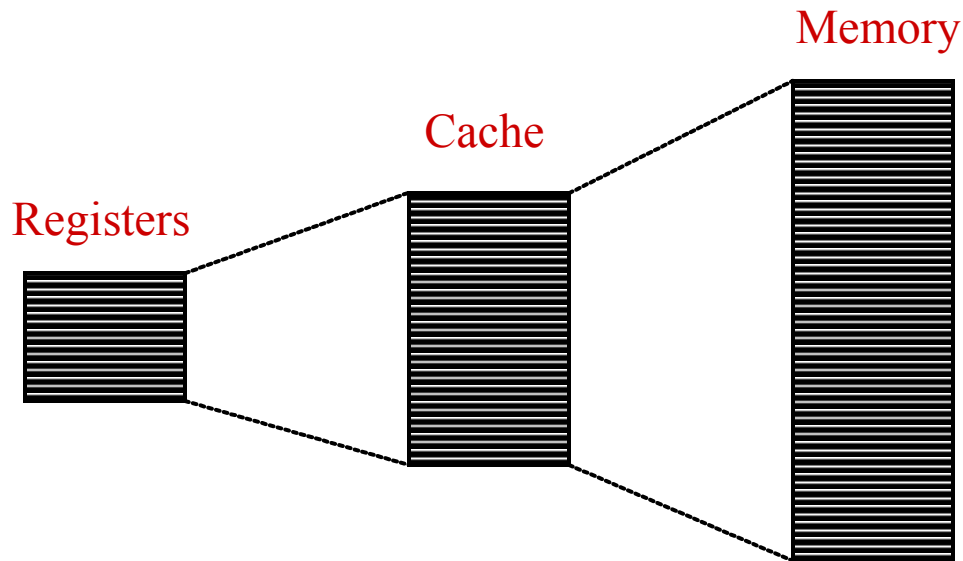


Memory Management

Operating Systems

Department of Computer Science
Rutgers University

Memory Hierarchy



Let's review how caches work as we'll need the terminology and concepts

As we move down the hierarchy, we ...

- decrease cost per bit
- decrease frequency of access
- increase capacity
- increase access time
- increase size of transfer unit

Memory Access Costs

Intel Pentium IV Extreme Edition (3.2 GHz, 32 bits)	Level	Size	Assoc	Block Size	Access Time
	L1	8KB	4-way	64B	2 cycles
	L2	512KB	8-way	64B	19 cycles
	L3	2MB	8-way	64B	43 cycles
	Mem				206 cycles
AMD Athlon 64 FX-53 (2.4 GHz, 64 bits, on-chip mem cntl)	L1	128KB	2-way	64B	3 cycles
	L2	1MB	16-way	64B	13 cycles
	Mem				125 cycles

Processors introduced in 2003

Memory Access Costs

Intel Core 2 Quad Q9450 (2.66 GHz, 64 bits)	Level	Size	Assoc	Block Size	Access Time
	L1	128KB	8-way	64B	3 cycles
	shared L2	6MB	24-way	64B	15 cycles
	Mem				229 cycles
Quad-core AMD Opteron 2360 (2.5 GHz, 64 bits)					
	L1	128KB	2-way	64B	3 cycles
	L2	512KB	16-way	64B	7 cycles
	shared L3	2MB	32-way	64B	19 cycles
	Mem				356 cycles

Processors introduced in 2008

Memory Access Costs

Quad-core Intel Core i7 3770K (3.5 GHz, 64 bits, integrated GPU, 22nm)	Level	Size	Access Time
	L1	64KB	4 cycles
	L2	256KB	12 cycles
	shared L3	8MB	19 cycles
	Mem		250 cycles

Processor introduced in 2012

Hardware Caches

Closer to the processor than the main memory

Smaller and faster than the main memory

Act as “attraction memory”: contain the value of main memory locations which were recently accessed (temporal locality)

Transfer between caches and main memory is performed in units called cache blocks/lines

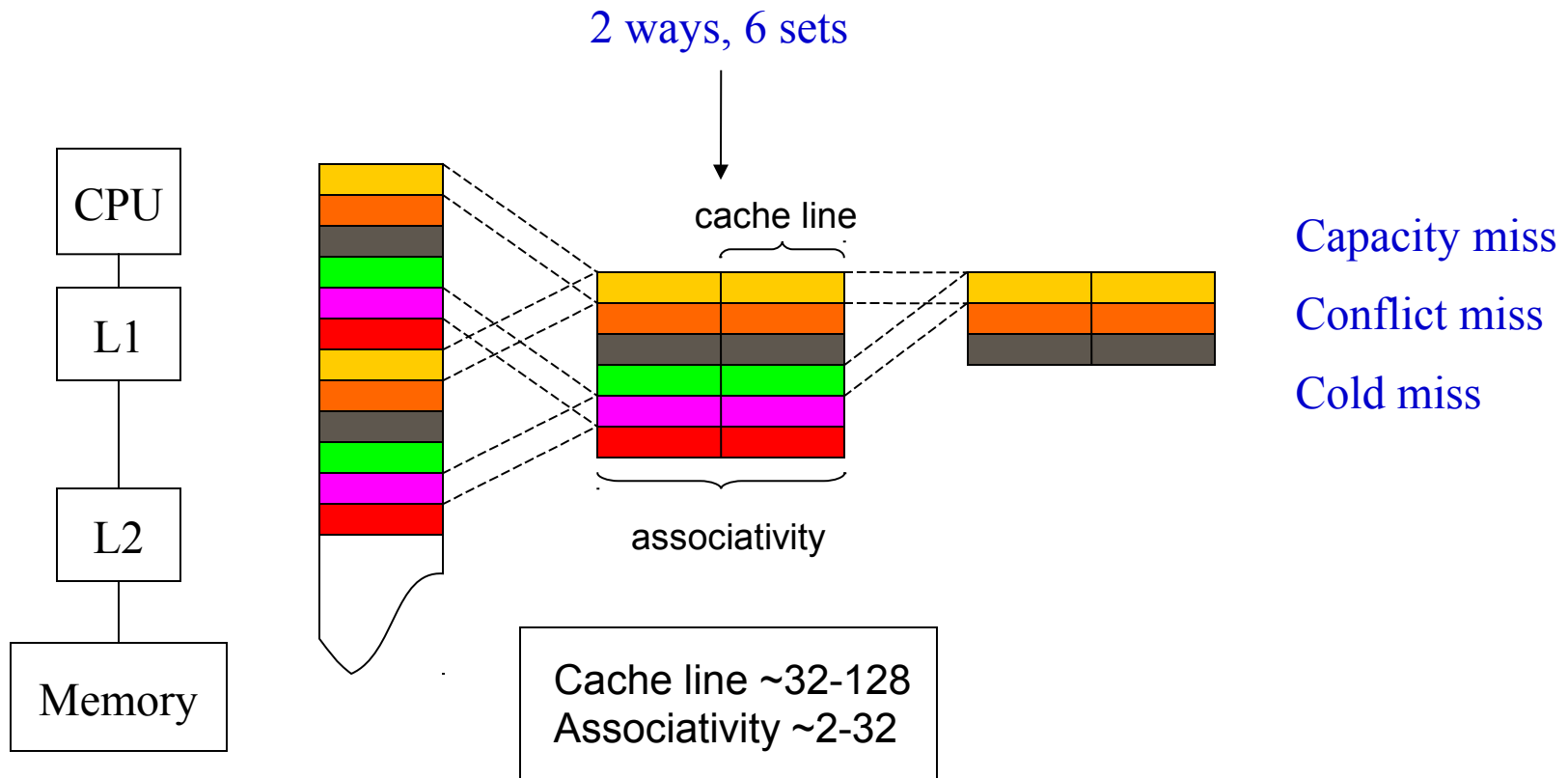
Caches also contain the value of memory locations that are close to locations that were recently accessed (spatial locality)

Mapping between memory and cache is (mostly) **static**

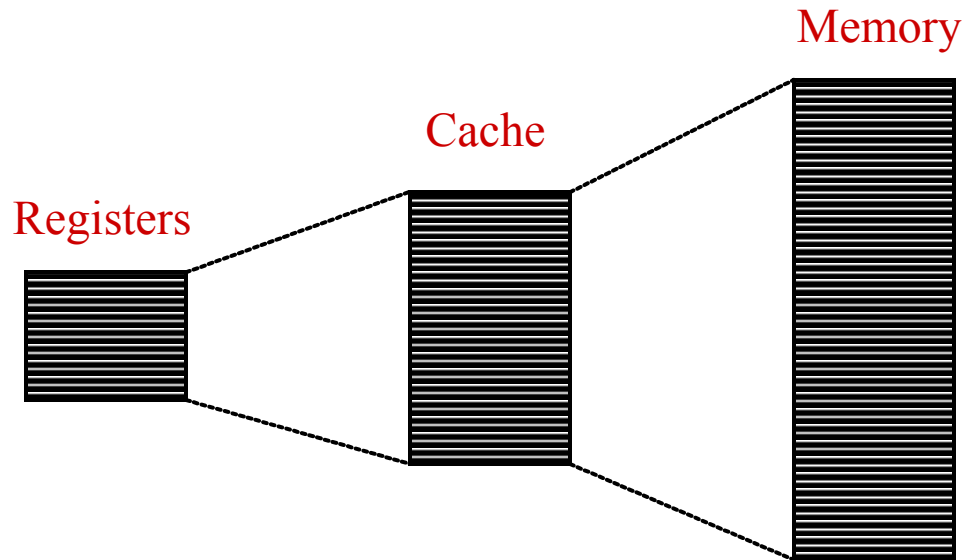
Fast handling of misses

Often L1 I-cache is separate from D-cache

Cache Architecture

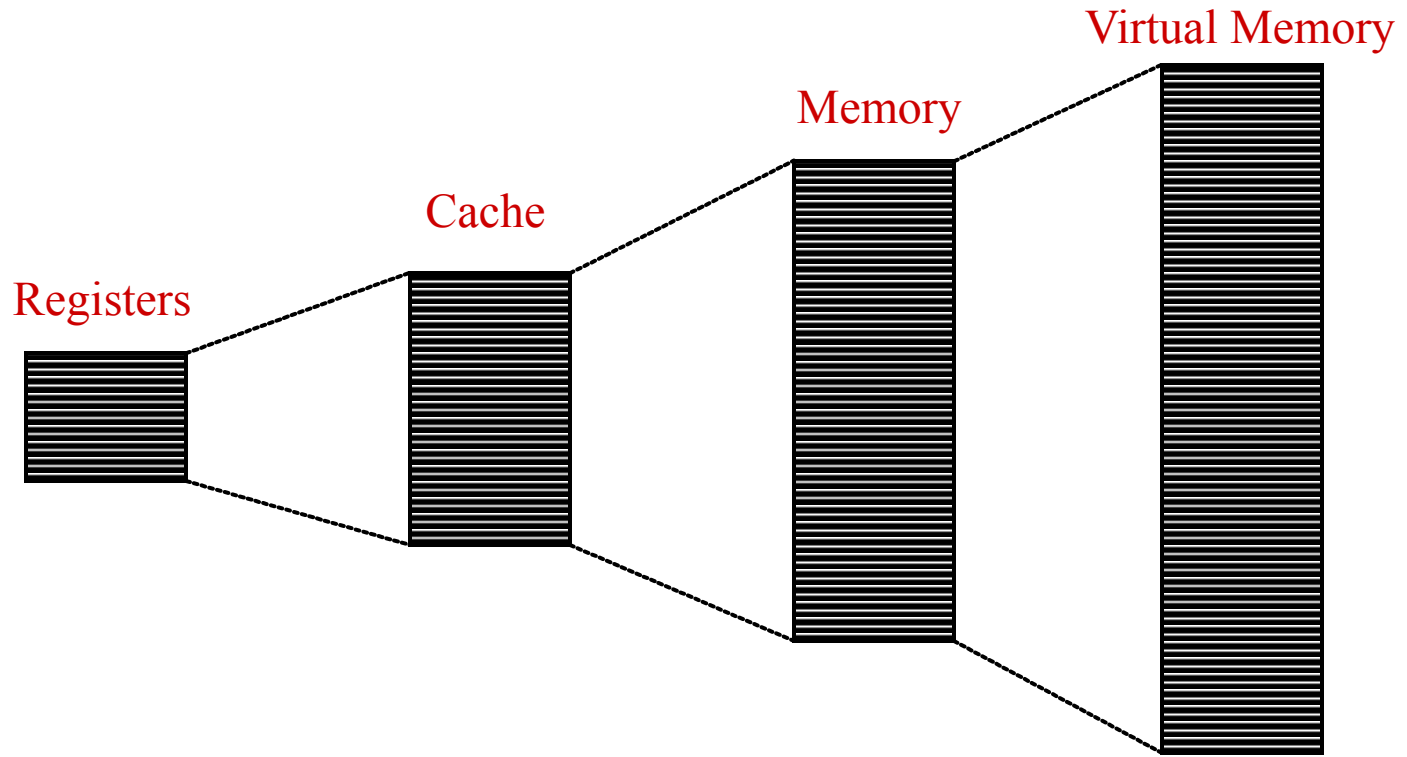


Memory Hierarchy



Question: What if we want to support programs that require more memory than what's available in the system?

