
- If something (either part of a process, or multiple processes) does not fit in memory, then it has to be kept on disk.
- Whenever that needs to be used by the CPU (to fetch the next instruction, or to read/write a data word), it has to be brought into memory from disk.
- Consequently, something else needs to be evicted from memory.
- Such transfer between memory and disk is called *swapping*.
- Disallow 1 process from accessing another process's memory.

Swap Space

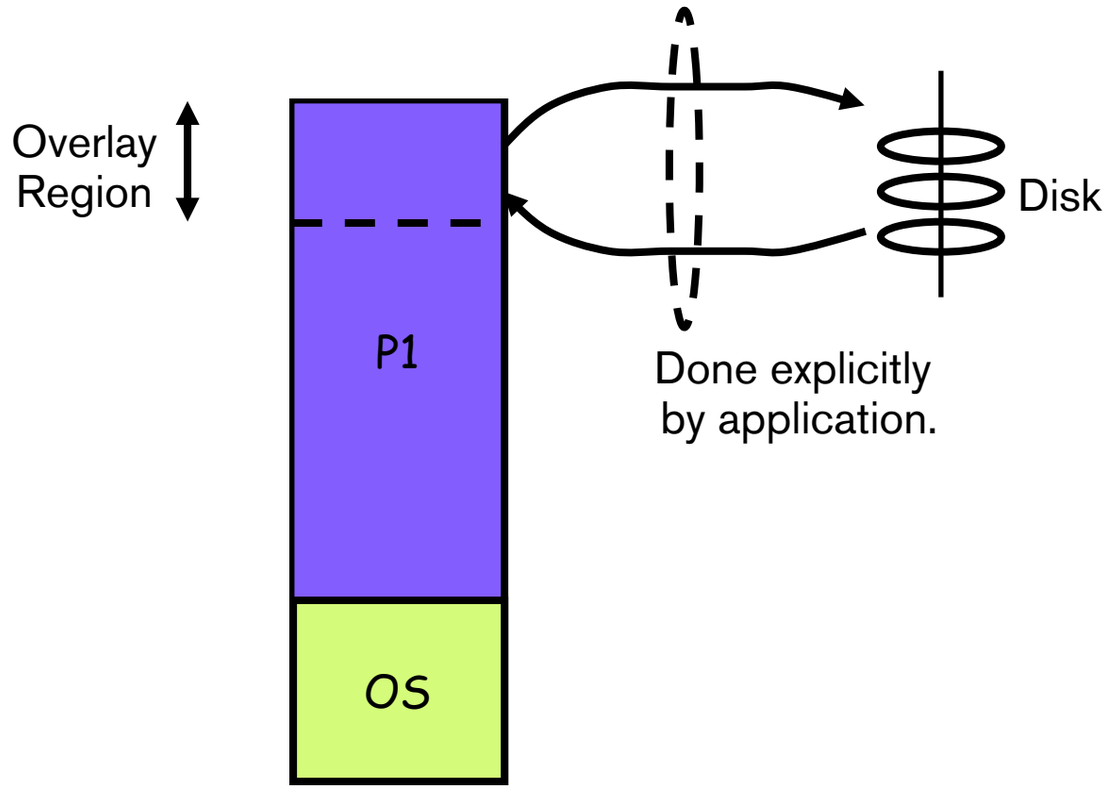
UNIVERSITY OF OREGON

- Usually a separate portion of the disk is reserved for swapping, that is referred to as *swap space*.
- Note that swapping is different from any explicit file I/O (read/write) that your program may contain.
- Typically swapping is transparent to your program.

Swapping


operating system
process P_1
1 swap out
process P_2
2 swap in
user space
backing store
main memory

Early Days: Overlays

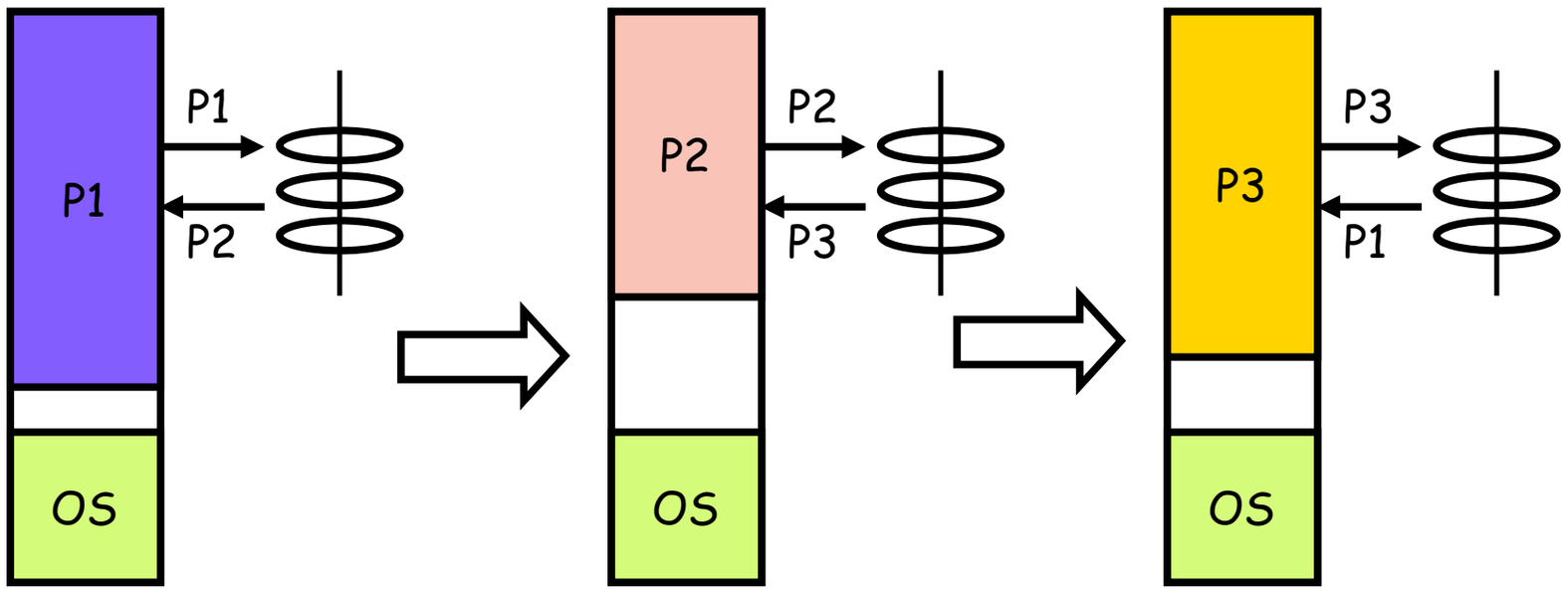


In this case, even a single application process does not fit entirely in memory

Swapping

UNIVERSITY OF OREGON

Even if a process fits entirely in memory, we do not want to do the following …



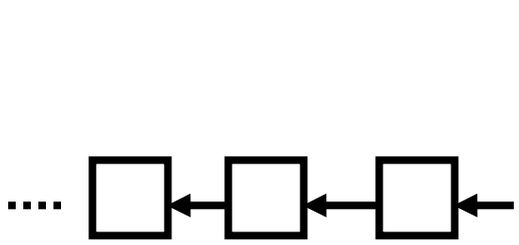
Context switching will be highly inefficient, and it defeats the purpose of multiprogramming.

Multiprogramming

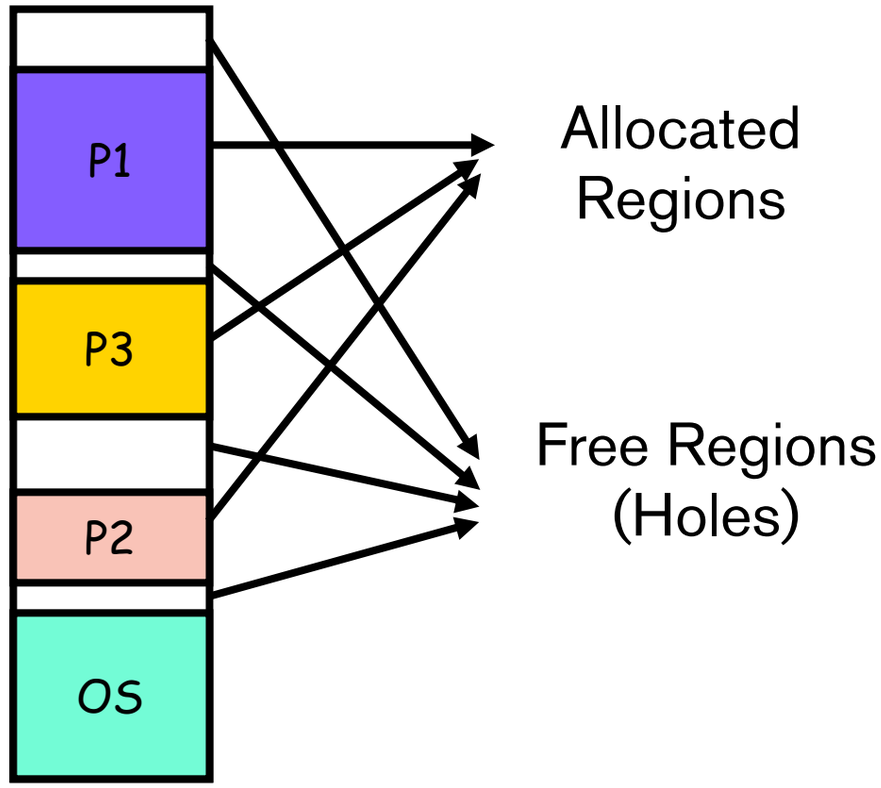
UNIVERSITY OF OREGON

- Say your program does explicit file I/O (read/write) for a fraction *f* of its execution time, then with *p* processes, CPU efficiency = (1 – *f*^*p*)
- To maintain high CPU efficiency, we need to increase *p*.
- But as we just saw, these processes cannot all be on disk. We need to keep as many of these processes in memory as possible.
- So even if we are not keeping all of the process, keep the essential parts of as many processes as possible in memory.
- We will get back to this issue at a later point!

Memory Allocation



Queue of waiting requests/jobs



Question: How do we perform this allocation?

Goals

UNIVERSITY OF OREGON

- Allocation() and Free() should be fairly efficient

- Should be able to satisfy more requests at any time (i.e. the sum total of holes should be close to 0 with waiting requests).

