Memory Management

Goals:

- Provide a convenient programming model
- Efficiently allocate a scarce resource
- Protect programs from each other
- Protect the OS from programs

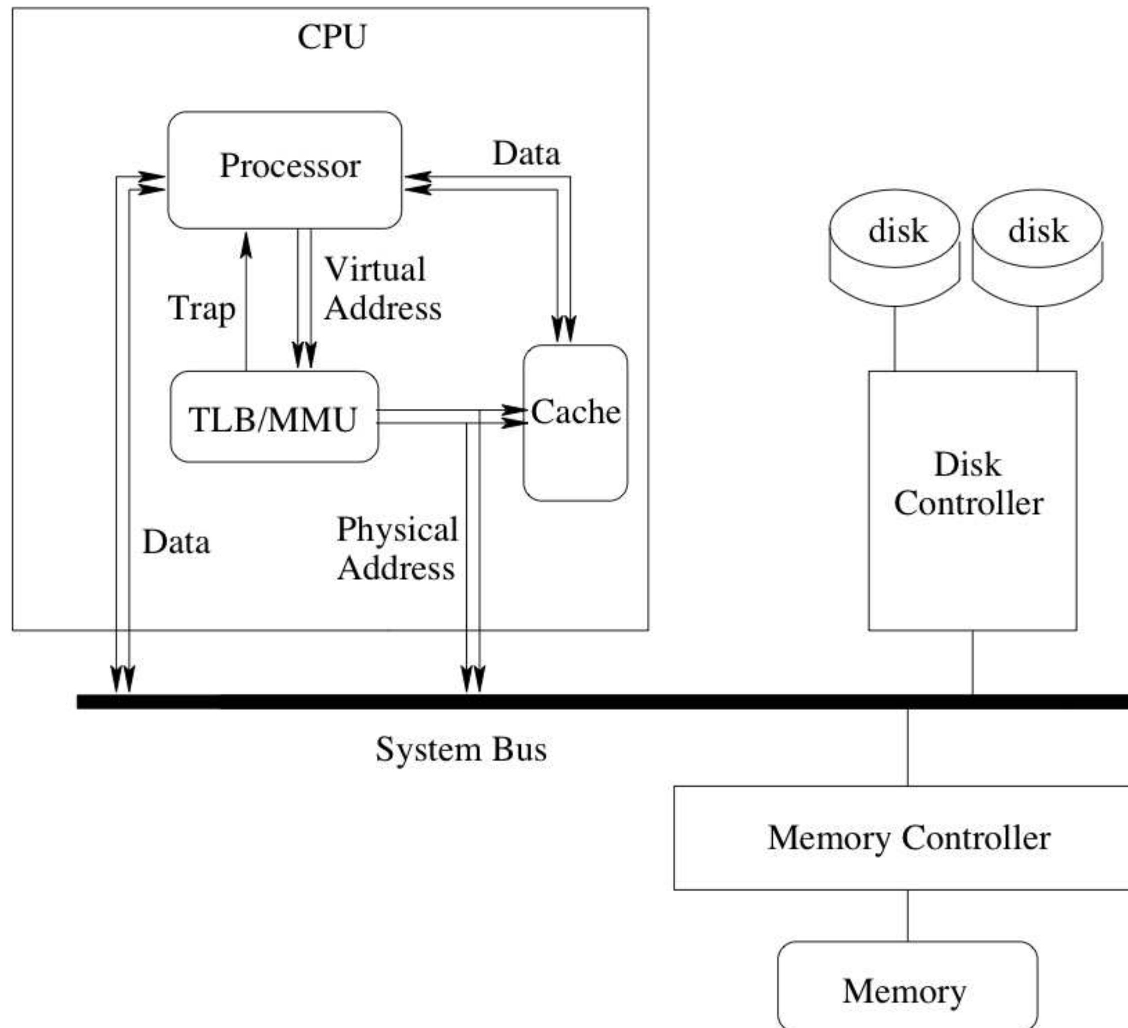
Mechanisms:

- Physical and virtual addressing
- Paging and segmentation
- Page table management

Policies:

- Page replacement algorithms

Memory and Architecture



- ***TLB***: Translation Look aside Buffer
- ***MMU***: Memory Management Unit

Physical Addresses

A ***physical address*** is one that can be directly used by the memory subsystem

With 512 MB of physical memory, the OS sees physical memory as an array of 512 million bytes:

- **`mem[0]`** is the first byte
- **`mem[256*1024*1024]`** is the middle byte
- **`mem[512*1024*1024]`** is the first illegal address

Virtual Addresses

Processes see only ***virtual addresses***

OS provides each process with a ***virtual address space*** that is 4 GB (on a 32-bit machine):

- **`vmem[10][0]`** is the first byte of memory for process 10
- **`vmem[10][4*1024*1024*1024 - 1]`** is the last byte of memory for process 10

Mapping Virtual to Physical Addresses

OS must provide a mapping from virtual addresses to physical addresses:

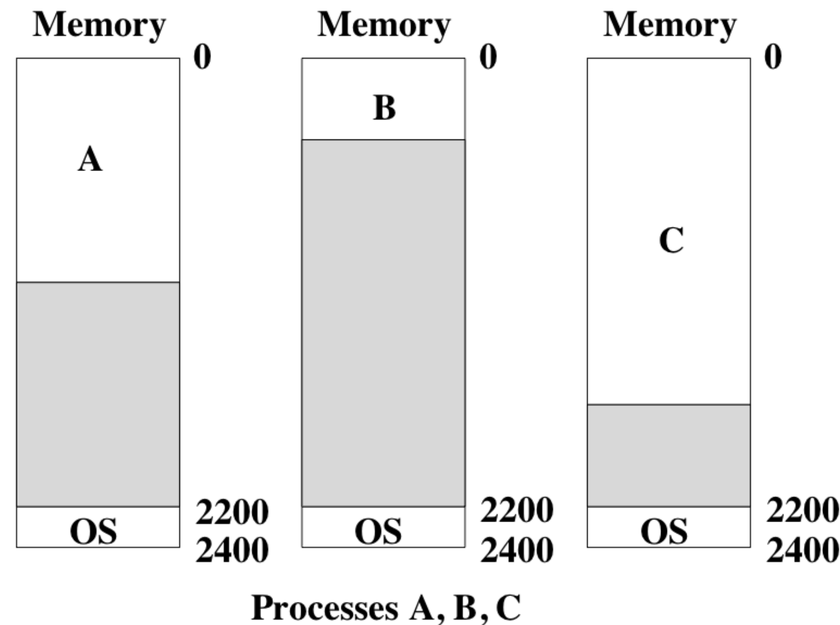
- The same virtual address in two different processes does not map to the same physical address
- Some virtual addresses do not map to any physical address

The arrays **`mem`** and **`vmem`** don't really exist, but they're a good way to model the problem

Where do Addresses Come From?