Memory Hierarchy



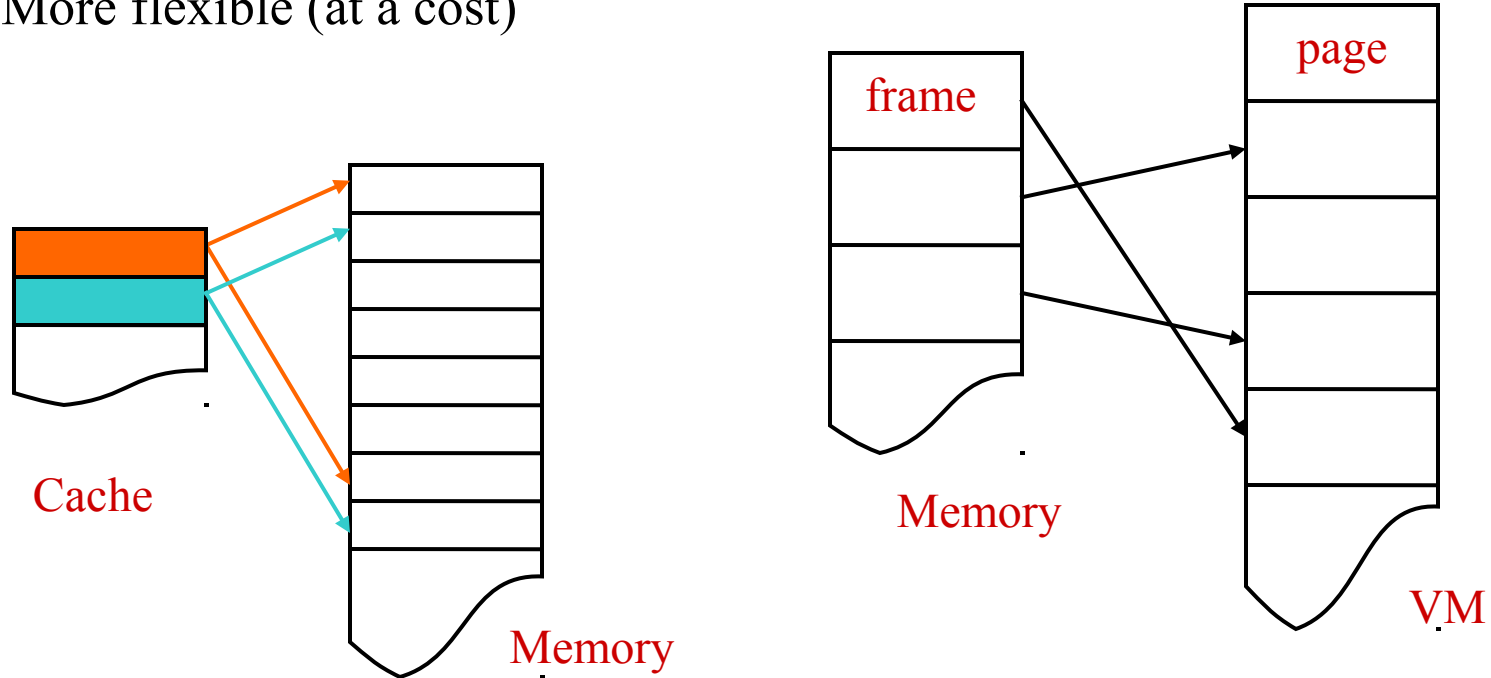
Answer: Pretend we had something bigger: Virtual Memory

Paging

A page is a cacheable unit of virtual memory

The OS controls the mapping between pages of VM and memory

More flexible (at a cost)



Starting from the beginning: Two Views of Memory

View from the hardware – shared physical memory

View from the software – what a process “sees”: private virtual address space

Memory management in the OS coordinates these two views

- Consistency: all address spaces should look “basically the same”

- Relocation: processes can be loaded at any physical address

- Protection: a process cannot maliciously access memory belonging to another process

- Sharing: may allow sharing of physical memory (must implement control)

Dynamic Storage Allocation Problem

How do we allocate processes in memory?

More generally, how do we satisfy a request of size n from a list of free holes?

First-fit: Allocate the first hole that is big enough.

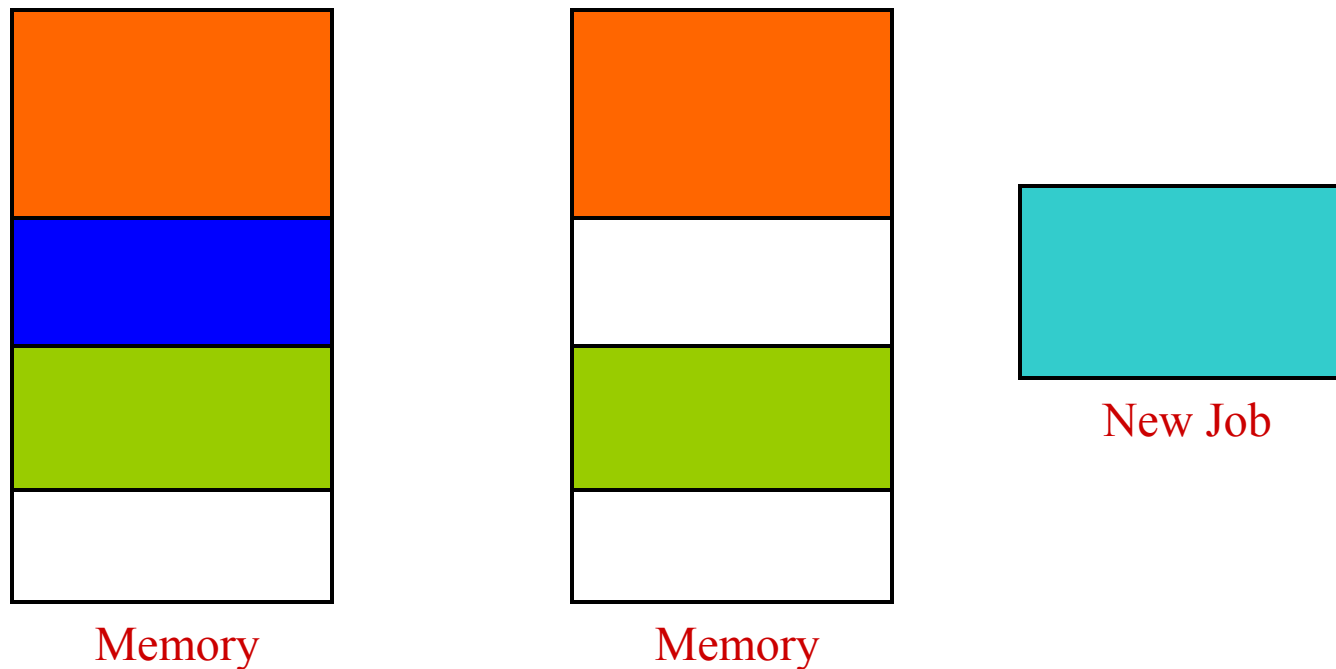
Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.

Worst-fit: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

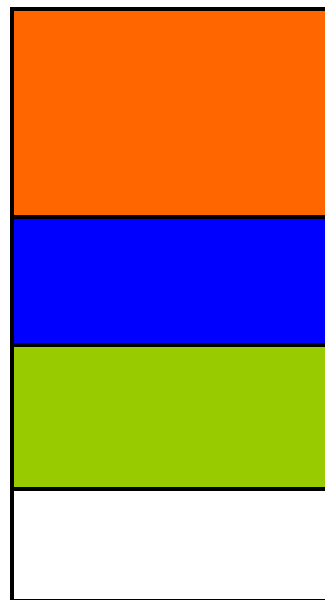
Fragmentation

Fragmentation: When entire processes are loaded into memory, there can be lots of unused memory space, but new jobs cannot be loaded

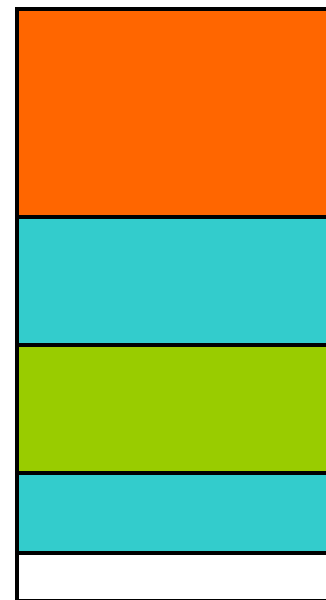


Paging From Fragmentation

Idea: Break processes into small, fixed-size chunks (pages), so that processes don't need to be contiguous in physical memory



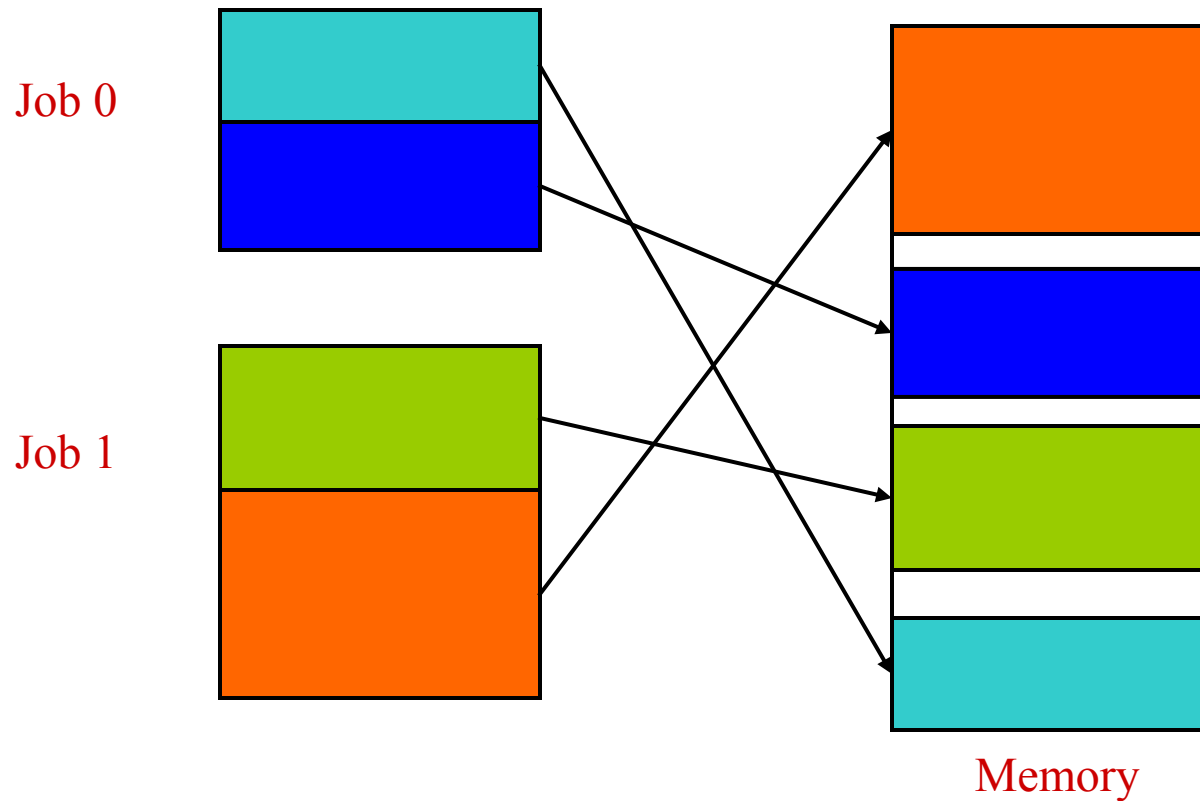
Memory



Memory

Segmentation

Segmentation: Same idea, but now variable-size chunks.