Solution Strategies

UNIVERSITY OF OREGON

- Contiguous allocation
  - The requested size is granted as 1 big contiguous chunk.
  - E.g. first-fit, best-fit, worst-fit, buddy-system.
- Non-contiguous allocation
  - The requested size is granted as several pieces (and typically each of these pieces is of the same – fixed - size).
  - E.g., paging

Contiguous Allocation

UNIVERSITY OF OREGON

- Data structures:
  - Queue of requests of different sizes
  - Queues of allocated regions and holes.
- Find a hole and make the allocation (and it may result in a smaller hole).
- Eventually, you may get a lot of holes that become small enough that they cannot be allocated individually.
- This is called *external fragmentation*.

Fragmentation

UNIVERSITY OF OREGON

- *External Fragmentation* – free space between allocated memory regions
- *Internal Fragmentation* – free space within an allocated region
  - allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time

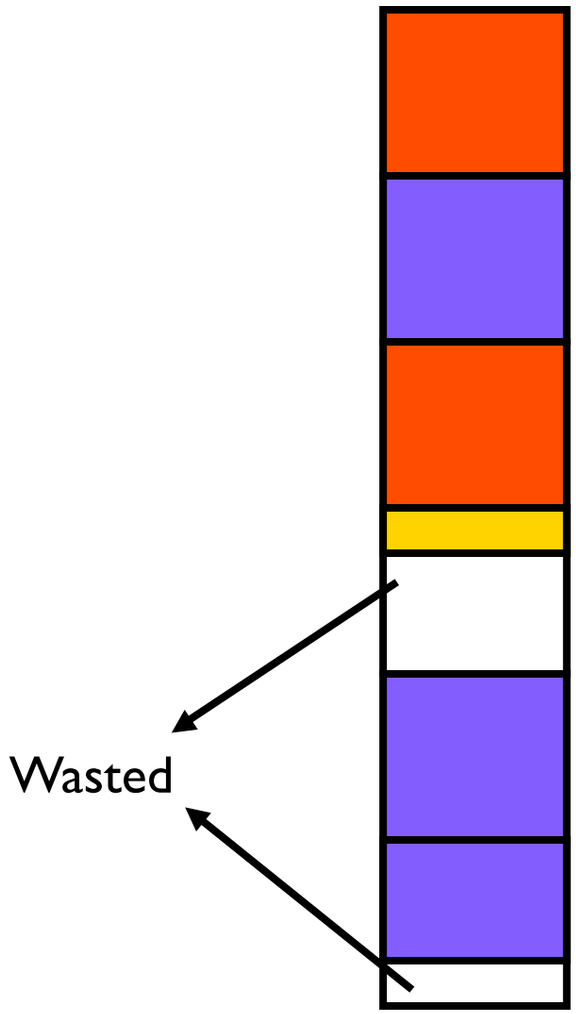
Internal Fragmentation

UNIVERSITY OF OREGON

- Partitioned allocation may result in very small fragments
  - Assume allocation of 126 bytes
  - Use 128 byte block, but 2 bytes left over
- Maintaining a 2-byte fragment is not worth it, so just allocate all 128 bytes
  - But, 2 bytes are unusable
  - Called *internal fragmentation*

Non-contiguous Allocation

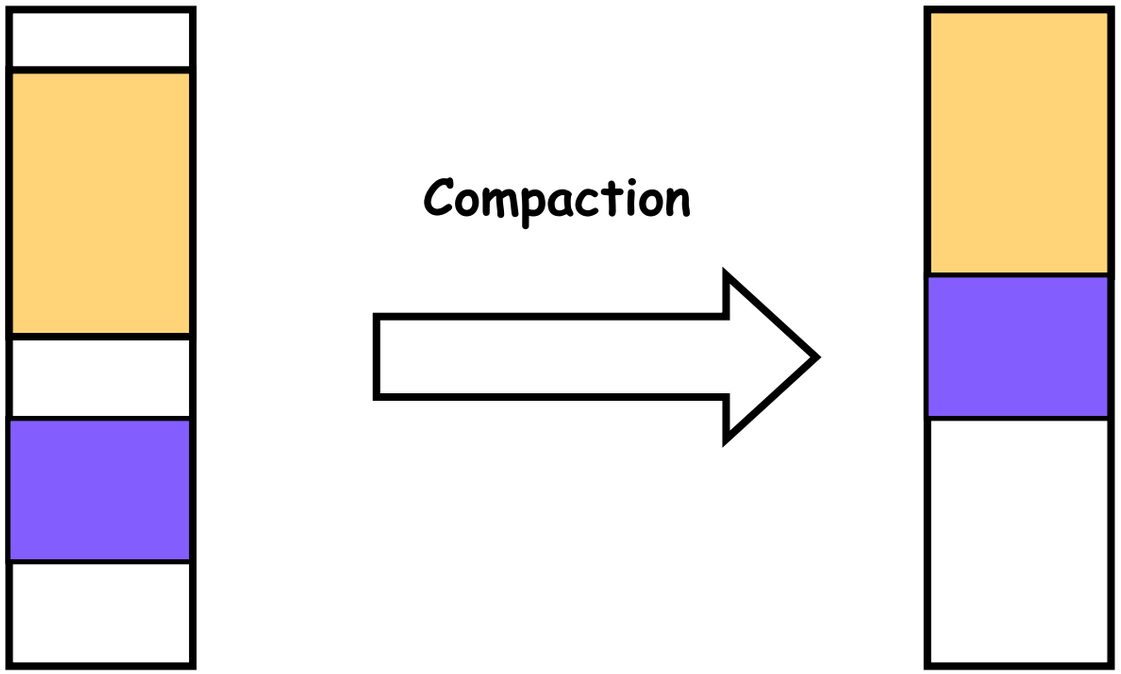
UNIVERSITY OF OREGON



This can result in more

Internal Fragmentation

Easing External Fragmentation



Note that this can be done only with relocatable code and data (use indirect/indexed/relative addressing)

But compaction is expensive and we want to do this as infrequently as possible.

Contiguous Allocation

UNIVERSITY OF OREGON

- Which hole to allocate for a given request?
- First-fit
  - Search through the list of holes. Pick the first one that is large enough to accommodate this request.
  - Though allocation may be easy, it may not be very efficient in terms of fragmentation.

Allocation Strategies

UNIVERSITY OF OREGON

- Best Fit
  - Search through the entire list to find the smallest hole that can accommodate the given request.
  - Requires searching through the entire list (or keeping it in sorted order).
  - This can actually result in very small sized holes making it undesirable.

Allocation Strategies

UNIVERSITY OF OREGON

- Worst fit
  - Pick the largest hole and break that.
  - The goal is to keep the size of holes as large as possible.
  - Allocation is again quite expensive (searching through entire list or keeping it sorted).

Freeing Memory

UNIVERSITY OF OREGON

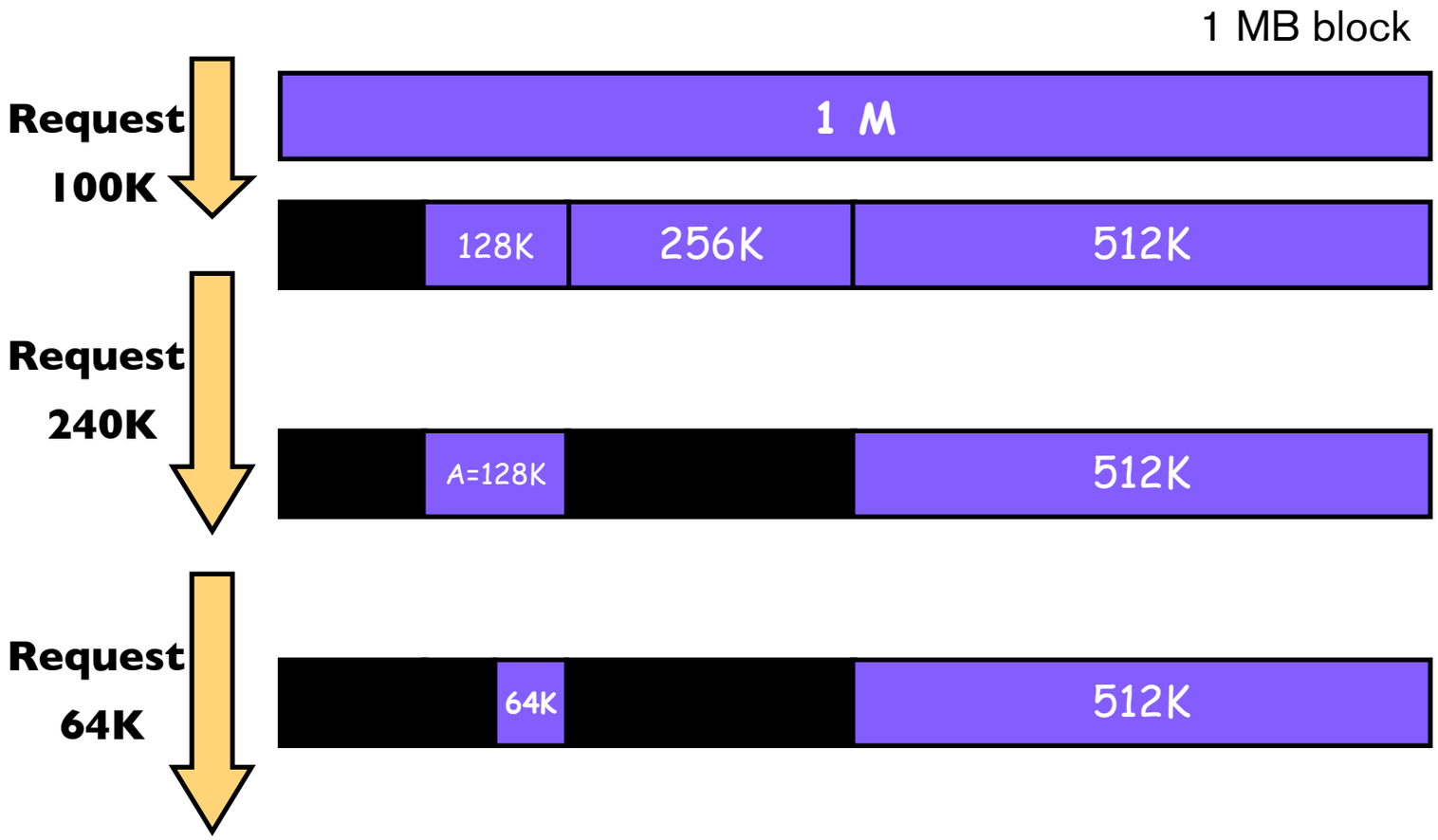
- You need to check whether nearby regions (on either side) are free, and if so you need to make a larger hole.
- This requires searching through the entire list (or at least keeping the holes sorted in address order).