- **Compile time**: The compiler generates the exact physical location in memory starting from some fixed starting position. The OS does nothing.
- **Load time:** Compiler generates an address, but at load time the OS determines the process' starting position. Once the process loads, it does not move in memory.
- **Execution time:** Compiler generates an address, and OS can move it around in memory as needed

Uniprogramming (e.g. DOS)



- OS gets a fixed part of memory
- Process is always loaded starting at address 0
- Process executes in a contiguous section of memory
- Compiler can generate physical addresses
- Maximum address = *Memory Size* - *OS Size*
- OS is protected from process by checking addresses used by process

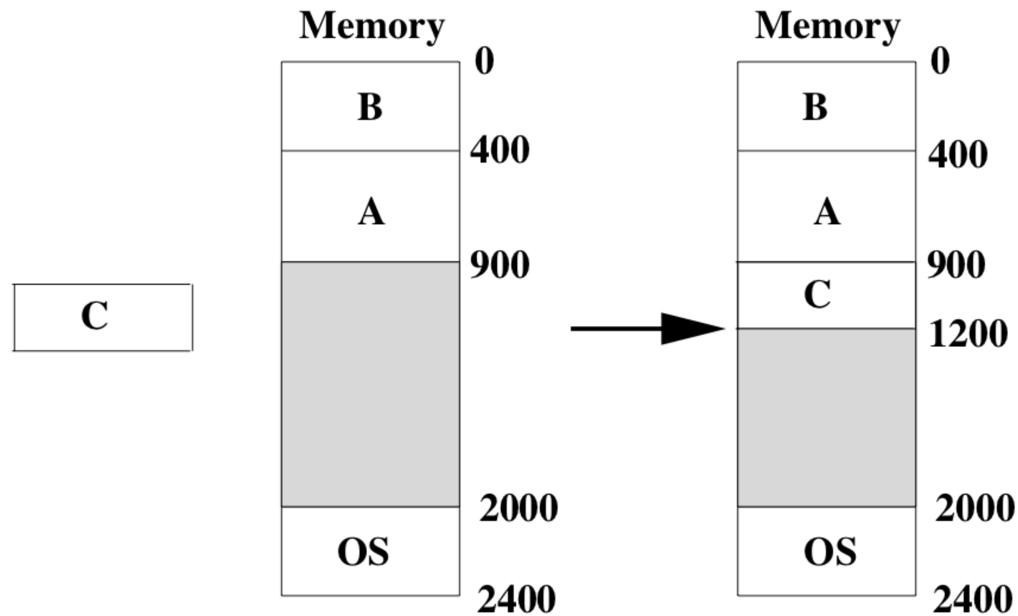
Multiprogramming

- **Transparency**:
  - Multiple processes can coexist in memory
  - No process should be aware that memory is shared
  - Processes should not care what physical portion of memory they get

- **Safety**:
  - Processes must not be able to corrupt each other
  - Processes must not be able to corrupt the OS

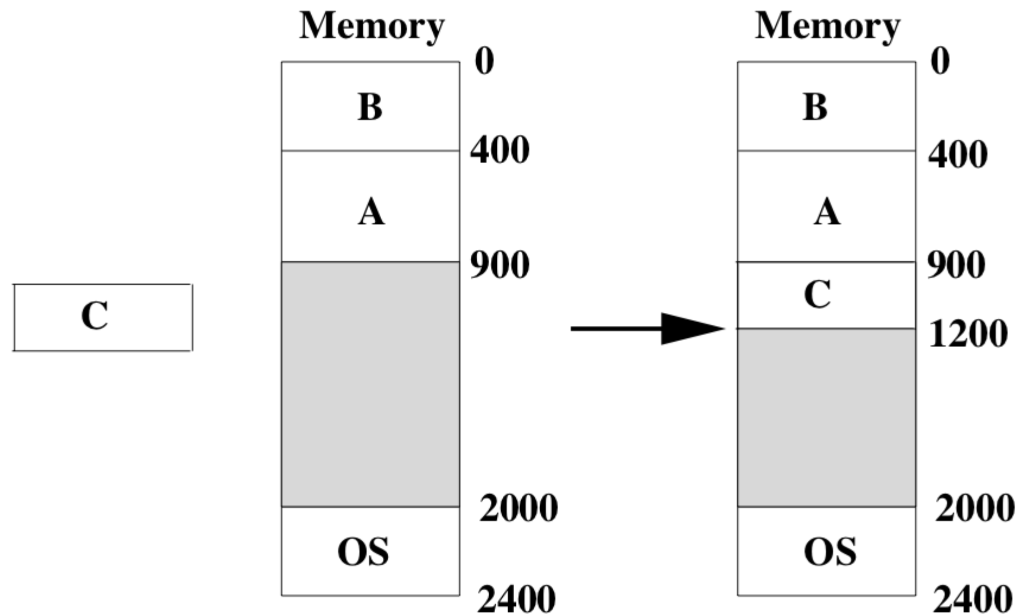
- **Efficiency**:
  - Performance should not degrade badly due to sharing

Relocation



- Put the OS in the highest memory
- Compiler/linker assumes that the process starts at 0
- When the OS loads the process, it allocates a contiguous segment of memory where process fits (if available)