Virtual Memory

VM is the OS abstraction that provides the illusion of an address space that is contiguous and may be larger than the physical address space. Thus, impossible to load entire processes to memory

VM can be implemented using either paging or segmentation but paging is presently most common

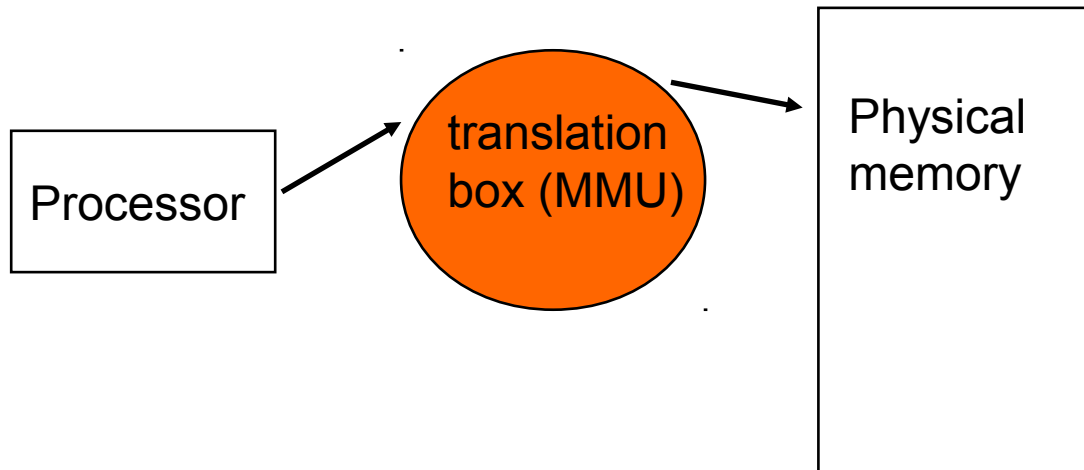
Actually, a combination is usually used but the segmentation scheme is typically very simple (e.g., a fixed number of variable-size segments)

VM is motivated by both

Convenience: (1) the programmer does not have to know where pages/segments are allocated in physical memory; and (2) the programmer does not have to deal with machines that have different amounts of physical memory

Fragmentation in multi-programming environments

Hardware Translation



Translation from virtual (aka logical) to physical addresses can be done in software but without protection

Why “without” protection?

Hardware support is needed for protection (& performance)

Simplest solution with two registers: base and size

Segmentation

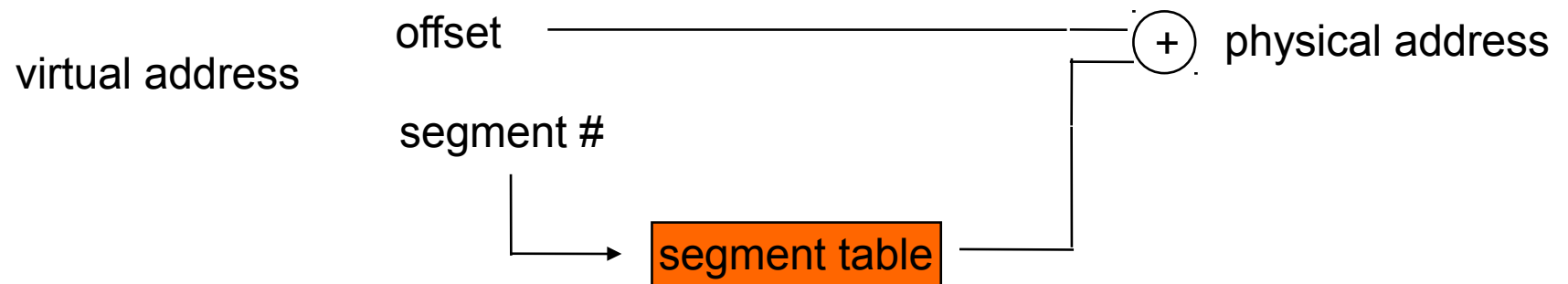
Segments are of variable size

Translation done through a set of (base, size, state) tuples - segment table indexed by segment number

State: valid/invalid, access permission, reference, modified bits

Segments may be visible to the programmer and can be used as a convenience for organizing the programs and data (e.g., code segment, global data segment)

Segmentation Hardware



Paging

Pages are of fixed size

The physical memory corresponding to a page is called a page frame

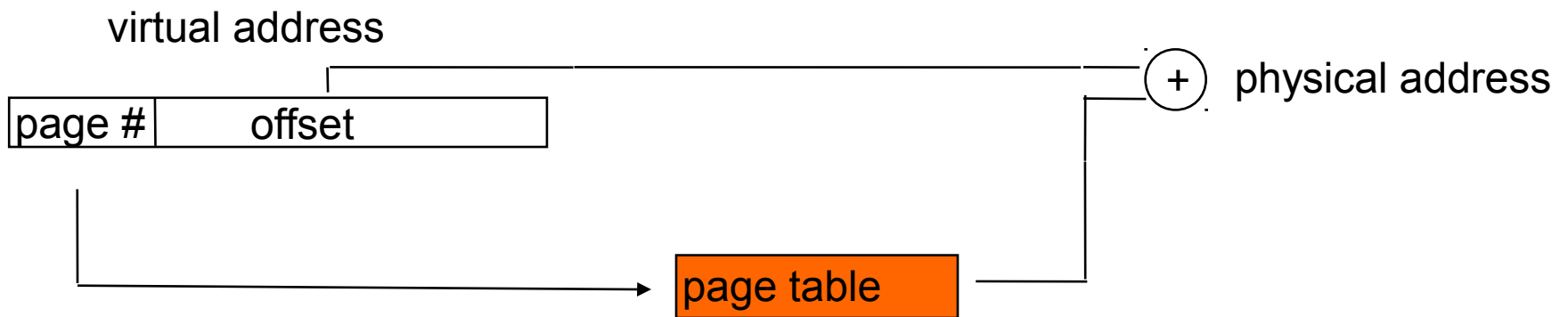
Translation done through a page table indexed by page number

Each entry in a page table contains the frame number that the virtual page is mapped to and the state of the page in memory

State: valid/invalid, access permission, reference, modified, caching bits

Paging is transparent to the programmer

Paging Hardware



Combined Paging and Segmentation

Some MMUs combine paging with segmentation

Virtual address: segment number + page number + offset

Segmentation translation is performed first

The segment entry points to a page table for that segment

The page number portion of the virtual address is used to index the page table and look up the corresponding page frame number

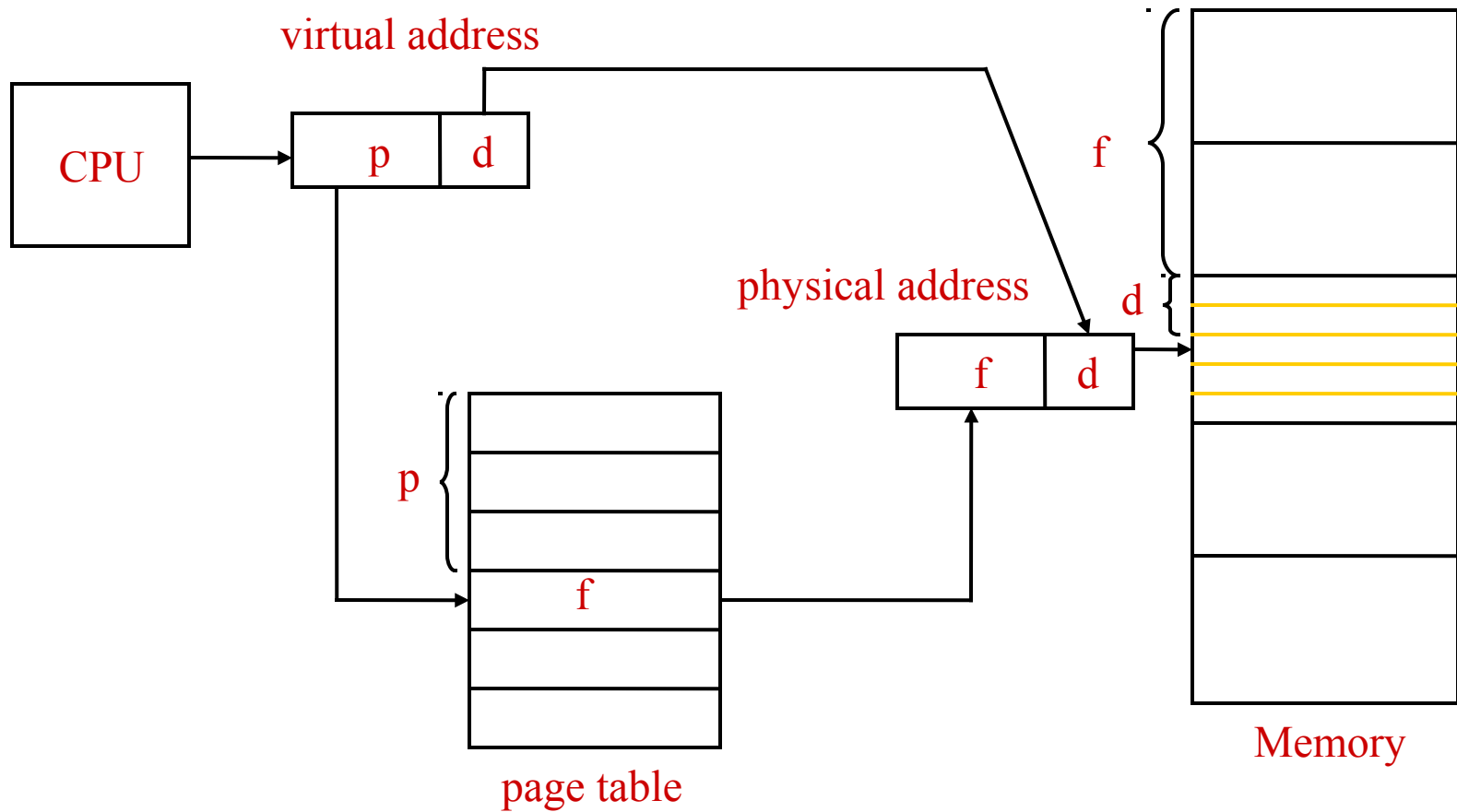
Example: 64-bit x86 uses segmentation and four-level paging

Segmentation not used much any more so we'll concentrate on paging

UNIX has a simple form of segmentation but does not require any hardware support

Example: Linux defines only four segments: kernel code, kernel data, user code, and user data. However, Linux “disables” segmentation on the x86 by setting segment base addresses = 0 and segment sizes = max memory size

Paging: Address Translation



Translation Lookaside Buffers

Translation on every memory access \Rightarrow must be fast

What to do?

Caching, of course ...

Why does caching work? That is, we still have to lookup the page table entry and use it to do translation, right?