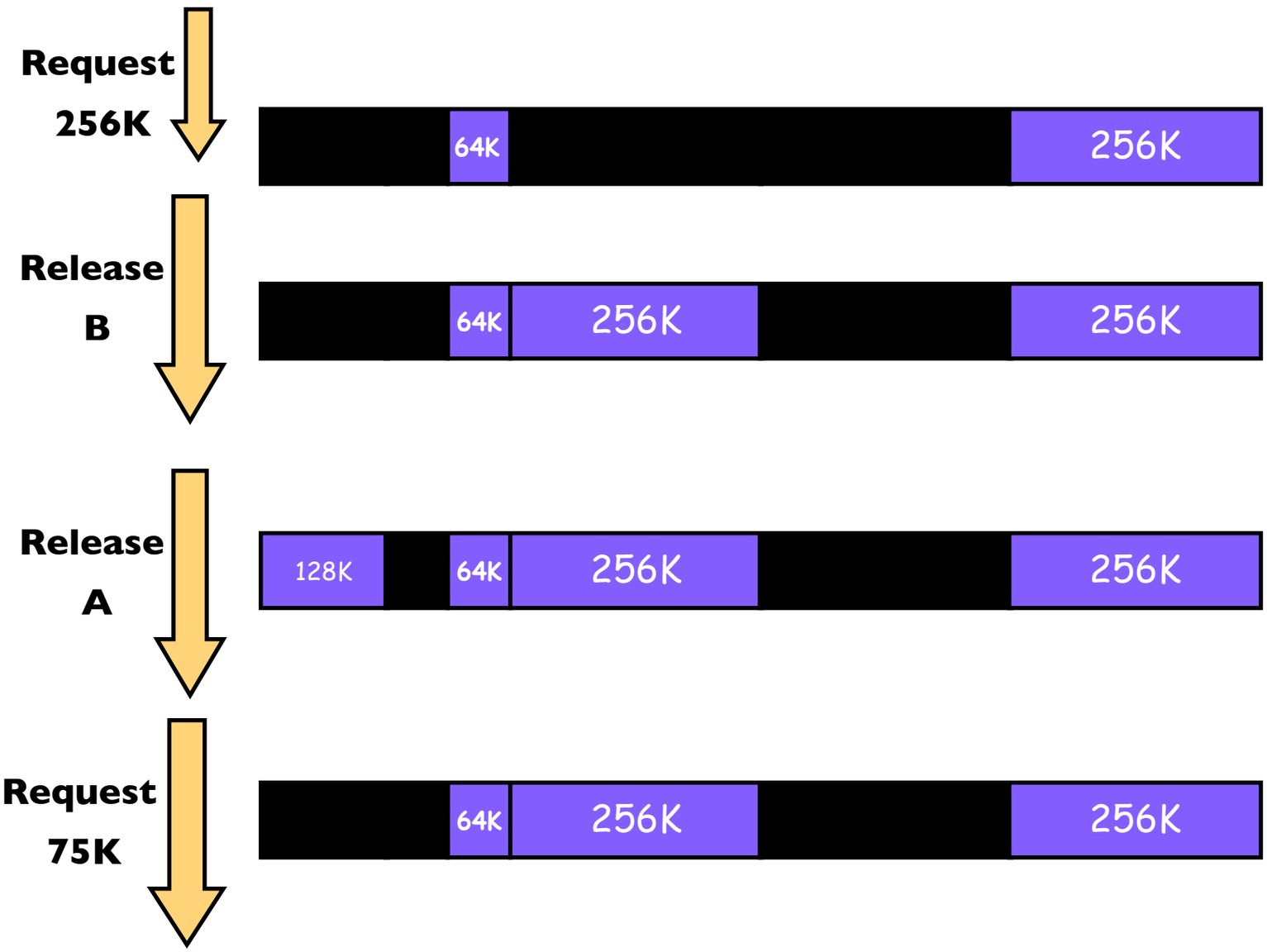
Costs

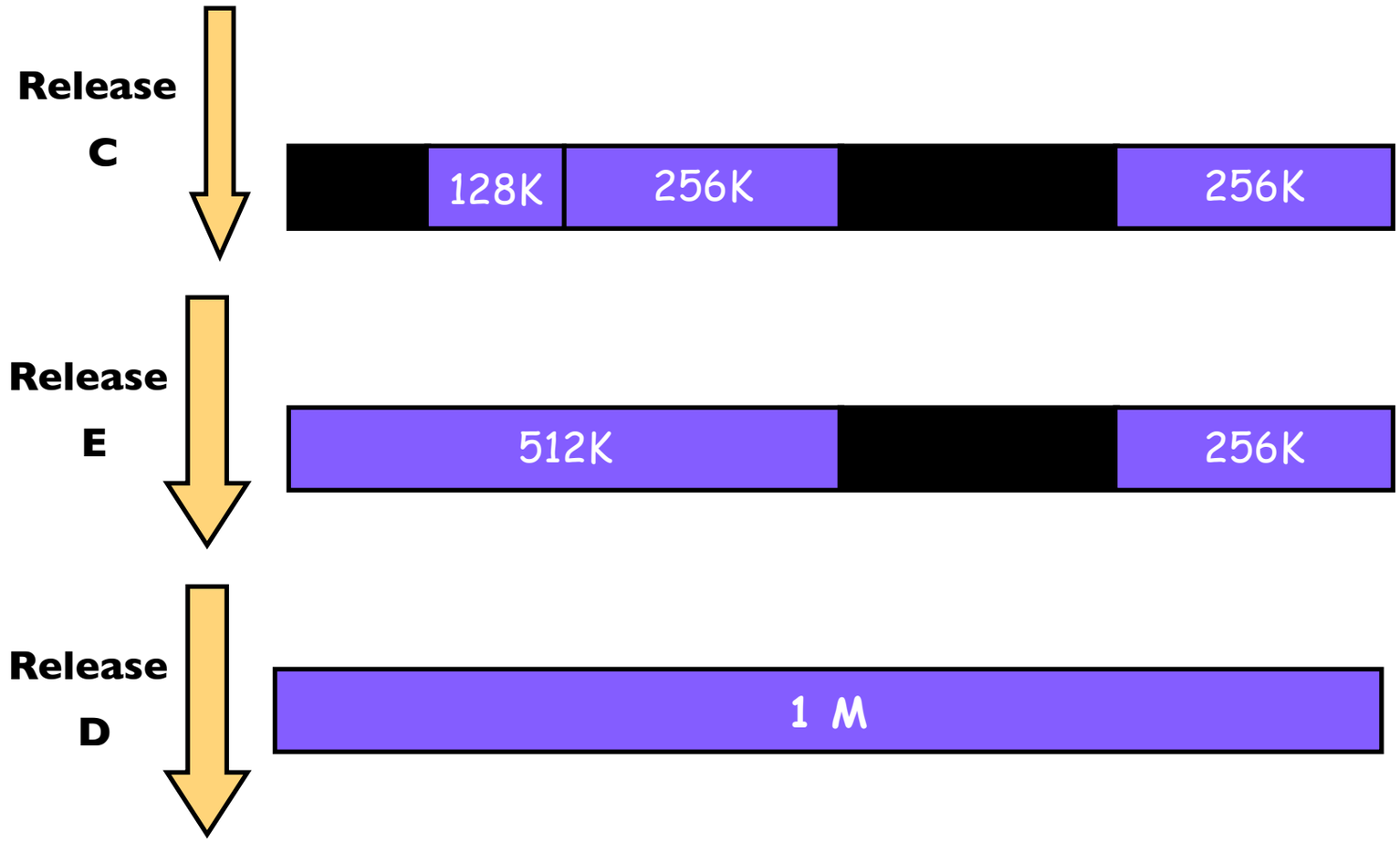
UNIVERSITY OF OREGON

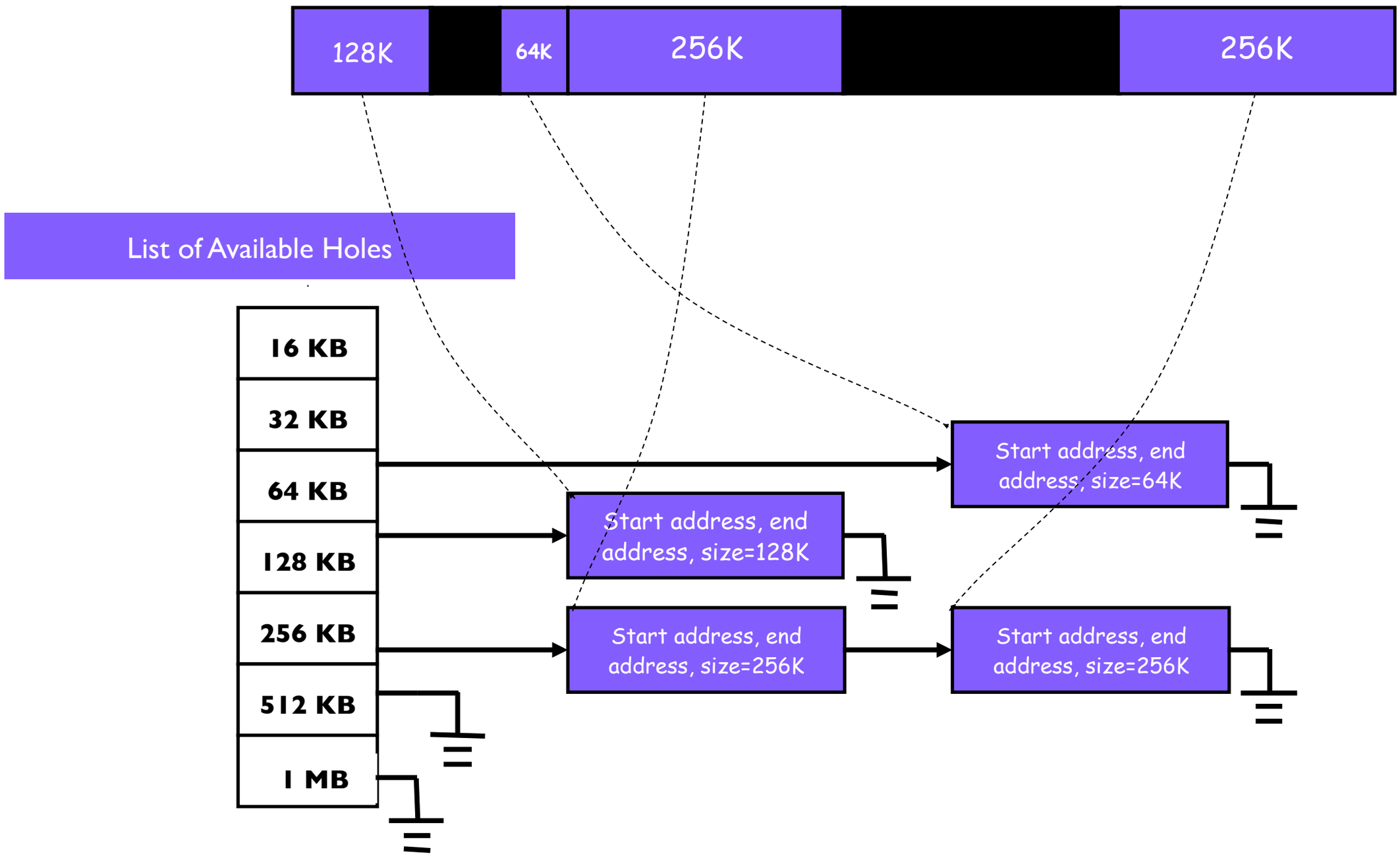
- Allocation: Keeping list in sorted size order or searching entire list each time (O(N)).
- Free: Keeping list in sorted address order or searching entire list each time (O(N)).
- Alternative used in Linux: the *buddy system*
  - Group free page frames into 11 block lists divided into power-of-2 blocks (1, 2, 4, 8, ... 1024)
  - Checks whether free blocks in each list in increasing order
  - Allocation/free become O(log(N)) operations