Relocation



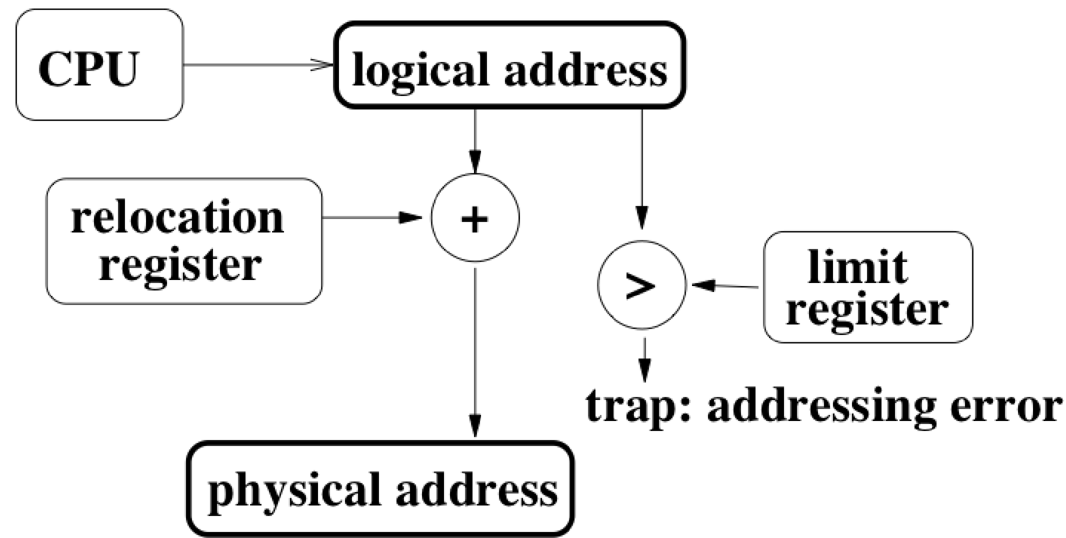
- The first physical address of the process is the ***base address*** and the largest physical address the process can access is the ***limit address***
- The base address is also known as the ***relocation address***, and may be kept in a special register

Static Relocation

Static resolution means

- At load time, the OS adjusts the addresses in a process to reflect its position in memory
- Once a process is assigned a place in memory and starts executing it, the OS cannot move it.

Dynamic Relocation



Dynamic resolution means

- Hardware adds relocation register (base) to virtual address to get a physical address
- Hardware compares address with limit register; address must be less than base
- If test fails, the processor takes an address trap and ignores the physical address

Dynamic Relocation

Advantages:

- OS can easily move a process during execution
- OS can allow a process to grow over time
- Simple, fast hardware: two special registers, an add, and a compare

Disadvantages:

- Slows down hardware due to the add on every memory reference

In the 60's, this was a show stopper!

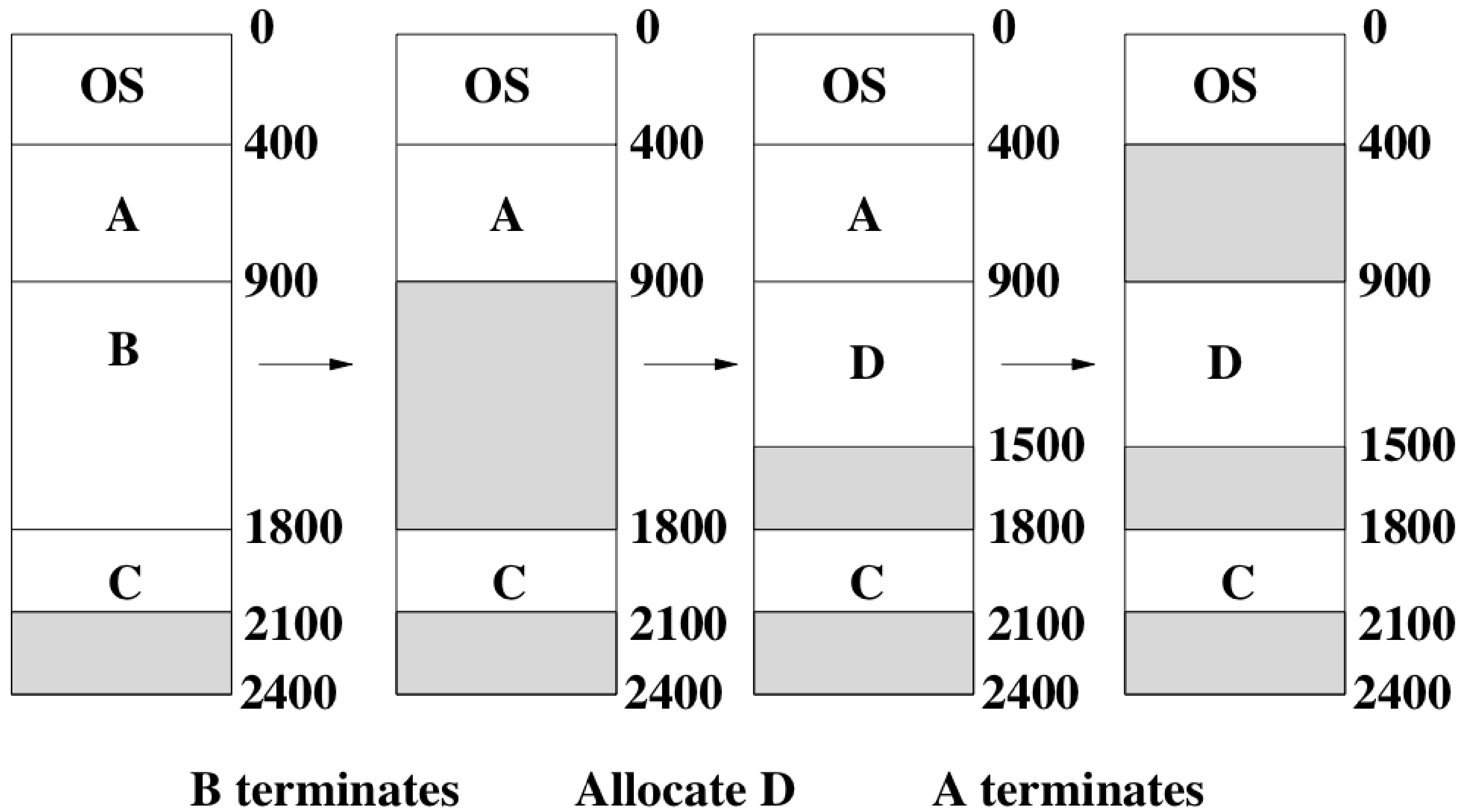
- Can’t share memory (such as program text) between processes
- Process is still limited to physical memory size
- Multiprogramming limited, since all memory of all active processes must fit in memory
- Complicates memory management