Same as normal hardware cache – cache is smaller so can spend more \$\$ to make it faster

Translation Lookaside Buffer

Cache for page table entries is called the Translation Lookaside Buffer (TLB)

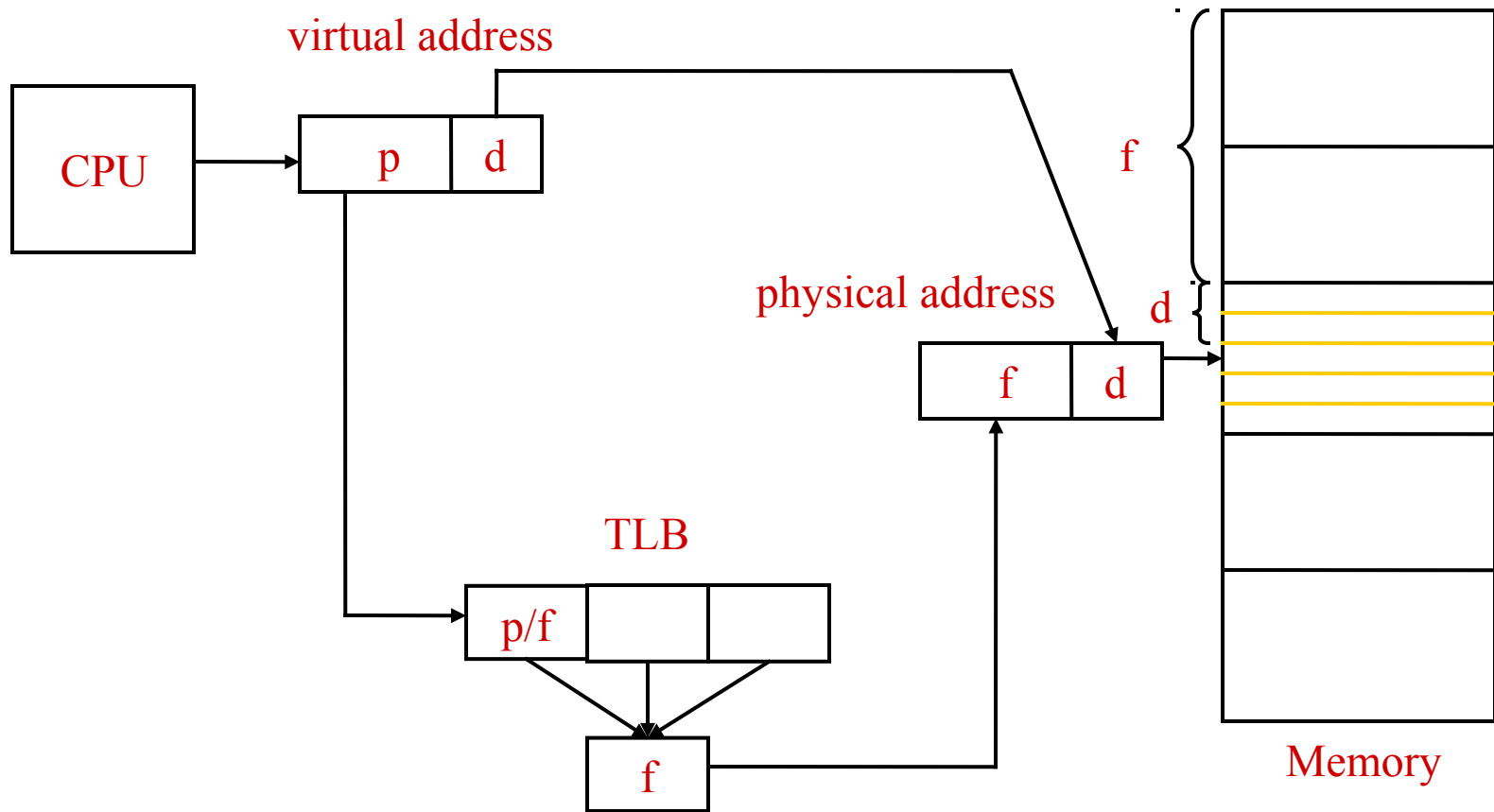
Traditionally, fully associative and single-level, but are becoming set associative and multi-level

Relatively small number of entries (e.g., Core i7 has 64 L1 and 512 L2 entries; both levels are 4-way associative)

Each TLB entry contains a page number and the corresponding PT entry

On each memory access, we look for the page \Rightarrow frame mapping in the TLB

Paging: Address Translation



TLB Miss

What if the TLB does not contain the right PT entry?

TLB miss

Evict an existing entry if does not have any free ones

Replacements? Pseudo-LRU common today: one bit represents each internal node of a binary search tree; cache lines are the leaves; an access sets the bits to the other direction in the tree

Bring in the missing entry from the PT

TLB misses can be handled in hardware (CISC, x86) or software (RISC, MIPS)

Software allows application to assist in replacement decisions

Where to Store Address Space?

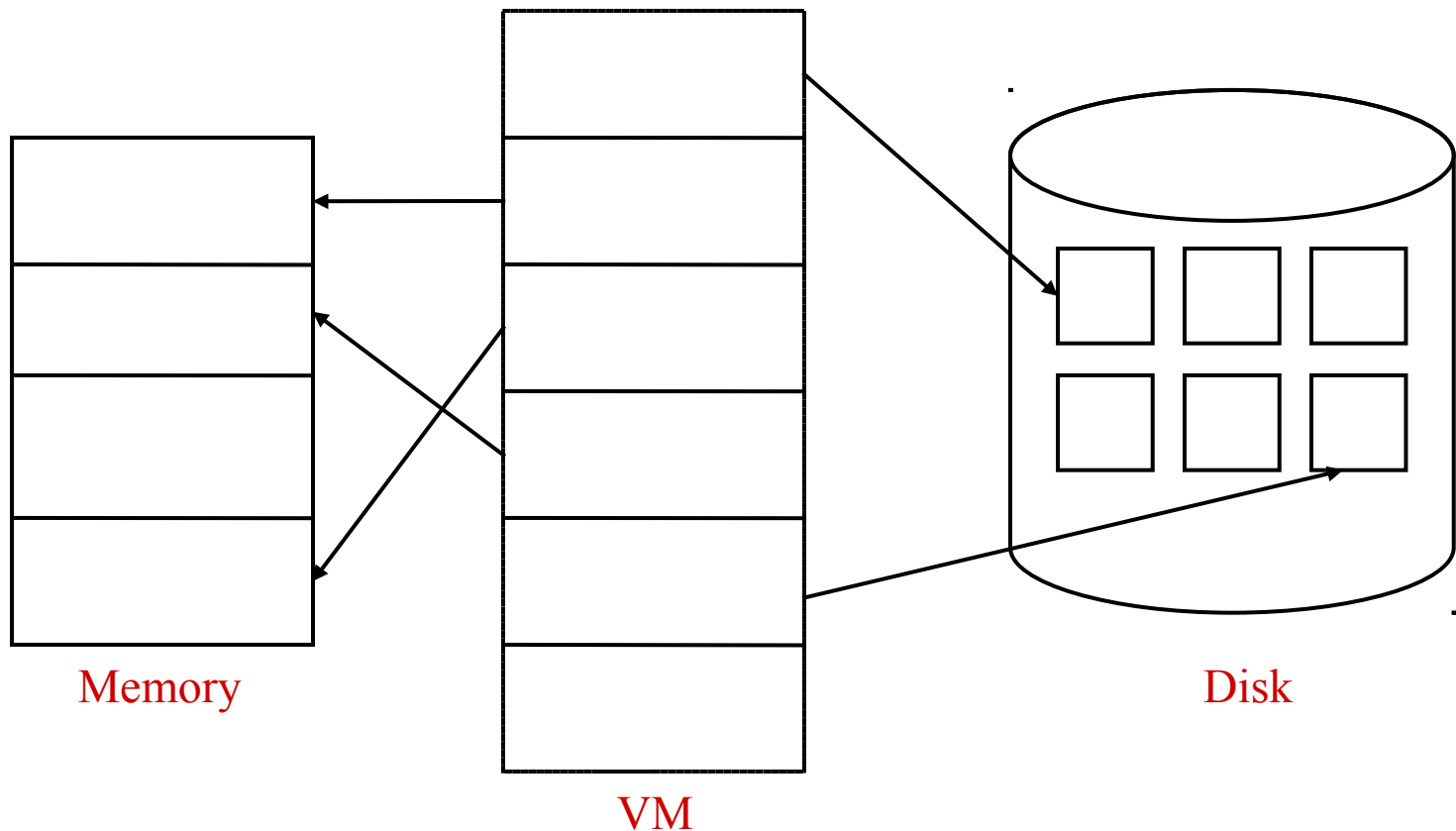
Virtual address space may be larger than physical memory

Where do we keep it?

Where do we keep the page table?

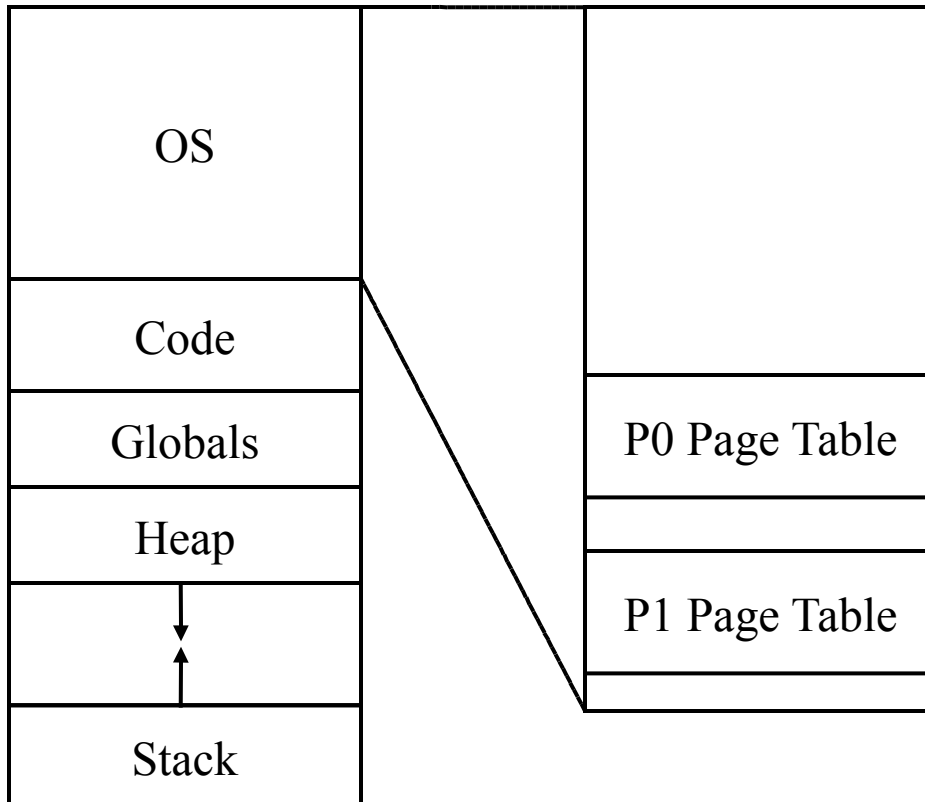
Where to Store Address Space?

On the next device down our memory hierarchy, of course ...



Where to Store Page Table?

In memory, of course ...



Interestingly, use memory to “enlarge” view of memory, leaving LESS physical memory

This kind of overhead is common

For example, OS uses CPU cycles to implement abstraction of threads

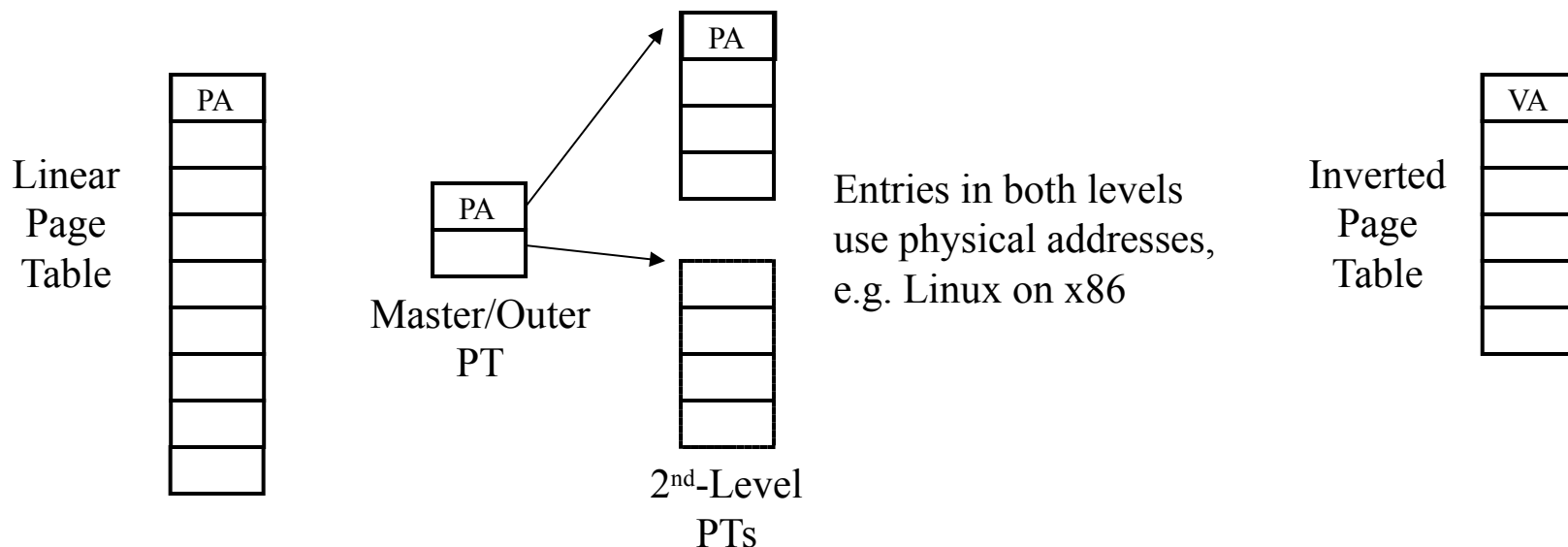
Got to know what the right trade-off is

Have to understand common application characteristics

Have to be common enough!