An Example


1 MB block
Request
100K
1 M
128K
256K
512K
Request
240K
A=128K
512K
Request
64K
64K
512K


Request
256K
64K
256K
Release
B
64K
256K
256K
Release
A
128K
64K
256K
256K
Request
75K
64K
256K
256K


Release
C
128K
256K
256K
Release
E
512K
256K
Release
D
1 M

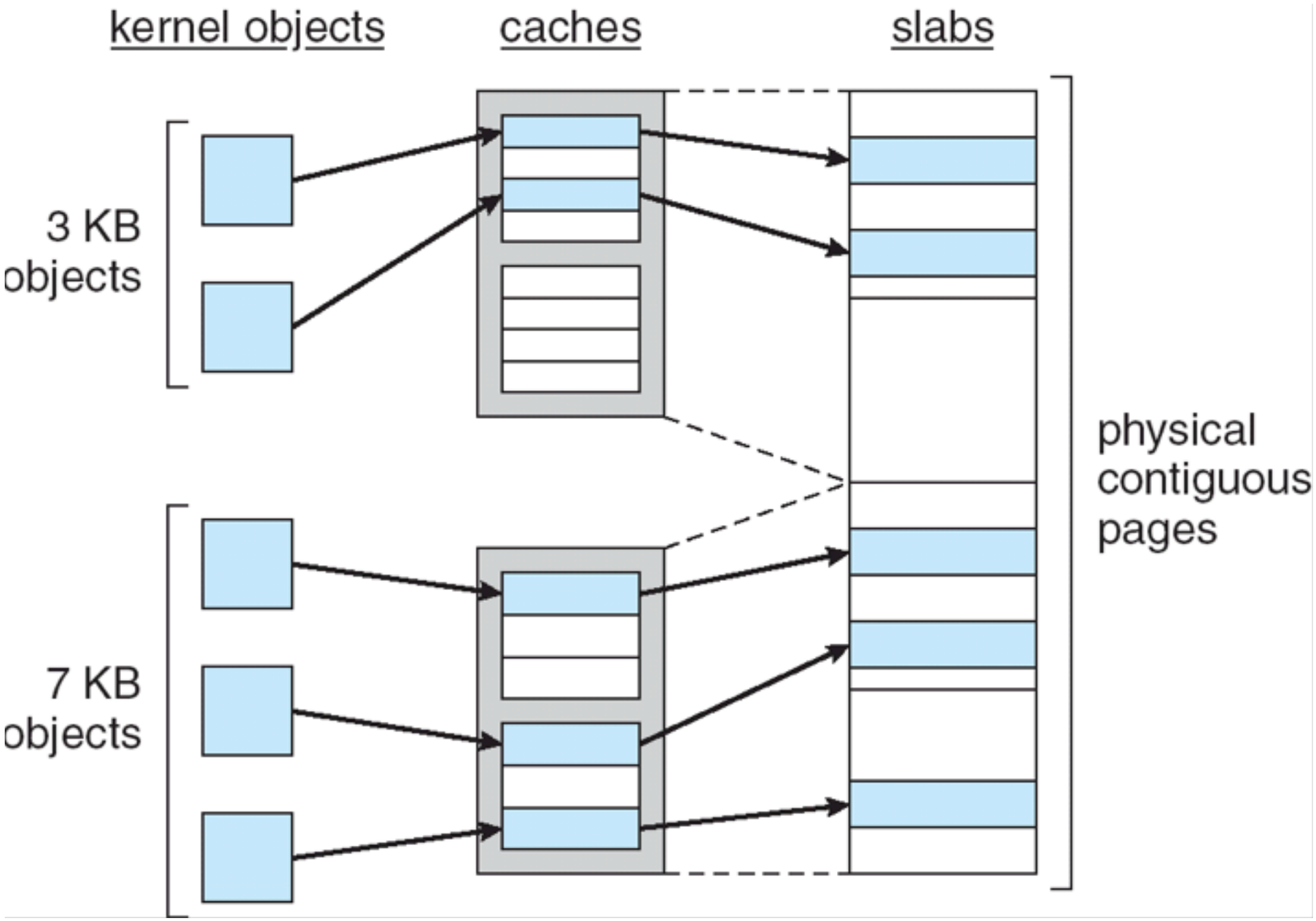

128K
64K
256K
256K
List of Available Holes
16 KB
32 KB
64 KB
128 KB
256 KB
512 KB
1 MB
Start address, end address, size=64K
Start address, end address, size=128K
Start address, end address, size=256K
Start address, end address, size=256K

Slab Allocator

UNIVERSITY OF OREGON

- *Slab* is one or more physically contiguous "pages"
- Cache consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with objects – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Memory Mgmt. Basics

UNIVERSITY OF OREGON

- Allocation
  - Previously
    - Allocate arbitrary-sized chunks (e.g., in old days, a process; now arbitrary allocation done on the heap)
  - Challenges
    - Fragmentation and performance
- Swapping
  - Need to use the disk as a backing store for limited physical memory
  - Problems
    - Complex to manage backing of arbitrary-sized objects
    - May want to work with subset of process (later)

Paging

UNIVERSITY OF OREGON

- Programs are provided with a virtual address space (say 1 MB).
- Role of the OS to fetch data from either physical memory or disk.
  - Done by a mechanism called *(demand) paging*.
- Divide the virtual address space into units called *"virtual pages"* each of which is of a fixed size (usually 4K or 8K).
  - For example, 1M virtual address space has 256 4K pages.
- Divide the physical address space into "physical pages" or "frames".
  - For example, we could have only 32 4K-sized pages.

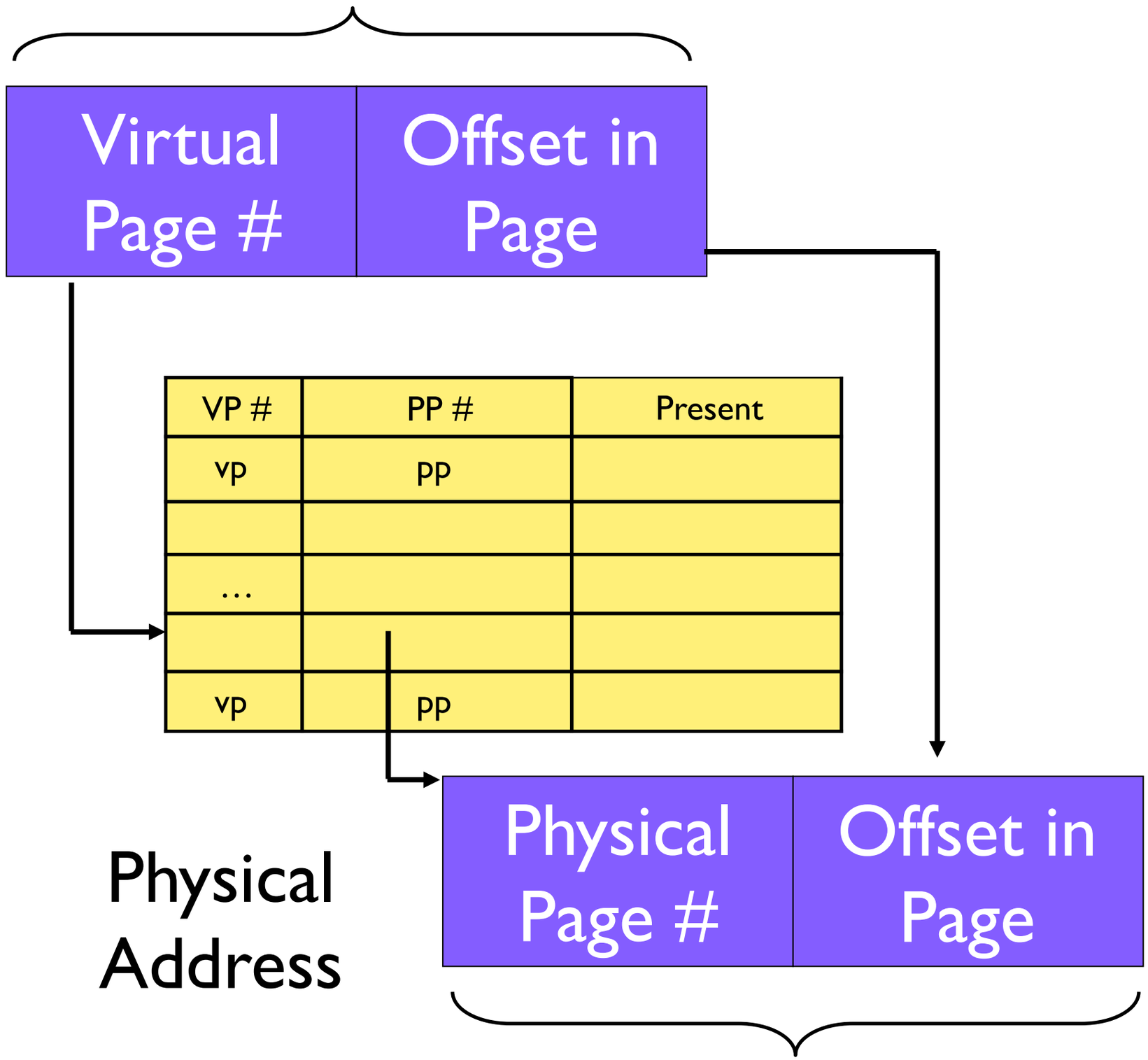
Page Table

UNIVERSITY OF OREGON

- Role of the OS to keep track of which virtual page is in physical memory and if so where?
  - Maintained in a data structure called "page-table" that the OS builds.
  - "Page-tables" map Virtual-to-Physical addresses.