Dynamic Relocation

Check our goals:

- **Transparency**: processes are largely unaware of sharing
- **Safety**: each memory reference is checked
- **Efficiency**: address translation and checks are done in hardware, so they are fast, but if a process grows, moving it is very slow

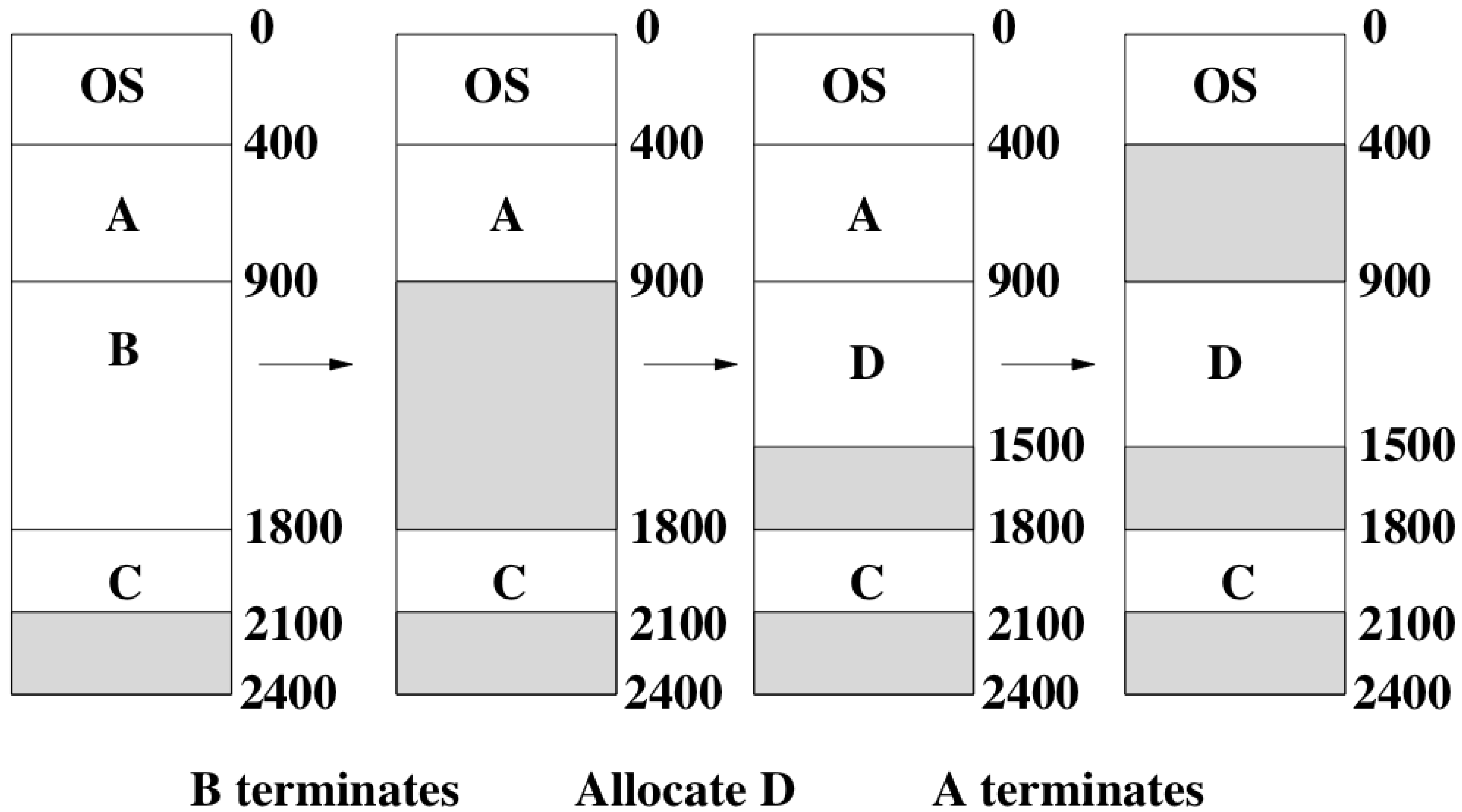
Memory Allocation



Holes: pieces of free memory (shaded above)