Page tables can get large. What to do?

Page Table Structure

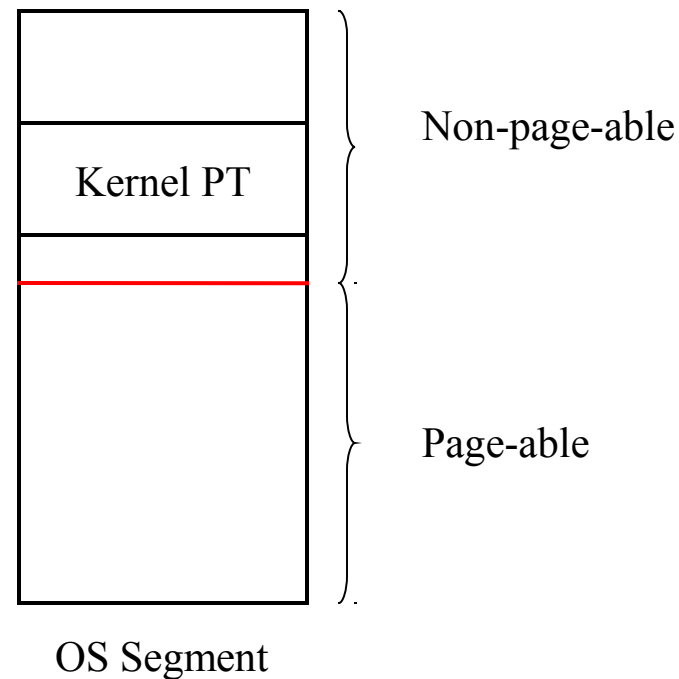
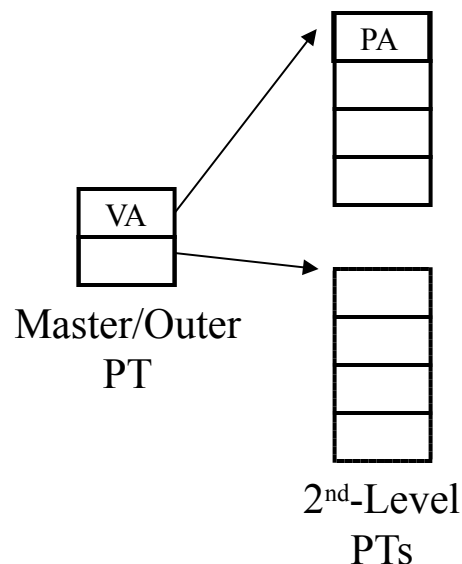


Linear (aka conventional) page table can become huge. What to do?

Two-Level PT: page the page table. Saves space by only allocating 2nd-level tables for virtual memory that has actually been allocated/touched (outer page table is always in memory). Doesn't need a large contiguous chunk of physical memory.

Inverted PT: Saves a lot of space, as it requires (a single chunk of) memory in proportion to the size of the physical address space. Translation through hash table

Page Table Structure



of the entries in the outer page table name virtual addresses...

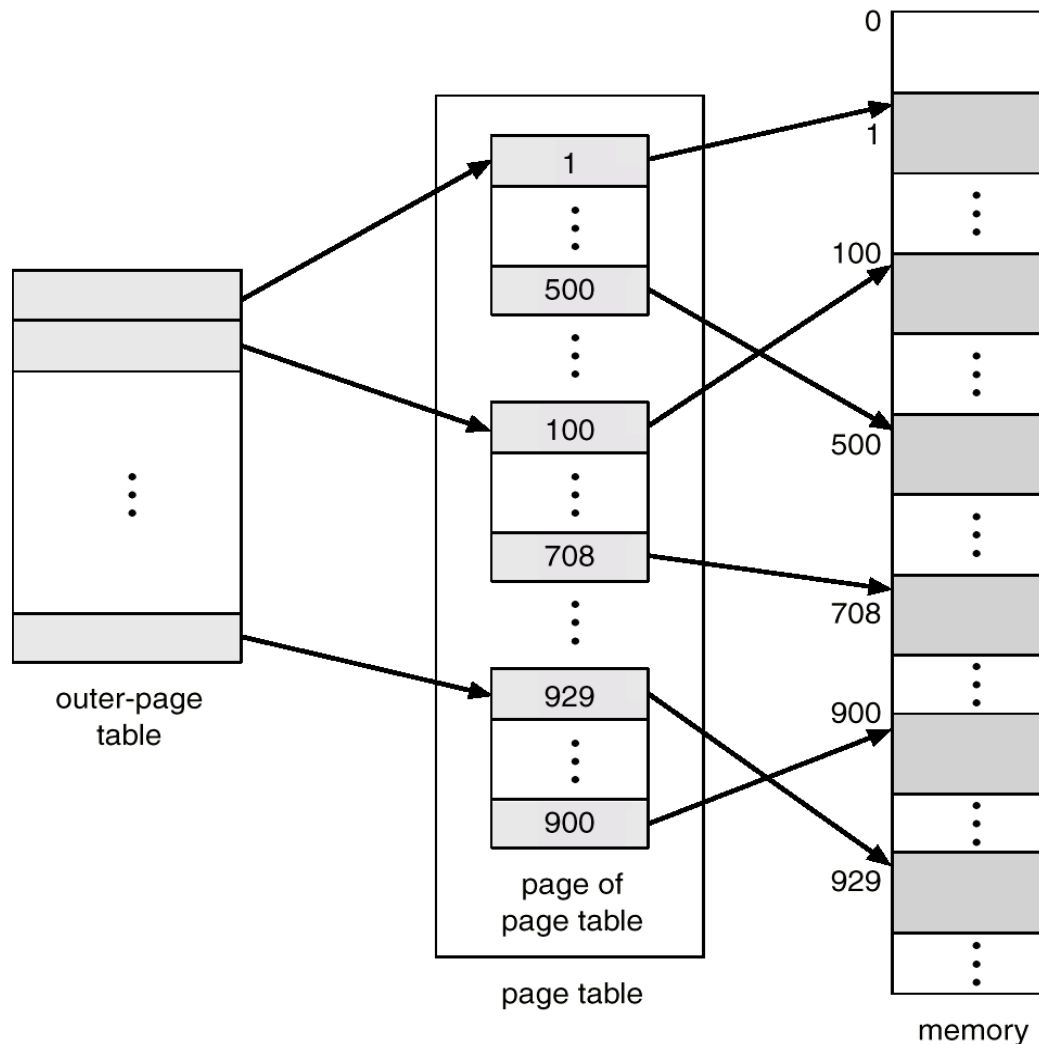
We need one more level of translation

kernel can take advantage of paging

Kernel's (outer) PT must remain in memory

Other parts that cannot be paged: page fault code, performance-critical code (e.g., interrupt handlers), pages undergoing I/O

Two-Level Page Table Scheme



Two-Level Paging Example

A virtual 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.

Two-Level Paging Example

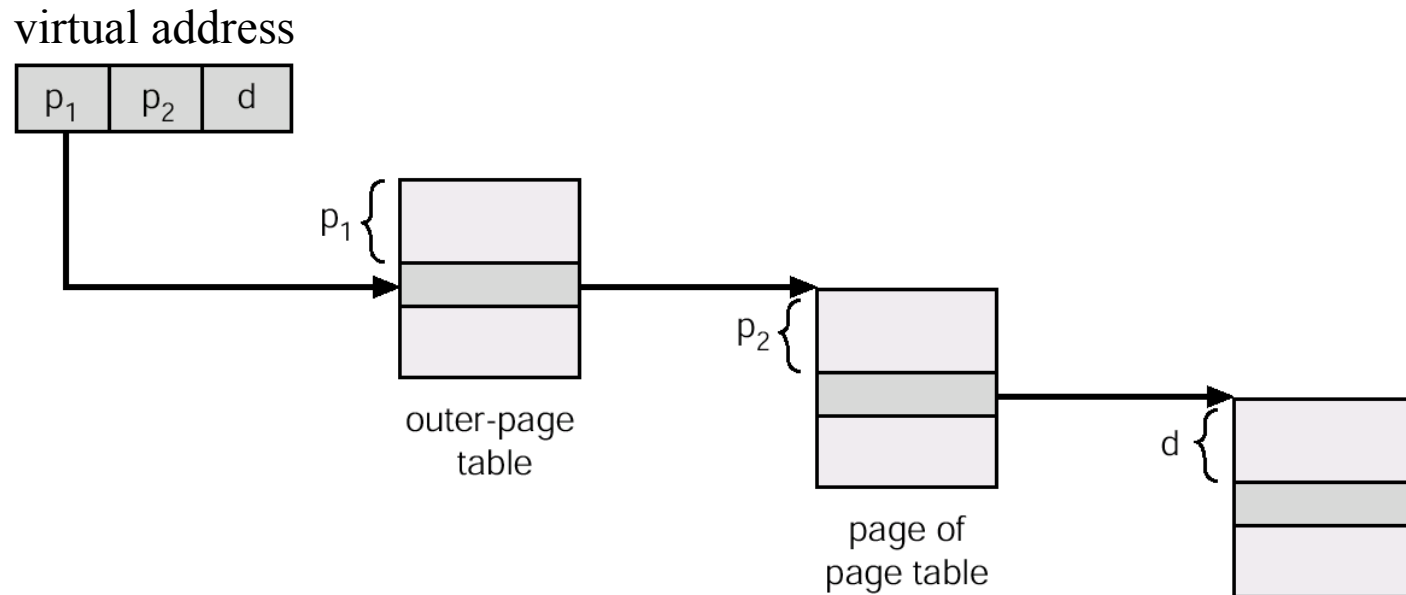
Thus, a virtual address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page to which the outer page table points.

Address Translation Scheme

Address translation scheme for a two-level 32-bit paging architecture



Multilevel Paging and Performance

Since each level is stored as a separate table in memory, translating a virtual address to a physical one may take n memory accesses for n -level page tables.

Nevertheless, caching keeps performance reasonable.

TLB hit rate of 98%, TLB access time of 2 ns, memory access time of 120 ns, and a 2-level PT yield:

$$\begin{aligned}\text{effective access time} &= 0.98 \times (2+120) + 0.02 \times (2+360) \\ &= 127 \text{ nanoseconds.}\end{aligned}$$

which is only a 6% slowdown in memory access time.

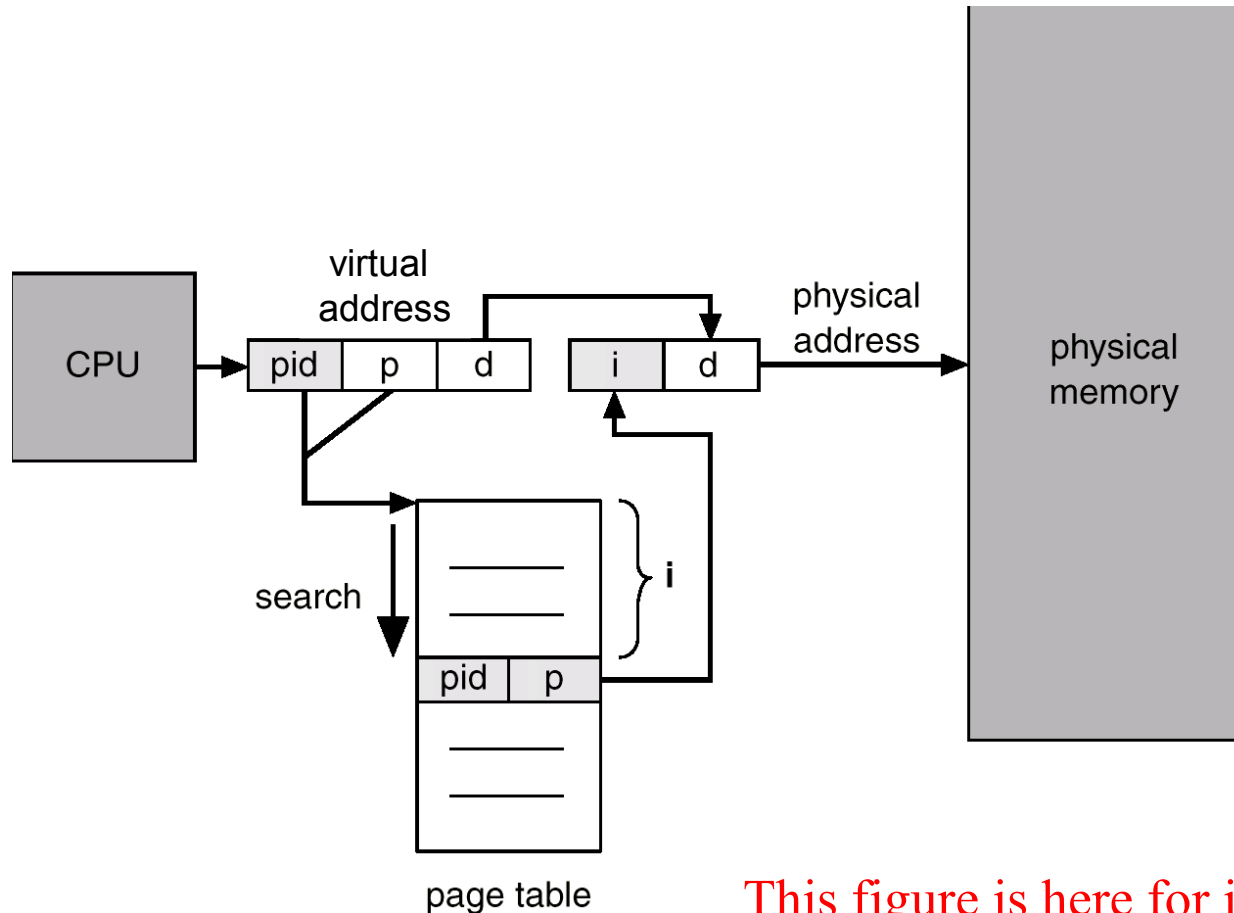
Inverted Page Table

One entry for each real memory frame.