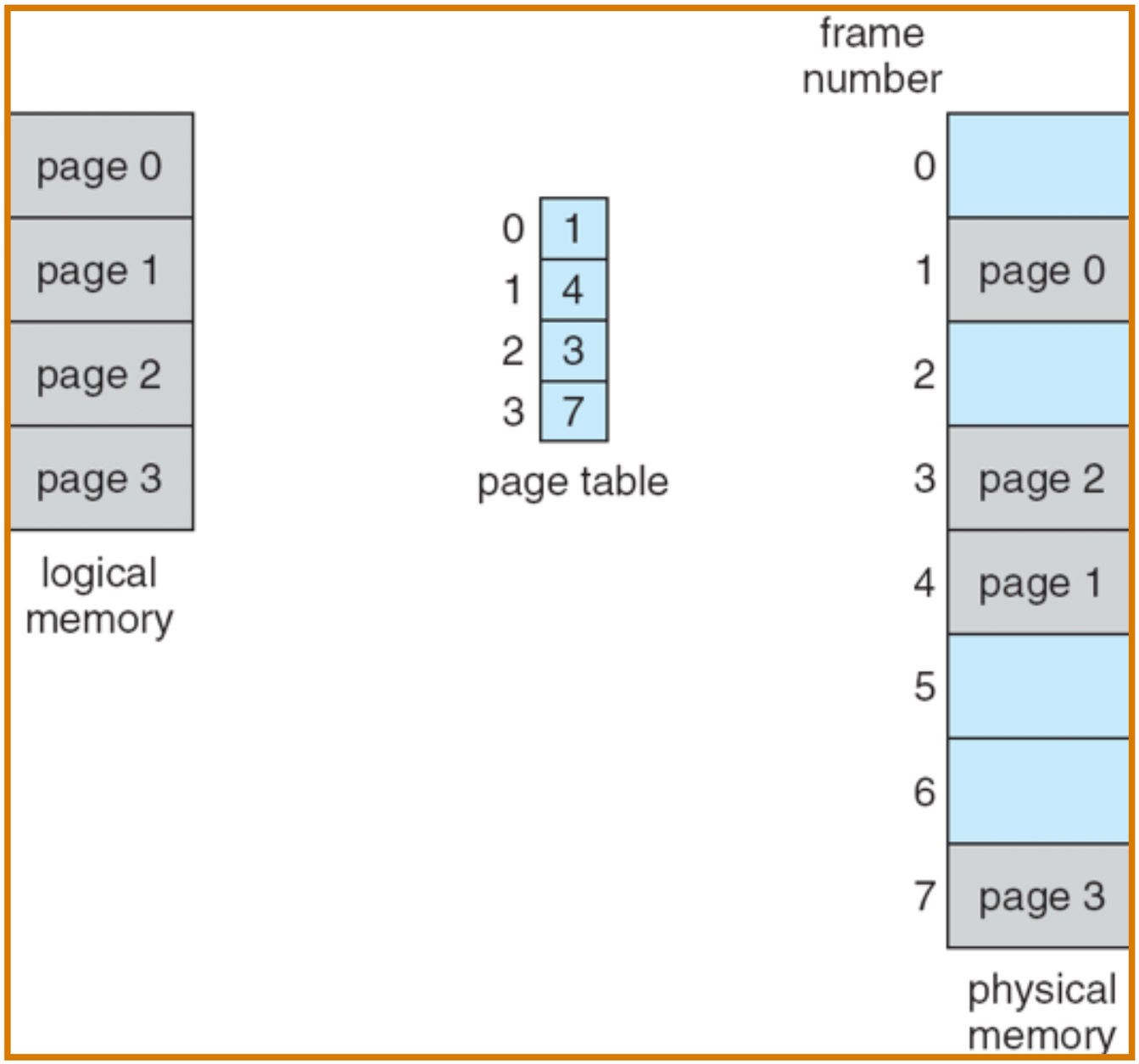
Page Tables

Virtual Address

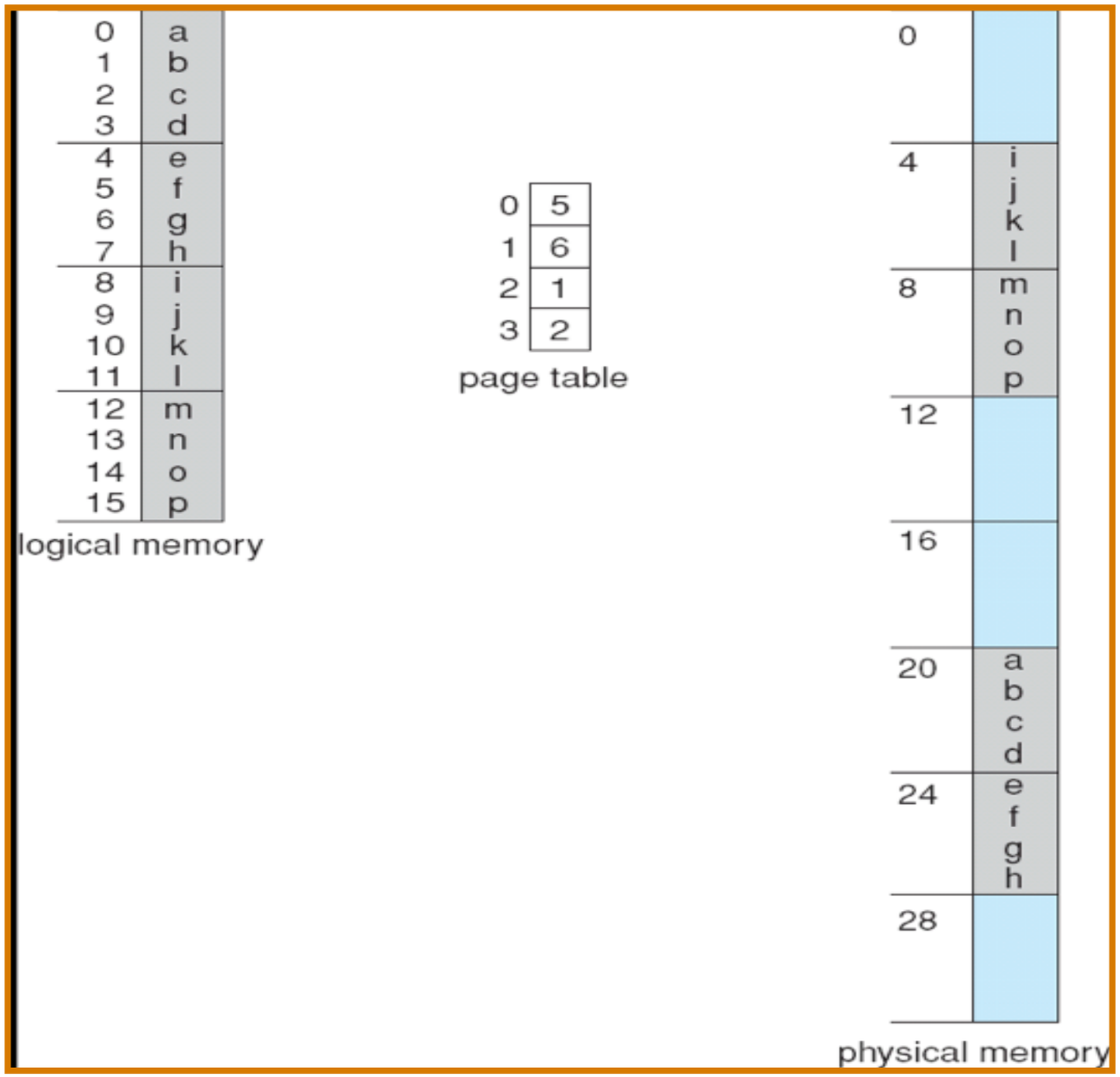


Physical Address

Logical to Physical Memory

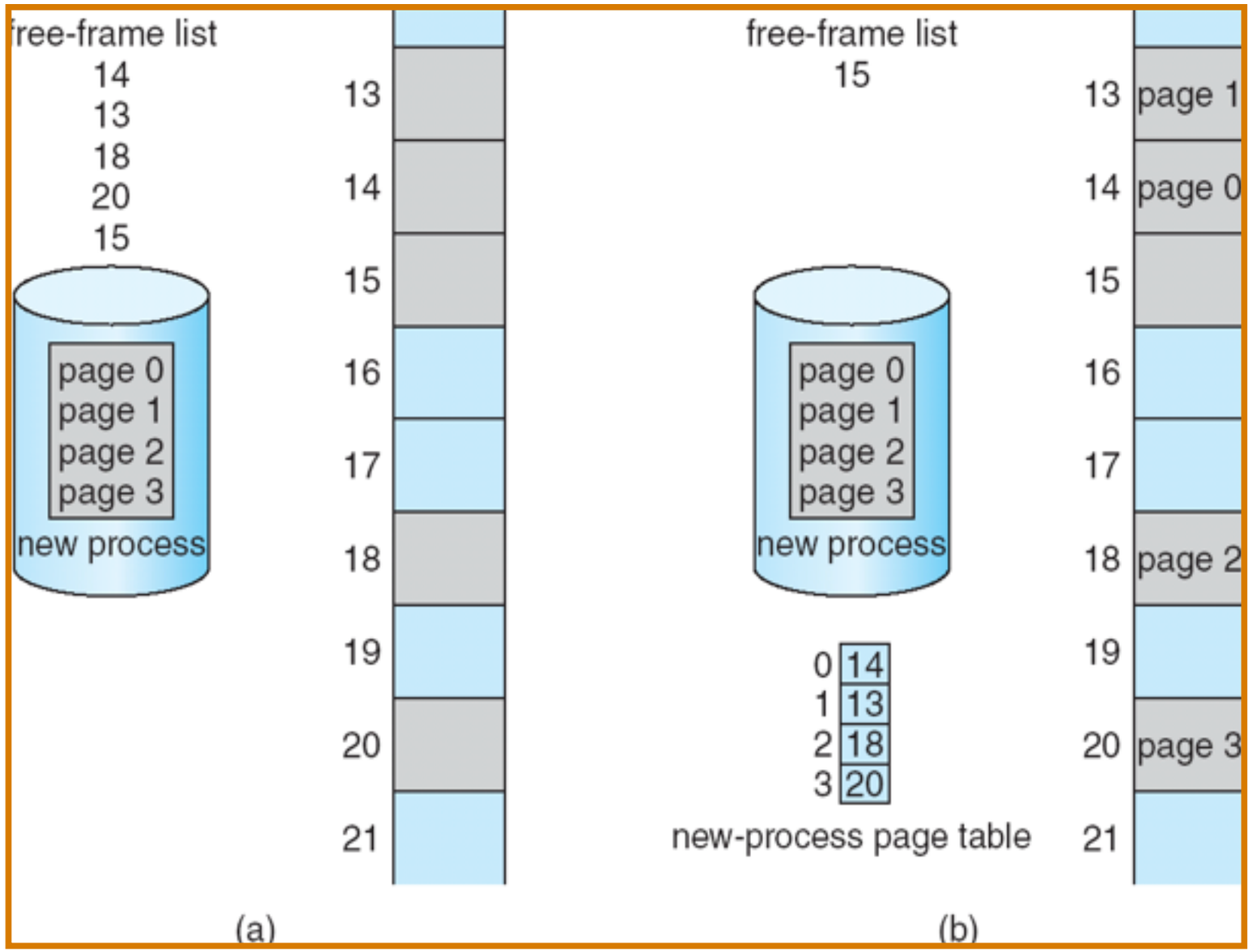


Paging Example



32-byte memory and 4-byte pages

Free Frames



Before allocation

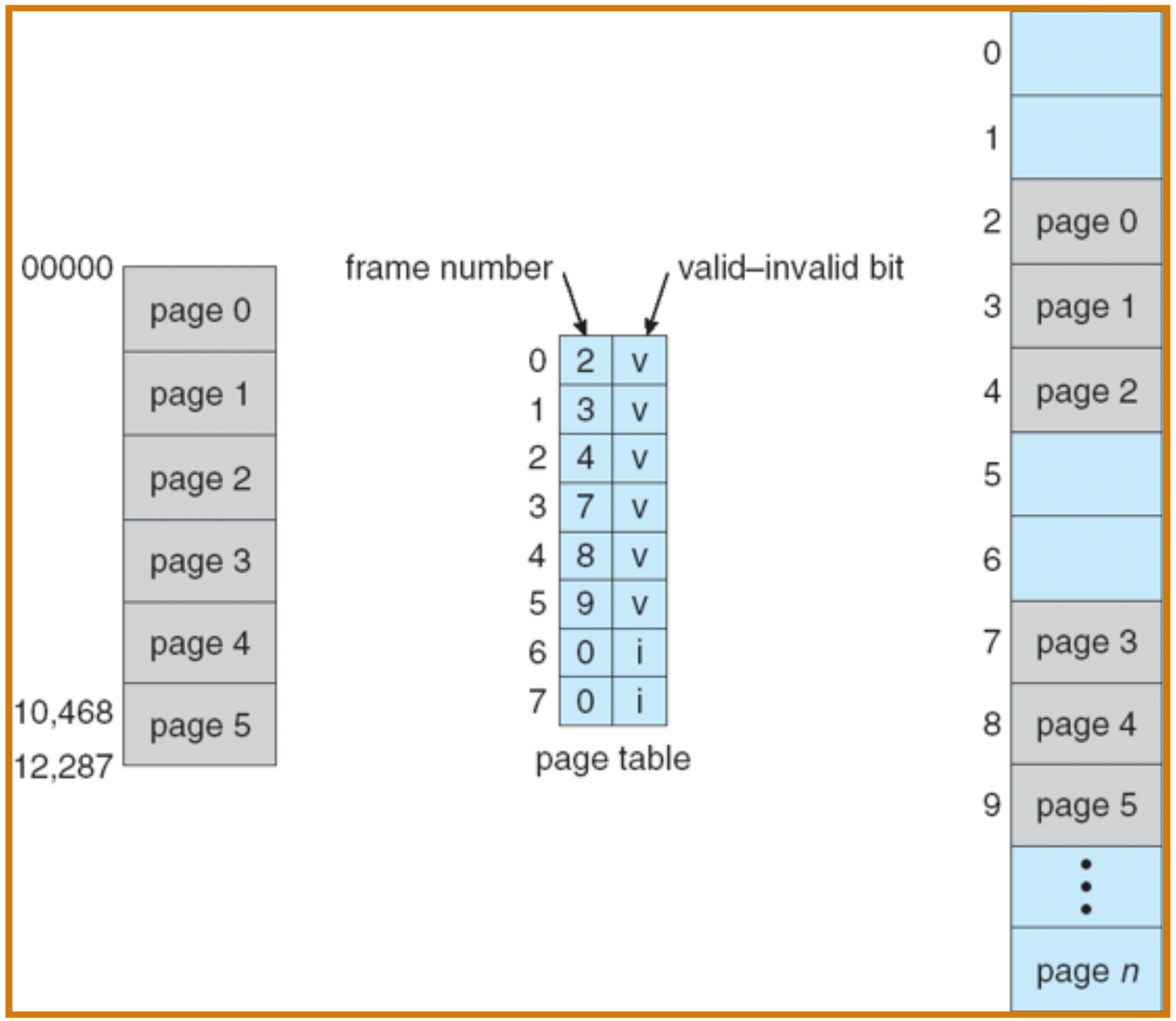
After allocation

Page Table Entry Format

UNIVERSITY OF OREGON

- Physical page Number
- Valid/Invalid bit.
- Protection bits (Read / Write / Execute)
- Modified bit (set on a write/store to a page)
  - Useful for page write-backs on a page-replacement.
- Referenced bit (set on each read/write to a page).
  - Will look at how this is used a little later.
- Disable caching.
  - Useful for I/O devices that are memory-mapped.