Given a memory request from a starting process, OS must decide which hole to use

Memory Allocation



External fragmentation = holes between processes

In simulations with some allocators, 1/3 of memory may be lost to fragmentation

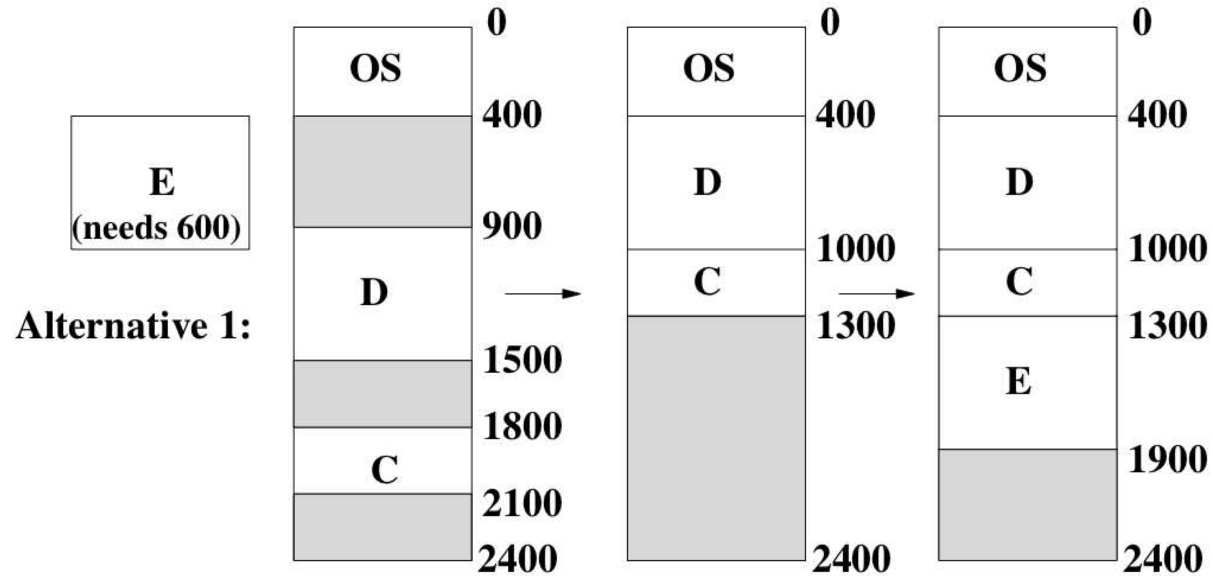
Memory Allocation Policies

- ***First-Fit***: allocate the first one in the list in which the process fits (but the search can start in different places)
- ***Best-Fit***: Allocate the smallest hole that is big enough to hold the process (meanwhile, coalesce adjacent holes?)
- ***Worst-Fit***: Allocate the largest hole to the process

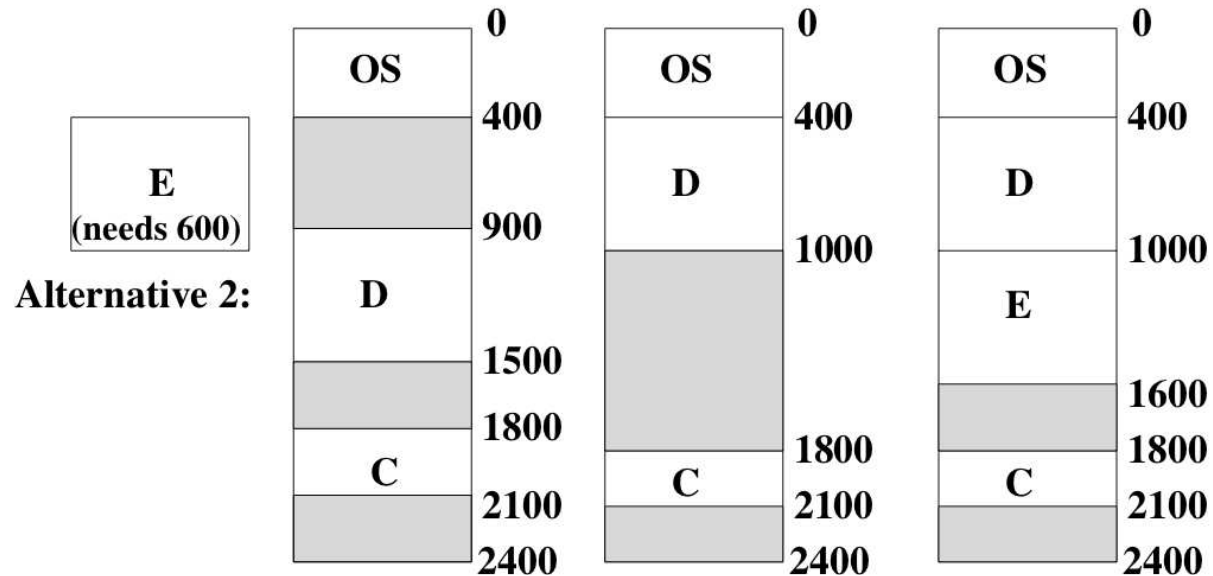
Simulations show

- First-fit and best-fit usually yield better storage utilization than worst-fit
- First-fit is generally faster than best-fit

Compaction



Compaction →



Swapping

- Move an inactive process to disk, releasing its memory
- When process becomes active, reload it in memory
  - Static relocation: process must be put in the same position
  - Dynamic relocation: OS finds a new position in memory

If swapping is part of the system, compaction is easy to add

How could/should swapping interact with CPU scheduling?

Where do Addresses Come From?

Compile time:

- Advantages: Simple, lookup is fast
- Disadvantages: OS cannot move process, so only one process may execute at a time

Load time:

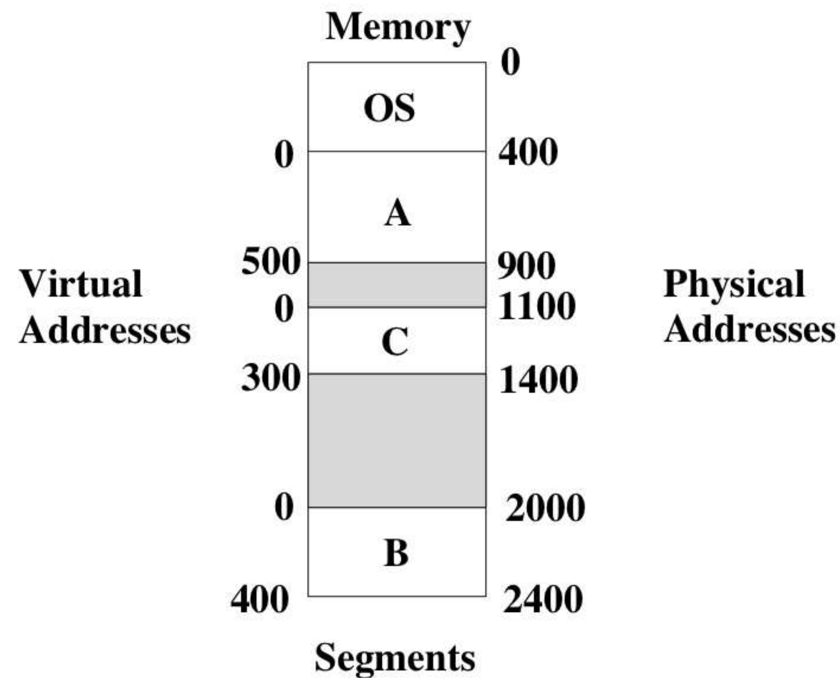
- Advantages: Sharing between processes, virtual memory, multiprogramming
- Disadvantages: Slightly slower, must run a process to completion without moving