Entry consists of the virtual address of the page stored in that frame, with information about the process that owns the page. Translations happen as shown in the following figure.

Inverted pages tables are used in the 64-bit UltraSPARC and PowerPC architectures.

Inverted Page Table Architecture



This figure is here for intuition;
searching is not used in practice.

Inverted Page Table with Hashing

This implementation of IPTs decreases the memory needed to store the PT, but increases the time needed to search it when a page reference occurs. For this reason, **searching is not used**.

Can use a hash table to limit the search to one — or at most a few — PT entries. Under hashing, each PT entry also has to include a frame number and a chain pointer.

The approach works by hashing the virtual page number + pid. The result indexes the hash table. Each entry of the hash table stores a pointer to the first entry of the chain. The virtual page number of each entry is compared to the referenced page number and, on a match, the corresponding frame number is used.

How to Deal with $VM > \text{Size of Physical Memory}$?

If the address space of each process is \leq the size of the physical memory, then no problem

Still useful to tackle fragmentation & provide a contiguous address space

When VM larger than physical memory

Part stored in memory

Part stored on disk

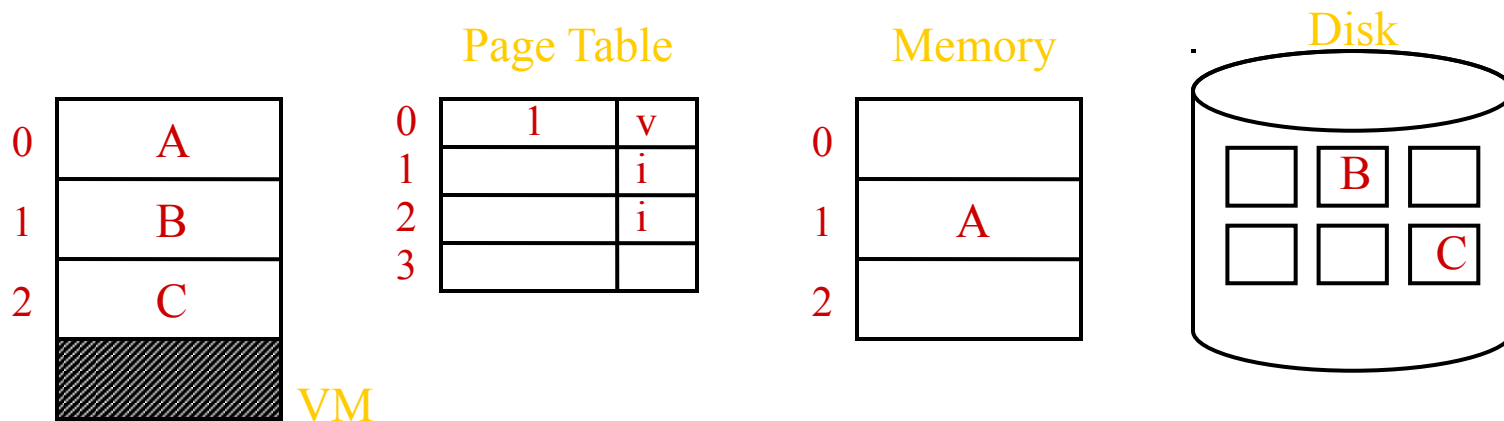
How do we make this work?

Demand Paging

To start a process (program), just load the code page where the process will start executing

As the process references memory (instruction or data) outside of the loaded page, bring in as necessary

How to represent fact that a page of VM is not yet in memory?



Page Fault

What happens when the process references a page marked as invalid in the page table?

Page fault exception

Check that the reference is valid

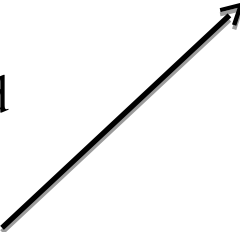
Find a free memory frame

Read the desired page from disk

Update the page table entry to point to the frame

Change the valid bit of the page to v

Restart the instruction that was interrupted by the exception



Text and data pages come from disk. Stack and heap pages are allocated in main memory first. Shared library pages may already be in main memory.

What happens if there is no free frame?

Cost of Handling a Page Fault

Exception, check the page table, find a free memory frame (or find victim) ...
about 200 - 600 μ s

Disk seek and read ... about 10 ms

Memory access ... about 100 ns

Page fault degrades performance by $\sim 1000000!!!!$

This doesn't even count all the additional things that can happen along the way

Better not have too many page faults!

If we want no more than 10% degradation, we can only have 1 page fault for every 1,000,000 memory accesses

OS better do a great job of managing the movement of data between secondary storage and main memory

Page Replacement

What if there's no free frame left on a page fault?

Free a frame that's currently being used

Select the frame to be replaced (victim)

Write the victim back to the disk

Change the page table to reflect that the victim is now invalid

Read the desired page into the newly freed frame

Change PT: the new page is in the freed frame and is now valid

Restart the faulting instruction

Optimization: do not need to write the victim back if it has not been modified (need a dirty bit per page).

Page Replacement

Highly motivated to find a good replacement policy

That is, when evicting a page, how do we choose the best victim in order to minimize the page fault rate?

Is there an optimal replacement algorithm?

If yes, what is the optimal page replacement algorithm?

Let's look at an example:

Suppose we have 3 memory frames and are running a program that has the following reference pattern

7, 0, 1, 2, 0, 3, 0, 4, 2, 3

Suppose we know the reference pattern in advance ...

Page Replacement

Suppose we know the access pattern in advance

7, 0, 1, 2, 0, 3, 0, 4, 2, 3

The optimal algorithm is to replace the page that will not be used for the longest period of time

What's the problem with this algorithm?

Realistic policies try to predict the future behavior on the basis of the past behavior

Works because of temporal locality

FIFO

First-in, First-out

Be fair, let every page live in memory for about the same amount of time, then toss it.

What's the problem?

Is this compatible with what we know about behavior of programs?

How does it do on our example?

7, 0, 1, 2, 0, 3, 0, 4, 2, 3

LRU

Least Recently Used

On access to a page, timestamp it

When need to evict a page, choose the one with the oldest timestamp

What's the motivation here?

Is LRU optimal?

In practice, LRU is quite good for most programs

Is it easy to implement?

Not Frequently Used Replacement

Have a reference bit (aka use bit) and software counter for each page frame

At each clock interrupt, the OS adds the reference bit of each frame to its counter and then clears the reference bit

When need to evict a page, choose the frame with the lowest counter value

What's the problem?

Doesn't forget anything, no sense of time – hard to evict a page that was referenced a lot sometime in the past but is no longer relevant to the computation

Updating counters is expensive, especially since memory is getting rather large these days

Can be improved with an aging scheme: counters are shifted right before adding the reference bit and the reference bit is added to the leftmost bit (rather than to the rightmost one)

Clock (Second-Chance)

Arrange frames in a circle, with a clock hand (initially points to the first frame)

Hardware keeps 1 use bit per frame. Sets the use bit on a memory reference to the corresponding frame.

If bit is not set, the frame hasn't been used for a while

On a page fault:

Advance the clock hand

Check the use bit

If 1, has been used recently, clear and go on

If 0, this is our victim

Can we always find a victim?

Nth-Chance

Similar to Clock except: maintain a counter as well as a use bit

On a page fault:

- Advance the clock hand

- Check the use bit

 - If 1, clear and set the counter to 0

 - If 0, increment the counter, if the counter $< N$, go on, otherwise, this is our victim

Why?

- N larger \Rightarrow better approximation of LRU

What's the problem if N is too large?

A Different Implementation of 2nd-Chance

Always keep a free list of some size $n > 0$

On a page fault, if the free list has more than n frames, get a frame from the free list and read the desired page into the frame

If the free list has only n frames, get a frame from it, read the desired page into the frame, then (in the background) choose a victim from the frames currently being used and put the frame on the free list

On a page fault, if the page is on a frame on the free list, don't have to read the page back in.

Works well, gets performance close to true LRU

Multi-Programming Environment

Why?

Better utilization of resources (CPU, disks, memory, etc.)

Problems?

Mechanism – TLB?

Fairness?

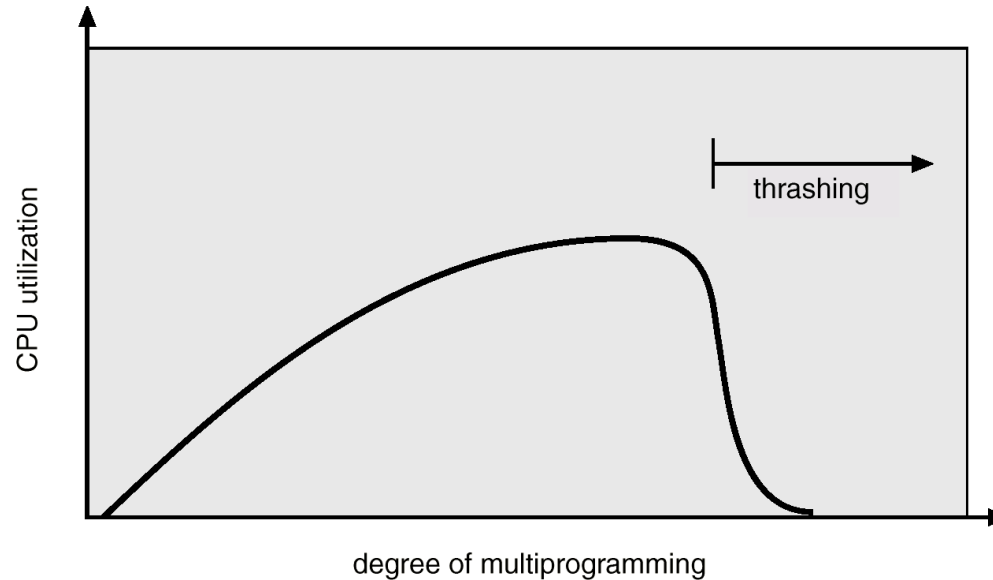
Over commitment of memory

What's the potential problem?

Each process needs its *working set* to be in memory to perform well

If too many processes are running, the system can thrash

Thrashing Diagram



Why does paging work? Locality

Process migrates from one locality (working set) to another

Why does thrashing occur?

Σ size of working sets $>$ total memory size

Support for Multiple Processes

More than one address space can be loaded in memory

A CPU register points to the current page table

OS updates the register when context switching between threads from different processes

Most TLBs can cache more than one PT

Store the process id to distinguish between virtual addresses belonging to different processes

If TLB caches only one PT then it must be flushed at process-switch time

Switching Between Threads of Different Processes

What if switching to a thread of a different process?

Caches, TLB, page table, etc.?