Valid (v)/Invalid (i) Bit in Page Table

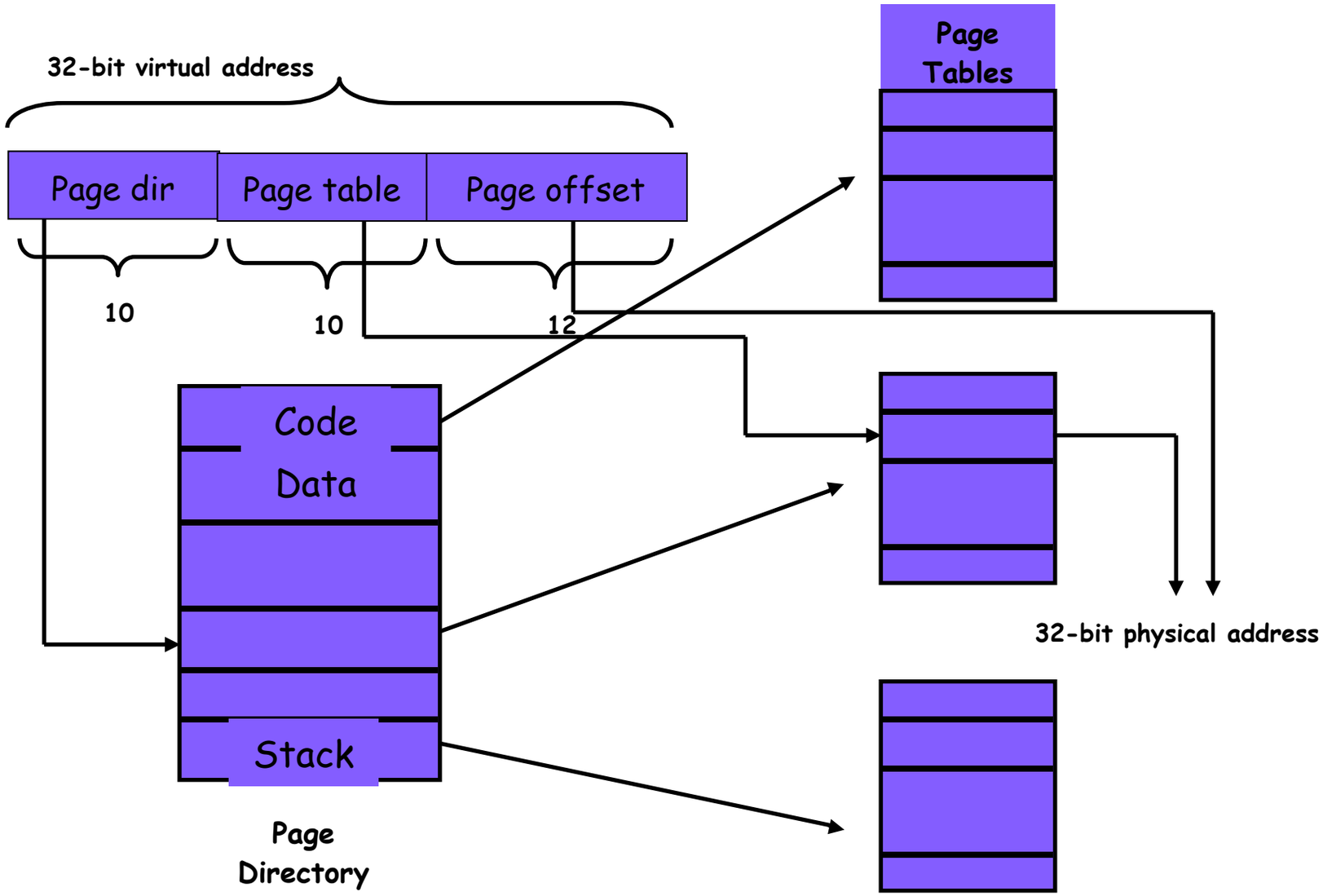


Issues to Address

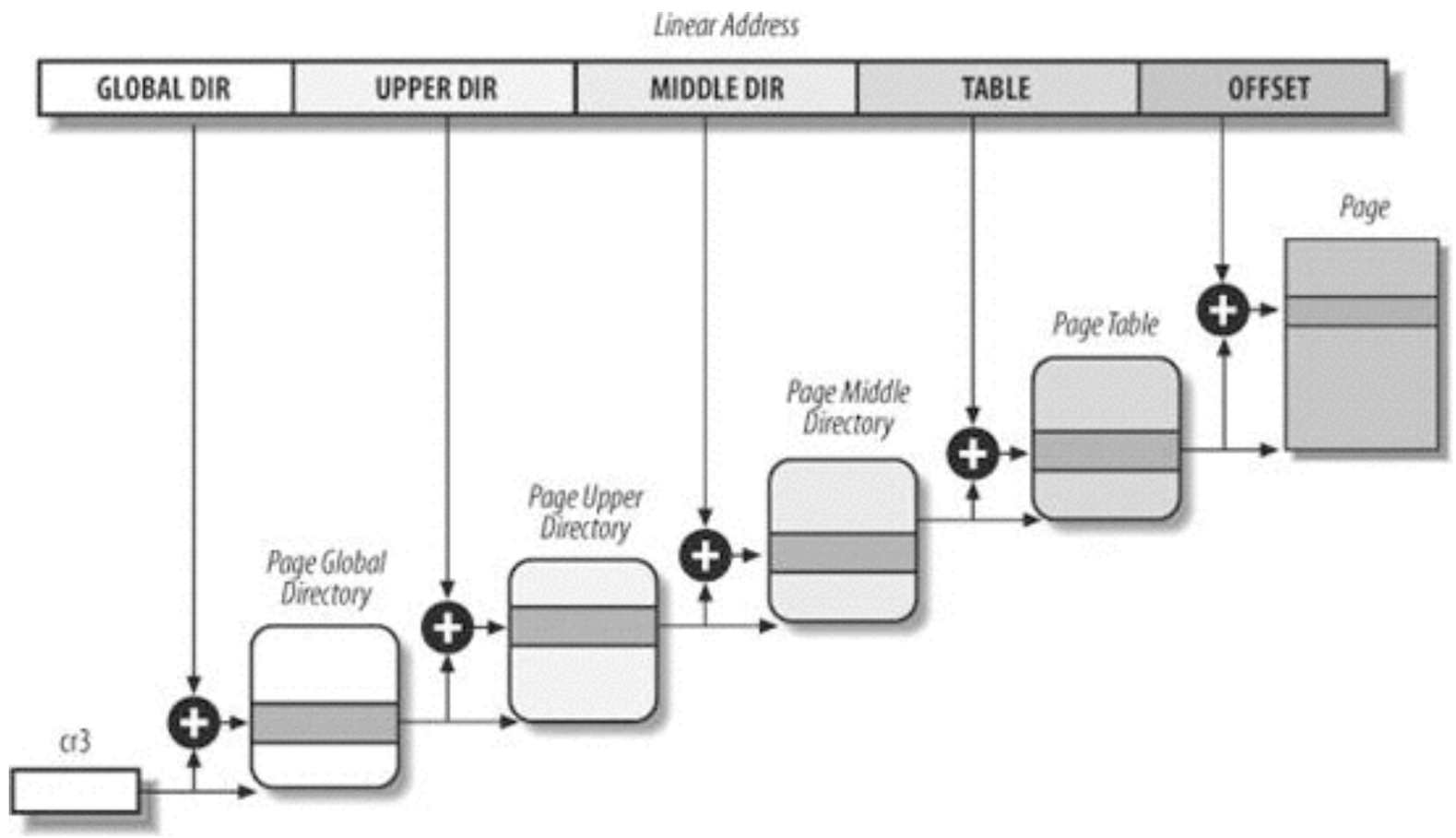
UNIVERSITY OF OREGON

- Size of page-tables would be very large!
- For example, 32-bit virtual address spaces (4 GB) and a 4 KB page size would have ~1 M pages/entries in page-tables.
- What about 64-bit virtual address spaces?!
- A process does not access all of its address space at once! Exploit this locality factor.
- Use multi-level page-tables. Equivalent to paging the page-tables.
- Inverted page-tables.

Example: 2-level Page Table



- Linux: 4-level page table (necessary for 64-bit arch)
- Virtual address space is sparse and widely scattered
  - stack at top, heap at bottom, dynamic libraries in the middle



MMU

UNIVERSITY OF OREGON

- Page-table lookup needs to be done on every memory-reference for both code & data!
  - Can be very expensive if this is done by software.
- Usually done by a hardware unit called the MMU (Memory-Management Unit).
  - Located between CPUs and caches.

Role of the MMU

UNIVERSITY OF OREGON

- Given a Virtual Address, index in the page-table to get the mapping.
- Check if the valid bit in the mapping is set, and if so put out the physical address on the bus and let hardware do the rest.
- If it is not set, you need to fetch the data from the disk (swap-space).
  - We do not wish to do this in hardware!

Requirements

UNIVERSITY OF OREGON

- Address translation/mapping must be very fast! Why?
  - Because it is done on every instruction fetch, memory reference instruction (loads/stores). Hence, it is in the critical path.
- Previous mechanisms access memory to lookup the page-tables. Hence it is very slow!
  - CPU-Memory gap is ever widening!
- Solution: Exploit the locality of accesses.

TLBs

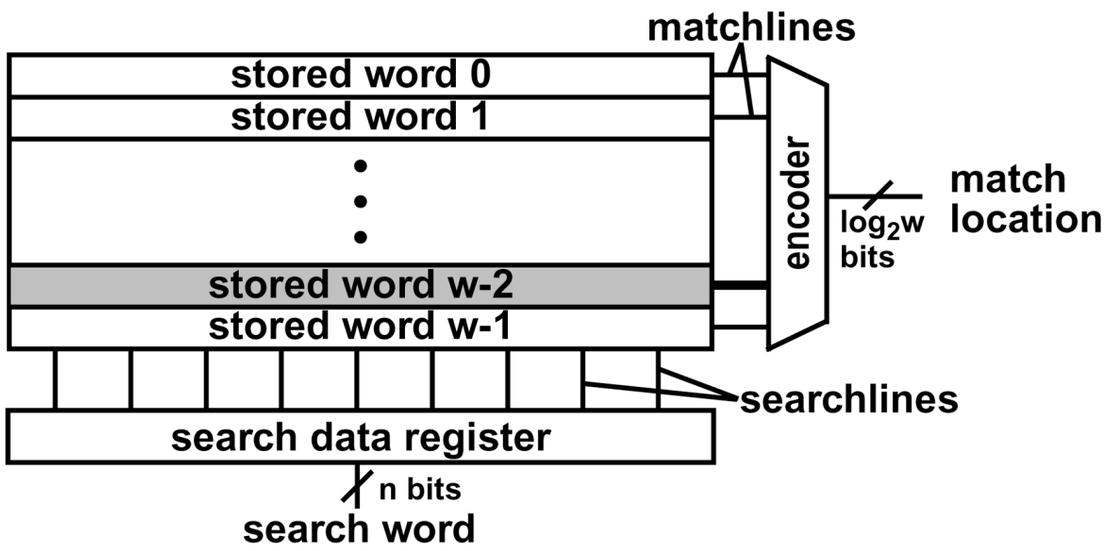
UNIVERSITY OF OREGON

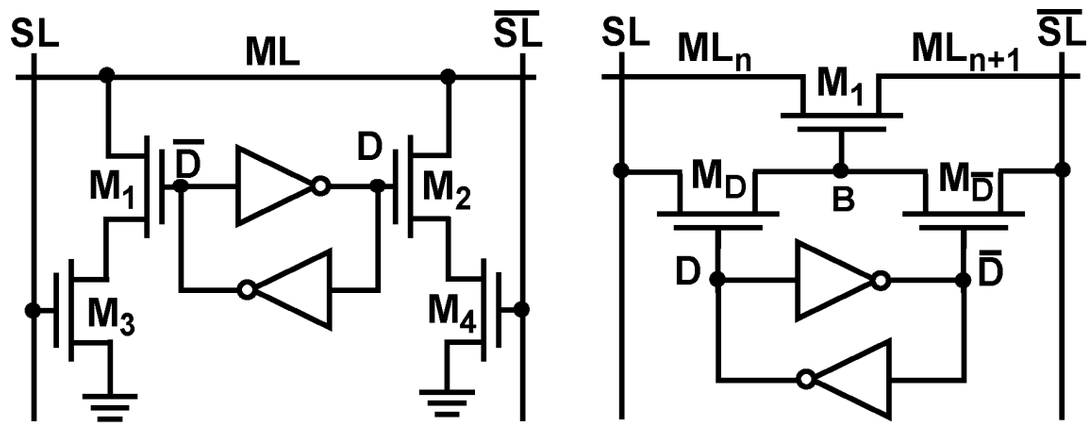
- *Translation Look-Aside Buffer*
- Typically programs access a small number of pages very frequently.
- Temporal and spatial locality are indicators of future program accesses.
- Temporal locality
  - Likelihood of same data being re-accessed in the near future.
- Spatial locality
  - Likelihood of neighboring locations being accessed in the near future.
- TLBs act like a cache for page-table.