Execution time:

- Advantages: most flexible, sharing, virtual memory, can move processes
- Disadvantages: slowest option

Contiguous Memory

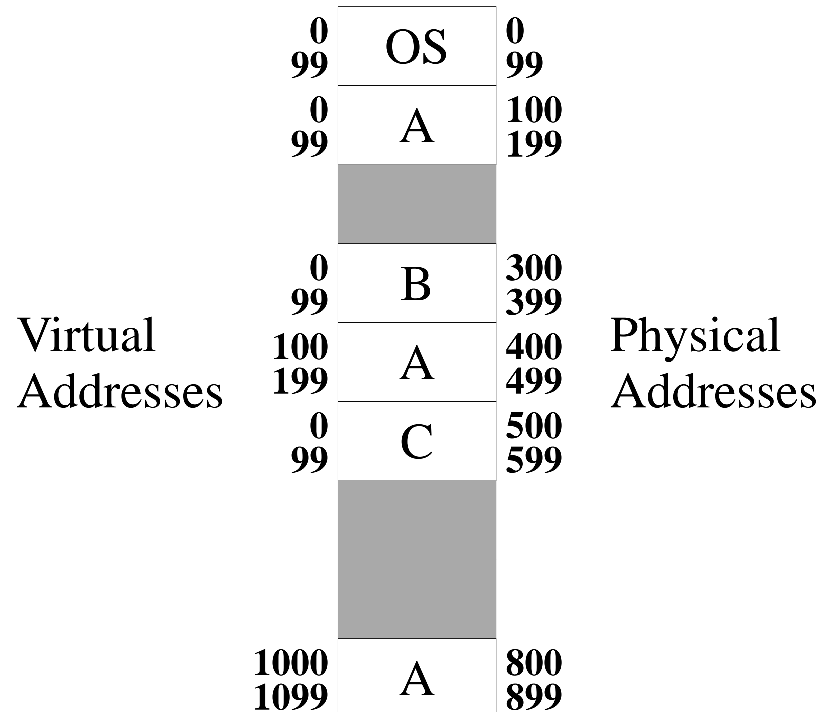
So far, we've just considered a mapping from contiguous virtual addresses to contiguous physical addresses:



Problems: External fragmentation, copying for compaction or resizing, difficulty sharing memory between processes, slow program loading, programs limited to physical memory size

Virtual Memory

Virtual memory means breaking the virtual address space into chunks of (potentially) discontiguous physical addresses



Note: *virtual address* is a general concept, while *virtual memory* is a specific technique for mapping virtual addresses

Pages

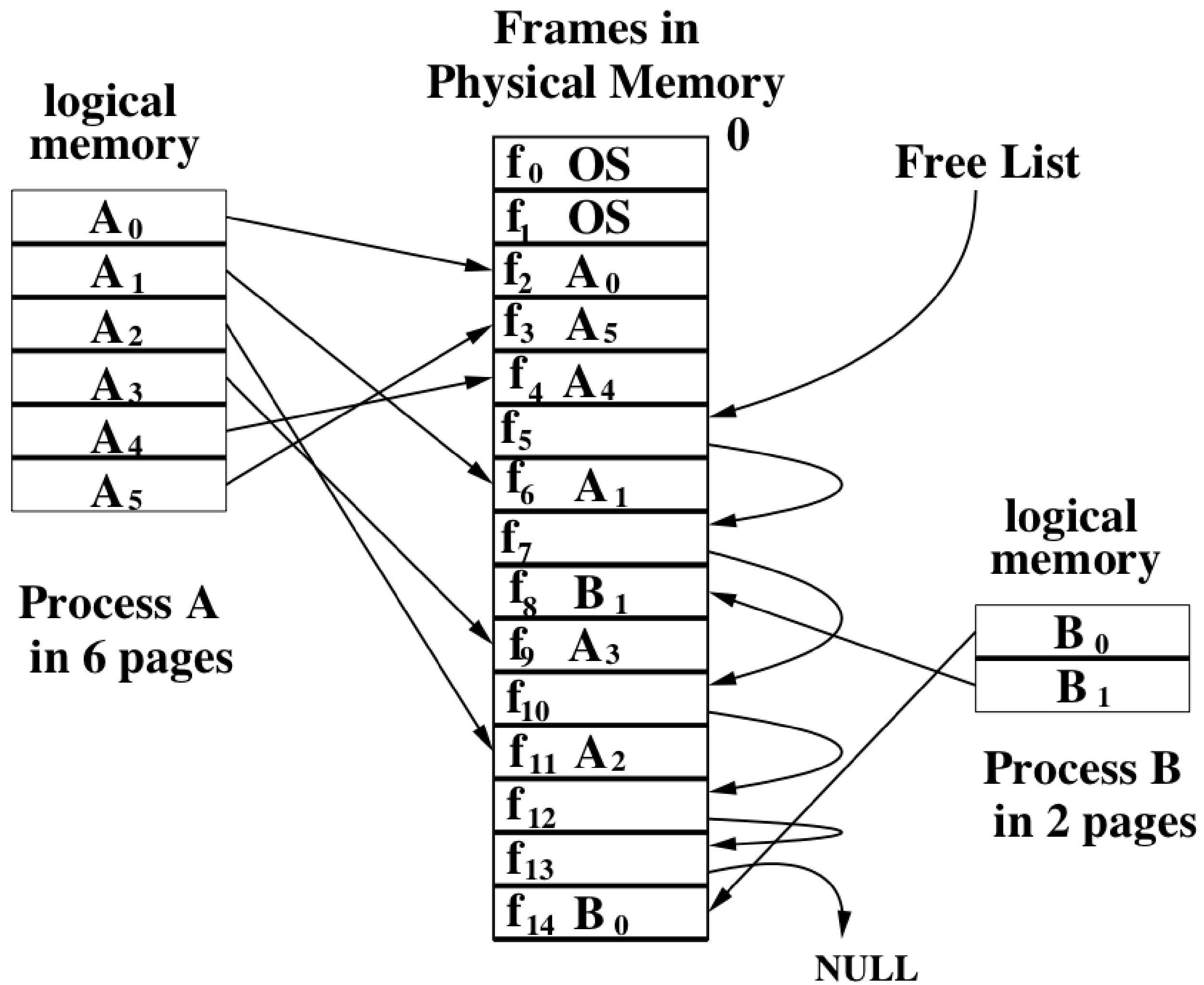
A ***page*** is a contiguous block of virtual addresses