Caches

Physical addresses: no problem

Virtual addresses: cache must either have process tag or must flush cache on context switch

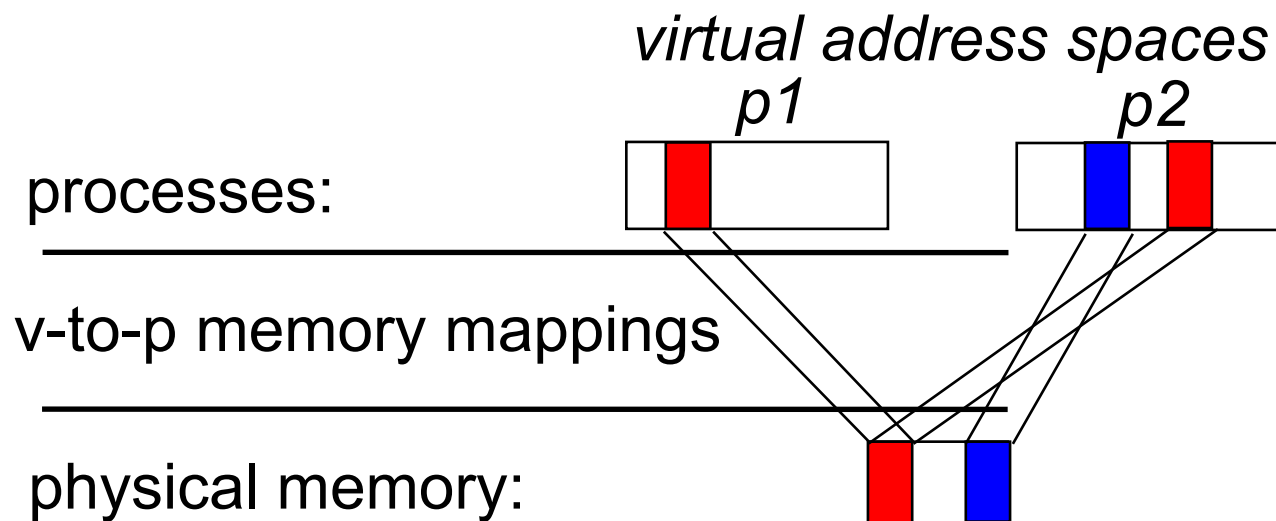
TLB

Each entry must have process tag or must flush TLB on switch

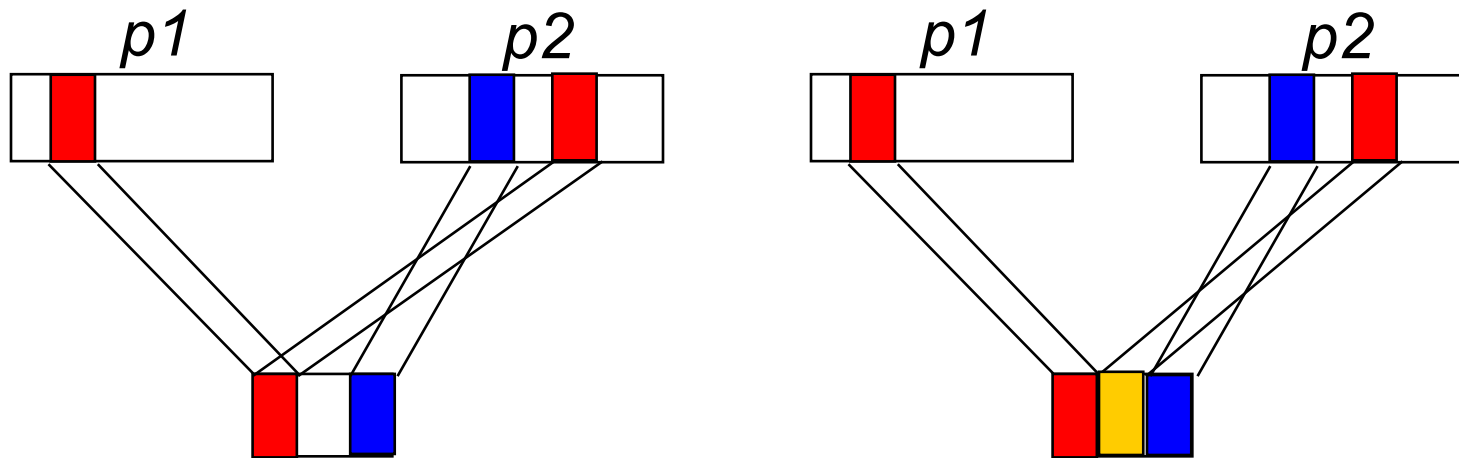
Page table

Page table pointer (register) must be reloaded on switch

Sharing



Copy-on-Write



Resident Set Management

How many pages of a process should be brought in?

Resident set size can be fixed or variable

Replacement scope can be local or global

Most common schemes implemented in the OS:

Variable allocation with global scope: simple, but replacement policy may not take working set issues into consideration, i.e. may replace a page that is currently in the working set of a process

Variable allocation with local scope: more complicated – from time to time, modify resident set size to approximate the working set size

Working Set

Working set is set of pages that have been referenced in the last window of time

The size of the working set varies during the execution of the process depending on the locality of accesses

If the number of frames allocated to a process covers its working set then the number of page faults is small

Schedule a process only if there is enough free memory to load its working set

How can we determine/approximate the working set size? We will see how soon...

Working-Set Model

$\Delta \equiv$ working-set window \equiv number of “virtual” time units, i.e. time elapsed while the process is actually executing

WSS_i (working set size of process P_i) =
total # of pages referenced in the most recent Δ (varies in time)

if Δ too small will not encompass entire locality.

if Δ too large will encompass several localities.

if $\Delta = \infty \Rightarrow$ will encompass entire program.

$D = \sum WSS_i \equiv$ total demand for frames

if $D > M$ (memory size) \Rightarrow Thrashing. We should suspend one of the processes. But which one?

Working-Set Model

$\Delta \equiv$ working-set window \equiv number of “virtual” time units, i.e. time elapsed while the process is actually executing

WSS_i (working set size of process P_i) =
total # of pages referenced in the most recent Δ (varies in time)

if Δ too small will not encompass entire locality.

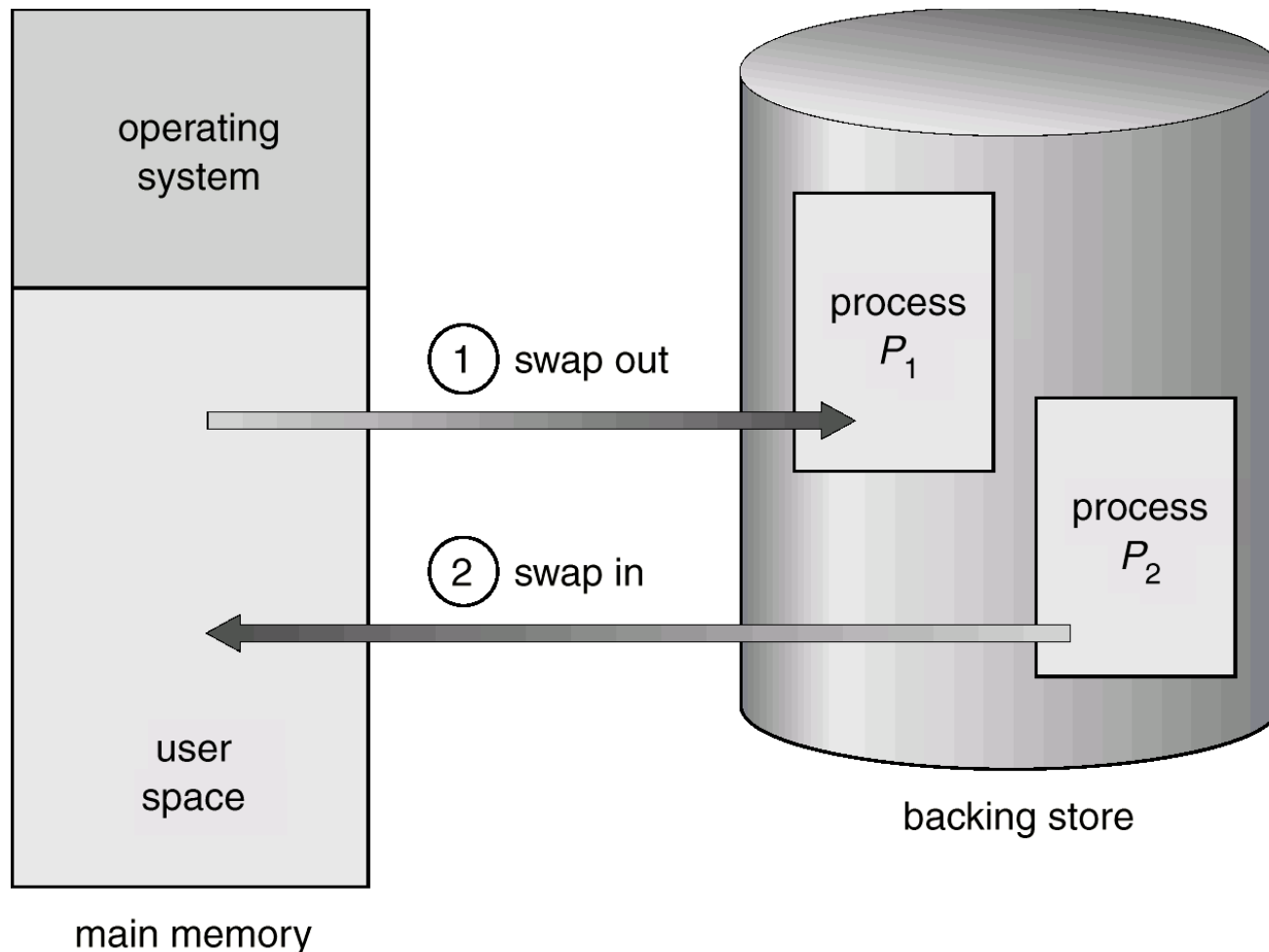
if Δ too large will encompass several localities.

if $\Delta = \infty \Rightarrow$ will encompass entire program.

$D = \sum WSS_i \equiv$ total demand for frames

if $D > M$ (memory size) \Rightarrow Thrashing. We should suspend one of the processes. But which one? Lowest priority, smallest resident set, last process activated, ...

Suspending and Reactivating Processes



An Approach to Keeping Track of the Working Set

Approximate with interval timer + a reference bit

Example: $\Delta = 10000$ cycles

Timer interrupts after every 5000 cycles.

Keep in memory 2 bits for each page.

When interrupted, copy and later reset all reference bits.

If one of the copied reference bits = 1 \Rightarrow page in working set.

Why is this not completely accurate?

An Approach to Keeping Track of the Working Set

Approximate with interval timer + a reference bit

Example: $\Delta = 10000$ cycles

Timer interrupts after every 5000 cycles.

Keep in memory 2 bits for each page.

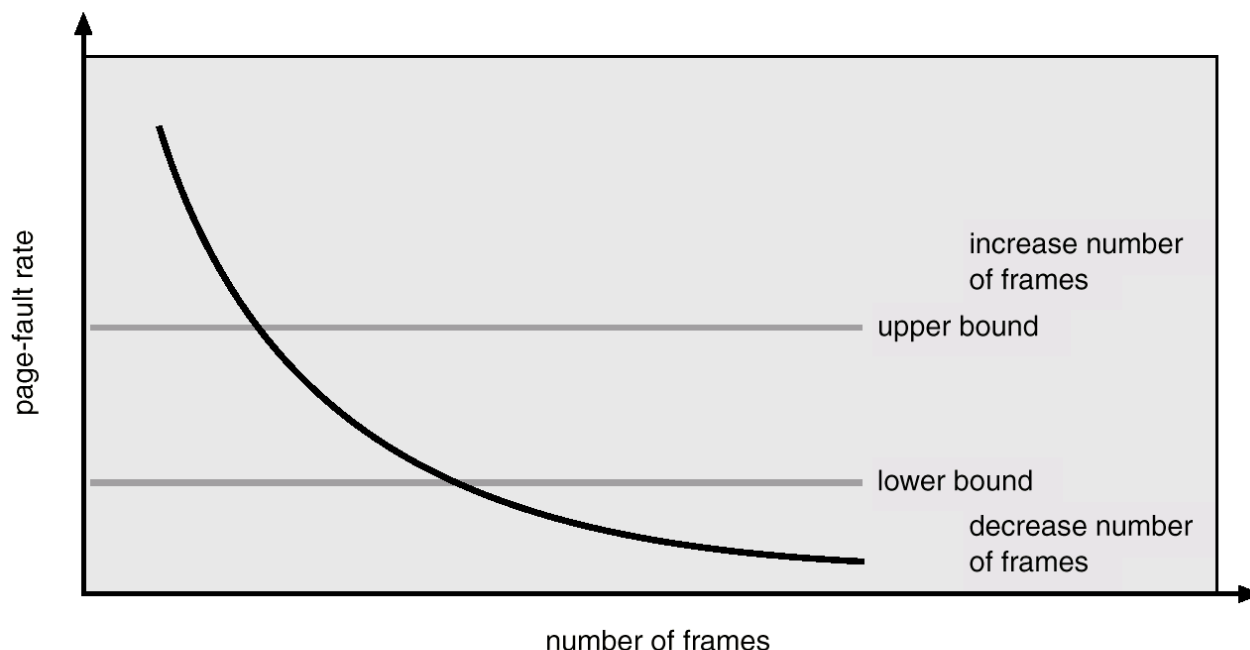
When interrupted, copy and later reset all reference bits.