CAM

UNIVERSITY OF OREGON

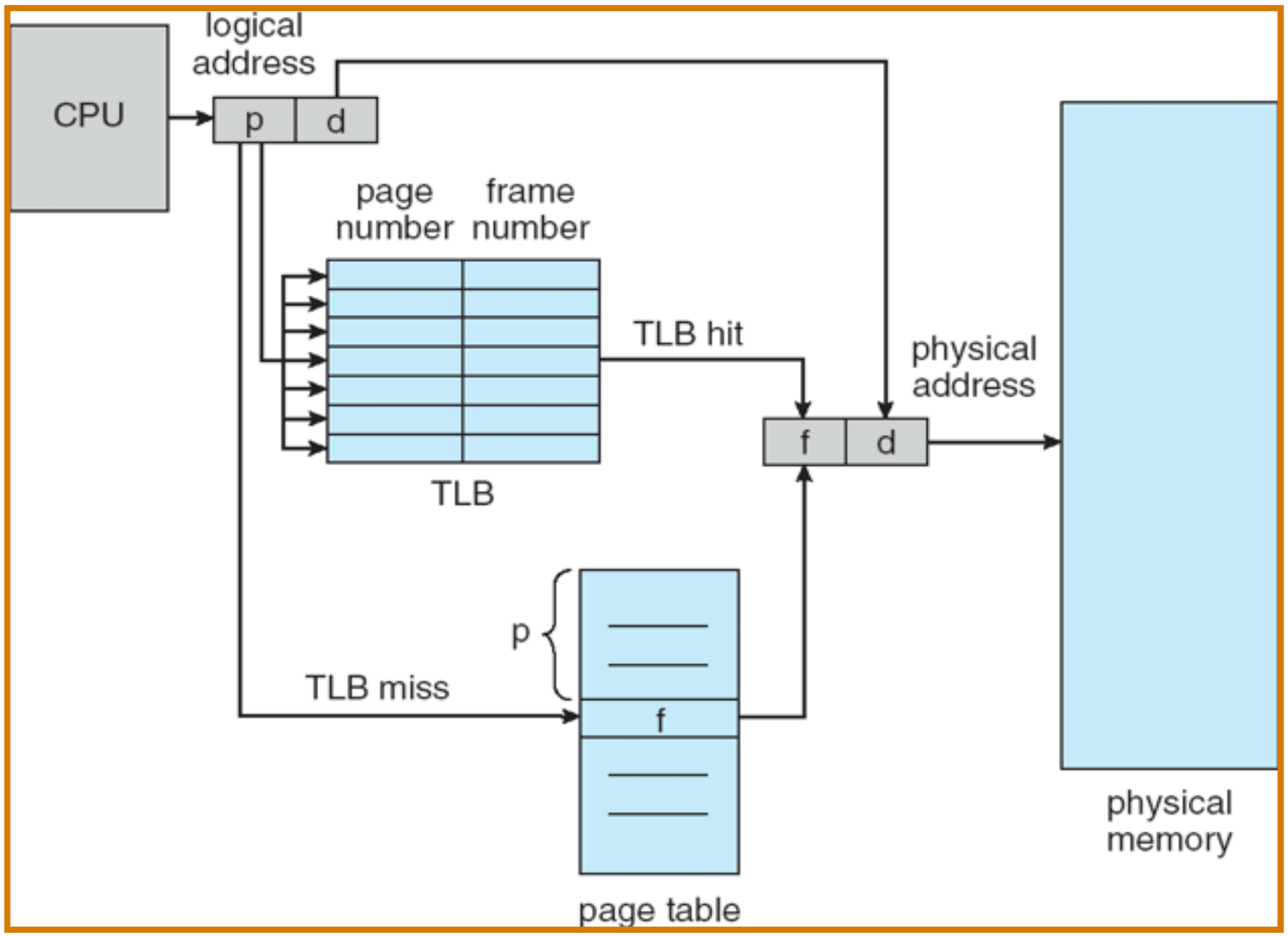
- Magic behind the TLB: associative memory
  - also called content-addressable memory (CAM)
- Implements lookup table functionality in a single clock cycle using dedicated comparison circuitry
  - How does this differ from regular memory?





Address Translation w. TLB

UNIVERSITY OF OREGON



TLB Cache

UNIVERSITY OF OREGON

- Typically, TLB is a cache for a few (8/16/32) Page-table entries.
- Given a virtual address, check this cache to see if the mapping is present, and if so we return the physical address.
- If not present, the MMU attempts the usual address translation.
- TLB is often designed as a fully-associative cache.
- TLB entry has
  - Used/unused bits, virtual page number, Modified bit, Protection bits, physical page number.

Address Translation Steps

UNIVERSITY OF OREGON

- Virtual address is passed from the CPU to the MMU (on instruction fetch or load/store instruction).
- Parallel search of the TLB in hardware to determine if mapping is available.
- If present, return the physical address.
- Else MMU detects miss, and looks up the page table as usual (*page walk:* this is not a page fault)
- If page table lookup succeeds, return physical address and insert mapping into TLB evicting another entry.
- Else it is a *page fault*.

Hit Ratio

UNIVERSITY OF OREGON

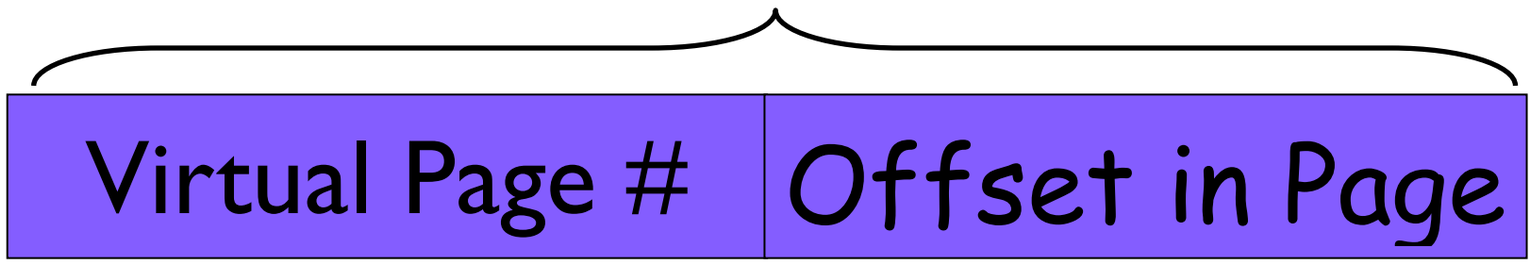
- Fraction of references that can be satisfied by TLB is called *hit ratio(h)*.
- For example, if it takes 100 ns to access page-table entry and 20 ns to access TLB,
  - average lookup time = 20 * *h* + 100 * (1 – *h*).

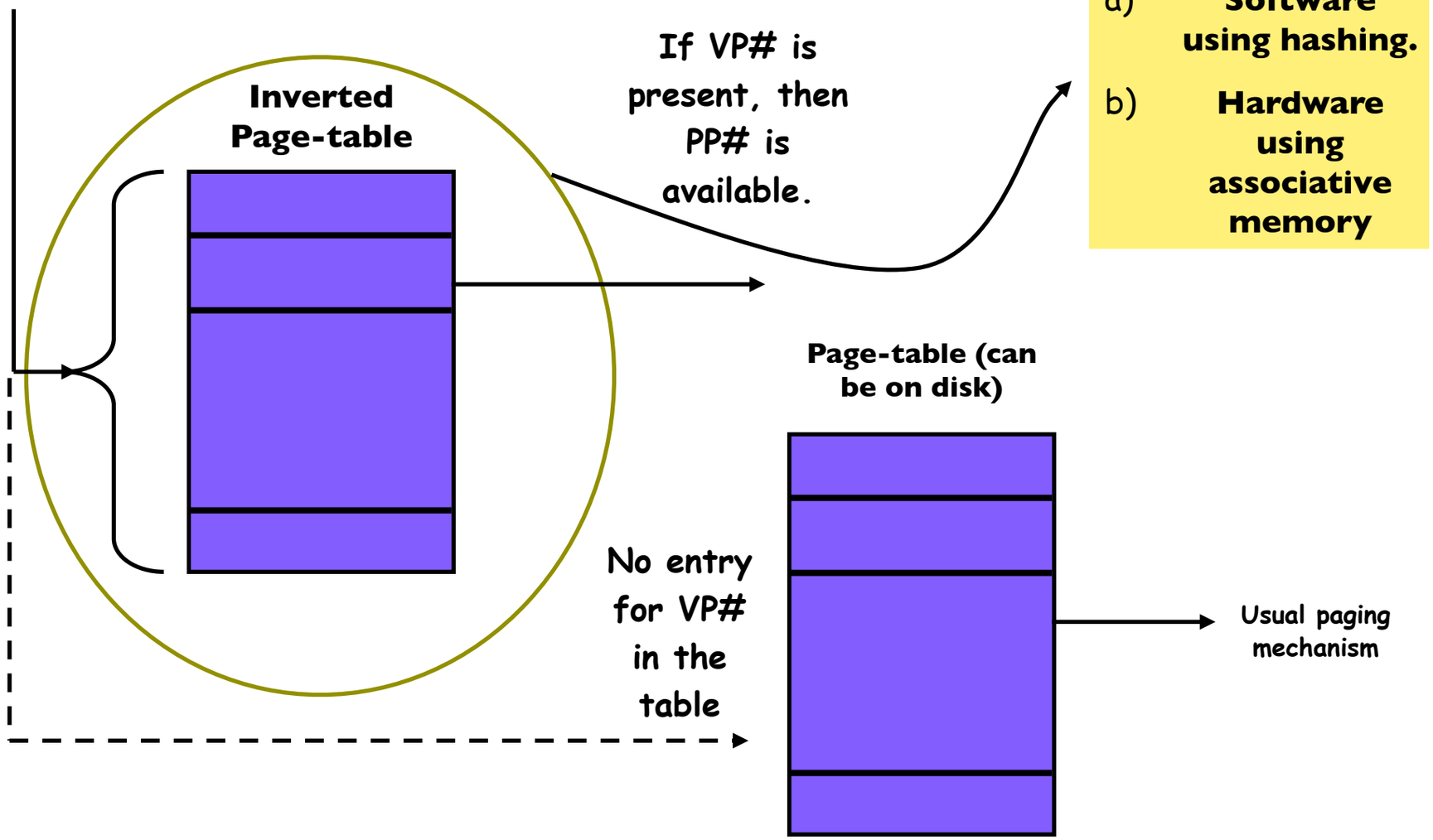
Inverted Page-tables

UNIVERSITY OF OREGON

- Page-tables could become quite large!
- Above mechanisms pages the page-tables and uses TLBs to take advantage of locality.
- Inverted page-tables organize the translation mechanism around physical memory.
- Each entry associates a physical page with the virtual page stored there!
  - Size of Inverted Page-table = Physical Memory size / Page size.

Virtual Address





IVT implemented in

a) **Software using hashing.**

b) **Hardware using associative memory**

Segmentation

UNIVERSITY OF OREGON

- Can be used as a programming convenience
- Several times you have different segments (code, data, stack, heap), or even within data/heap you may want to define different regions.
- You can then address these segments/regions using base + offset.
- You can also define different protection permissions for each segment.
- However, segmentation by itself has all those original problems (contiguous allocation, fitting in memory, etc.)

Segmentation with Paging

UNIVERSITY OF OREGON

- Define segments in the virtual address space.
- In programs, you refer to an address using [Segment Ptr + Offset in Segment].
  - E.g Intel family
- Segment Ptr leads you to a page table, which you then index using the offset in segment.
- This gives you physical frame #. You then use page offset to index this page.
- Virtual address = (Segment #, Page #, Page Offset)

Summary

UNIVERSITY OF OREGON

- Memory Management
  - Limited physical memory resource
- Keep key process pages in memory
  - Swapping (and paging, later)
- Memory allocation
  - High performance
  - Minimize fragmentation

Summary

UNIVERSITY OF OREGON

- Paging
  - Non-contiguous allocation
  - Pages and frames
  - Fragmentation
  - Page tables
  - Hardware support
  - Plus, Segmentation

- Next time: Virtual Memory