A ***frame*** is a contiguous block of physical addresses

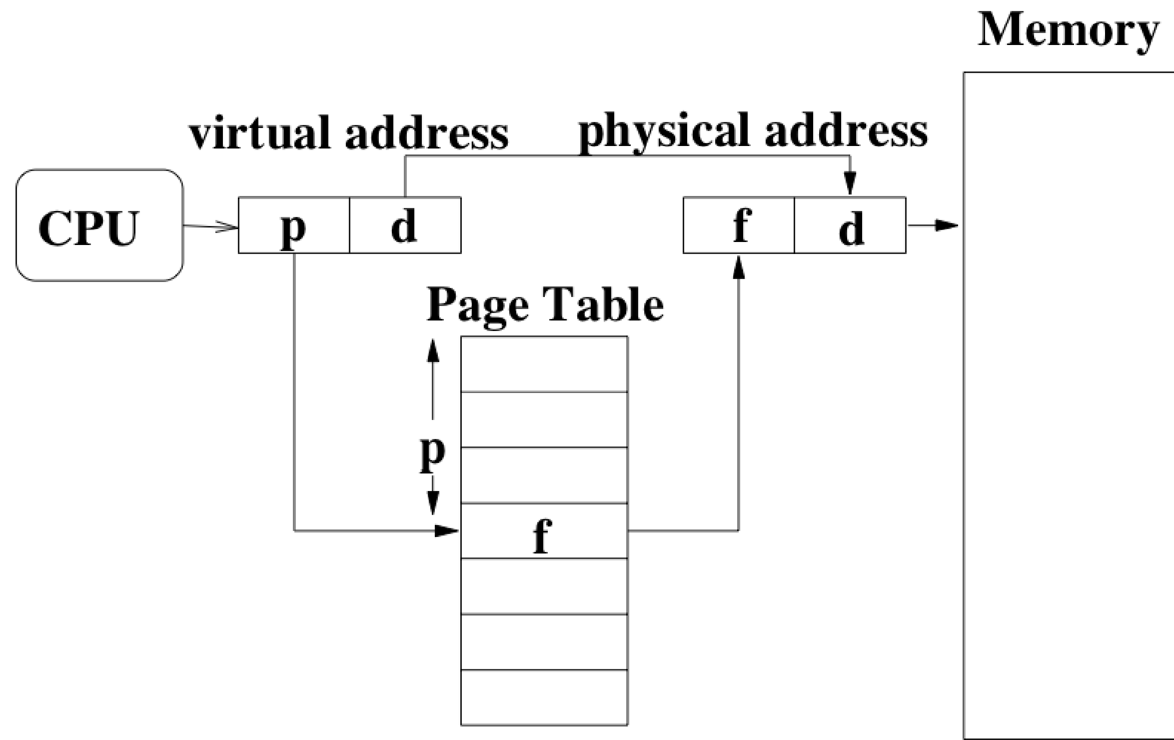
A page is mapped to a frame

- Page/frame size is fixed by OS (typically 4KB or 8KB)
- Page always start on a multiple of the page size
- Pages greatly simplify the hole-fitting problem
- Even if logical memory of the process is contiguous, pages need not be allocated to contiguous frames

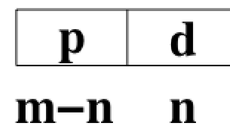
Pages



Mapping Pages to Frames



A ***page table*** tracks the page→frame mapping



p: page number
d: page offset

Mapping Pages to Frames

virtual memory

A_0
A_1
A_2
A_3

page table

page	frame
0	2
1	6
2	11
3	9

Frames in Memory

f_0 — 0
f_1 — 32 bytes
A_0 f_2
f_3 — 64 bytes
f_4
f_5 — 96 bytes
A_1 f_6
f_7 — 128 bytes
f_8
A_3 f_9 — 160 bytes
f_{10}
A_2 f_{11} — 192 bytes
f_{12}
f_{13} — 224 bytes
f_{14}
f_{15} — 256 bytes

virtual / physical memory size = 256 bytes

page size = 16 bytes

□ = 4 bytes

Mapping Pages to Frames

Paging is a form of dynamic relocation, where each virtual address is bound by the paging hardware to a physical address

- Think of the page table as a set of relocation registers, one for each frame

Mapping is invisible to the process; the OS maintains the mapping, and the hardware does the translation

Protection is provided with the same mechanisms as used in dynamic relocation

Making Page Translation Fast

Where do we put the page table?

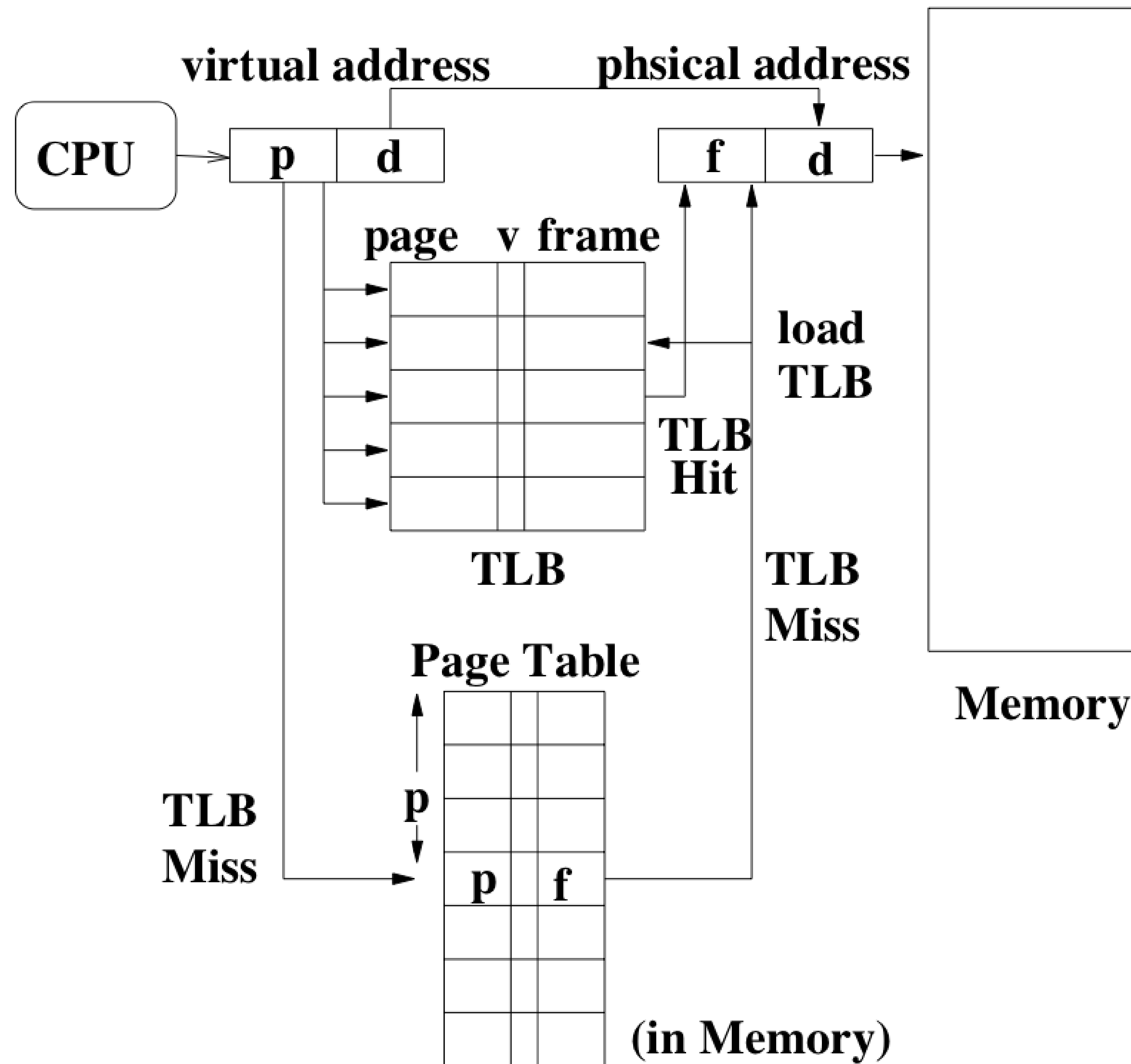
- **In registers**: fast, but limited page-table size
- **In memory**: larger page-table possible, but slow

The usual solution: a cache

Translation Look aside Buffer (TLB) caches page to frame translation

- 4KB entries is a typical size

TLB



v: “valid” bit, indicates whether the entry is up-to-date

Starting a Process with Virtual Memory

- Process needing *k* pages arrives
- Assuming that *k* page frames are free, OS allocates all *k* pages to the free frames
- OS puts the first page in a frame, and puts the frame number in the first entry in the page table; it then puts the second page in the frame and its frame number in the second entry of the page table; etc.
- OS marks all TLB entries as invalid (i.e., it ***flushes*** the TLB)
- OS starts process
- As process executes, OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full

Switching Processes

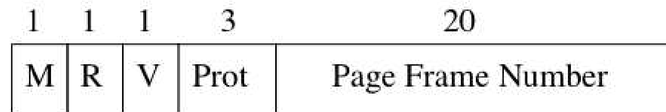
Process Control Block (PCB) must be extended to contain

- the page table
- a copy of the TLB, maybe

Steps:

- Copy the page table base register value to the PCB
- Copy the TLB to the PCB (optionally)
- Flush the TLB
- Restore the page table base register
- Restore the TLB, if it was saved

Page Table Entries



Page table entries (PTEs) contain more information than just address translations:

- Page frame number
- Valid bit
- Reference bit
- Modify bit
- Protecton — read, write, execute

Real PTEs (e.g. x86) contain a lot more

Multilevel Paging

Page tables can be large...

Example: x86

- 4KB page size = 2^{12}
- 32-bit PTE
- $\Rightarrow 4 * 2^{20} = 4$ MB

Per process!

Solution: 2-level page tables

- Break page number into two parts
- Don't allocate tables for unmapped first parts

Some Sparc and Alpha chips use three-level page tables

Sharing

Paging introduces the possibility of sharing since the memory used by a process no longer needs to be contiguous

- Shared code must be reentrant: the processes that are using it cannot change it (e.g., no data in reentrant code)
- Sharing of pages is similar to the way threads share text and memory with each other
- A shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address
- The user program (e.g., emacs) marks text segment of a program as reentrant with a system call
- The OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program

Paging Summary

Paging is a big improvement over segmentation:

- Eliminates external fragmentation and need for compaction
- Permits sharing of code pages among processes, reducing overall memory requirements
- Enables processes to run when they are only partially loaded in main memory

However, paging has costs:

- Translating from a virtual address to a physical address is more time-consuming
- Paging requires hardware support in the form of a TLB to be efficient enough to actually use
- Paging requires more complex OS to maintain the page table