If one of the copied reference bits = 1 \Rightarrow page in working set.

Why is this not completely accurate? Does not say when a reference occurs during the 5000 cycle interval.

Improvement? 10 bits and interrupt every 1000 time units. Cost of more frequent interrupts is higher.

Another Approach: Page Fault Frequency Scheme



Establish an “acceptable” page fault rate.

If the actual rate is too low (wrt threshold), the process loses a frame.

If the actual rate is too high (wrt threshold), the process gains a frame.

Page Fault Frequency

A counter per process stores the virtual time between page faults

An upper threshold for the virtual time is defined

On a page fault, if the amount of time since the last fault is less than the threshold (i.e., page faults are happening at a high rate), the new page is added to the resident set

A lower threshold can be used in a similar fashion to discard pages from the resident set

Resident Set Management

What is the problem with the management policies that we have just discussed?

Resident Set Management

What is the problem with the management policies that we have just discussed? Policies deal with stable and transient (going from one locality or working set to another) periods in the same way.

During transient periods, we would like to change timer intervals or paging rate thresholds, so that the resident set of the process does not grow excessively.

Exercise: Can you think of a policy that does that? How do we determine that we are in a transient period?

Summary

Virtual memory is a way of introducing another level in our memory hierarchy in order to abstract away the amount of memory actually available on a particular system

This is very important for “ease-of-programming”

Allows contiguous address spaces that are larger than physical memory, as well as protection

Can be implemented using paging (sometimes segmentation)

Page fault is expensive so can't have too many of them

Important to implement good page replacement policy

Have to watch out for thrashing!!

Other Considerations

Prepaging vs. demand paging

Page size selection has to balance

- Fragmentation

- Page table size

- I/O overhead

- Locality

TLB reach can be increased by using

- Larger pages

- More entries

Other Considerations (Cont.)

Program structure vs. locality

Array A[1024, 1024] of integer

Each row is stored in one page

Assume a single memory frame

```
Program 1  for  $j := 1$  to 1024 do  
           for  $i := 1$  to 1024 do  
               A[ $i, j$ ] := 0;
```

1024 x 1024 page faults

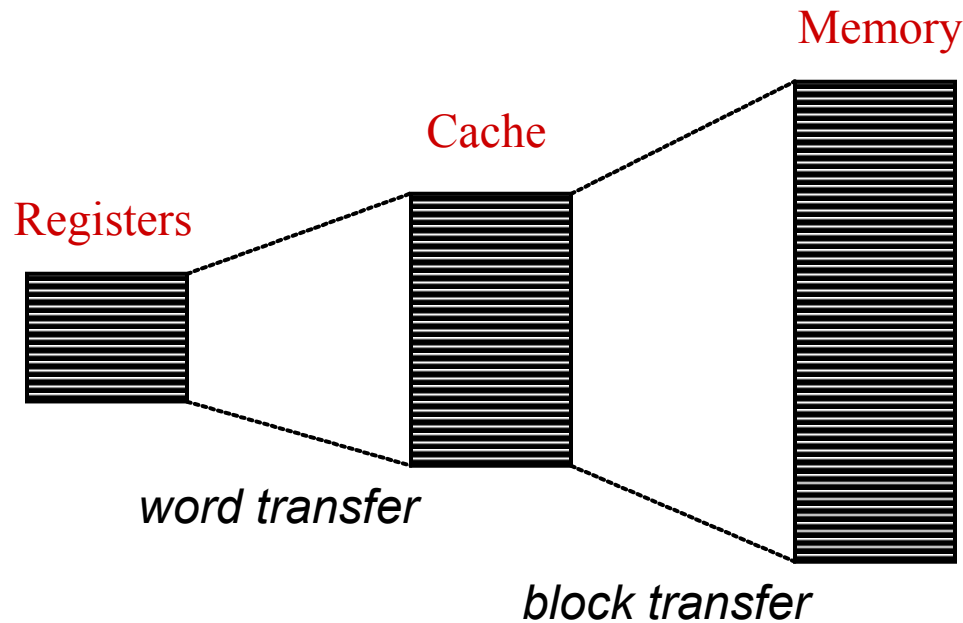
```
Program 2  for  $i := 1$  to 1024 do  
           for  $j := 1$  to 1024 do  
               A[ $i, j$ ] := 0;
```

1024 page faults

Pinning pages to memory for I/O

Backup Slides

Cache Design Issues



Cache size and cache block size

Mapping: physical/virtual caches, associativity

Replacement algorithm: random or (pseudo) LRU

Write policy: write through/write back

Overview (TLB Misses in Software)

So, what can happen on a memory access (pageable PT and TLB misses handled in software)?

TLB miss \Rightarrow read page table entry

TLB miss \Rightarrow read kernel page table entry

Page fault for necessary page of process page table

All frames are used \Rightarrow need to evict a page \Rightarrow modify a process page table entry

TLB miss \Rightarrow read kernel page table entry

Page fault for necessary page of process page table

Go back to finding a frame

Read in needed page, modify page table entry, fill TLB

Overview (TLB Misses in Hardware)

So, what can happen on a memory access (pageable PT and TLB misses handled in hardware)?

TLB miss \Rightarrow read page table entry

Page fault for necessary page

All frames are used \Rightarrow need to evict a page \Rightarrow modify a process page table entry

TLB miss \Rightarrow read kernel page table entry

Page fault for necessary page of process page table

Go back to finding a frame

Read in needed page, modify page table entry, fill TLB

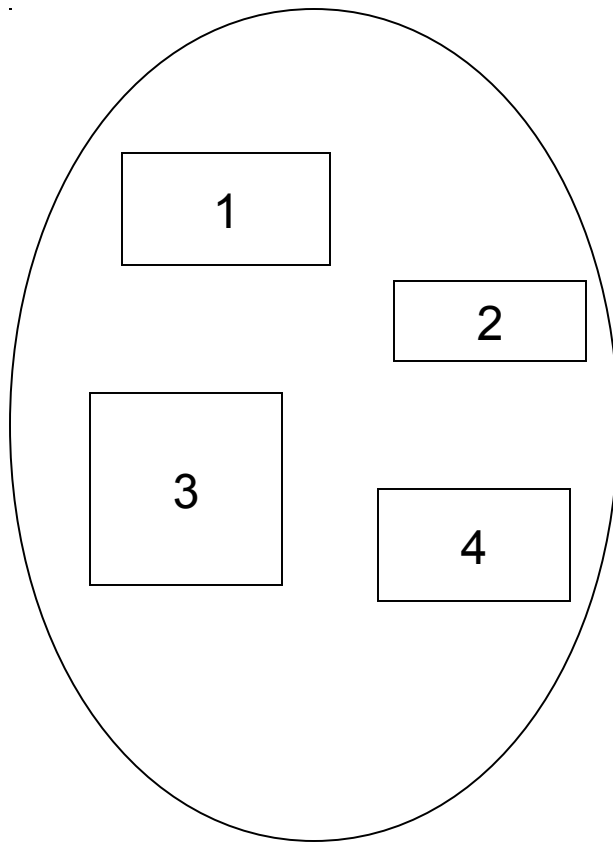
Segmentation

Memory-management scheme that supports user view of memory.

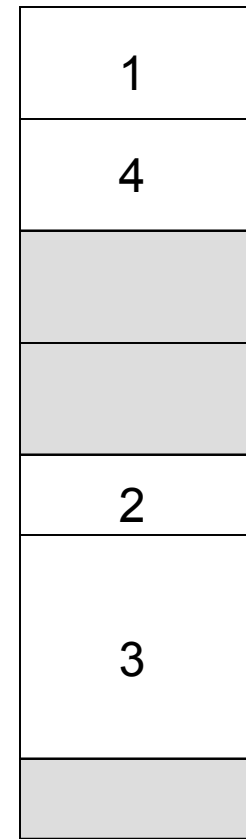
A program is a collection of segments. A segment is a logical unit such as:

- main program,
- procedure,
- function,
- local variables, global variables,
- common block,
- stack,
- symbol table, arrays

Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

Logical address consists of a two tuple: <segment-number, offset>

Segment table – maps two-dimensional physical addresses; each table entry has:

- base – contains the starting physical address where the segments reside in memory.

- limit – specifies the length of the segment.

Segment-table base register (STBR) points to the segment table's location in memory.

Segment-table length register (STLR) indicates number of segments used by a program; segment number s is legal if $s < \text{STLR}$.

Segmentation Architecture (Cont.)

Relocation.

- dynamic

- by segment table

Sharing.

- shared segments

- same segment number

Allocation.

- first fit/best fit

- external fragmentation

Segmentation Architecture (Cont.)

Protection. With each entry in segment table associate:

validation bit = 0 \Rightarrow illegal segment

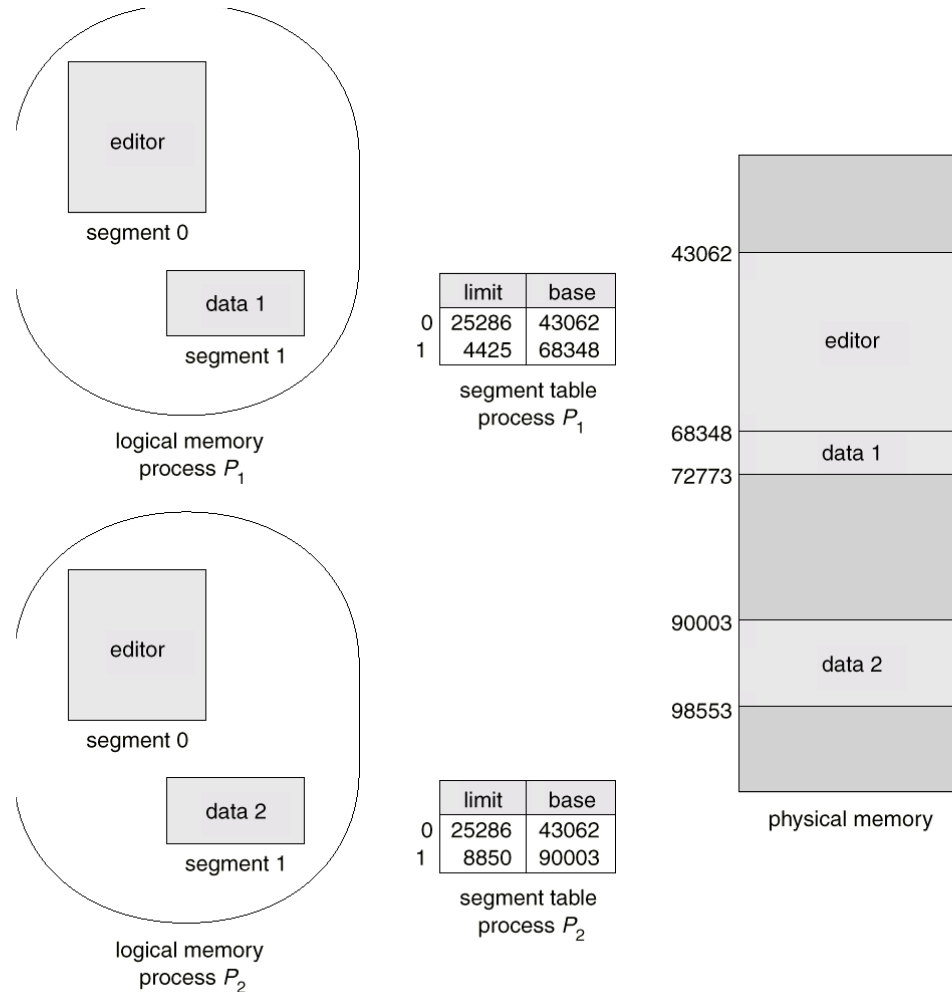
read/write/execute privileges

Protection bits associated with segments; code sharing occurs at segment level.

Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

A segmentation example is shown in the following diagram

Sharing of segments

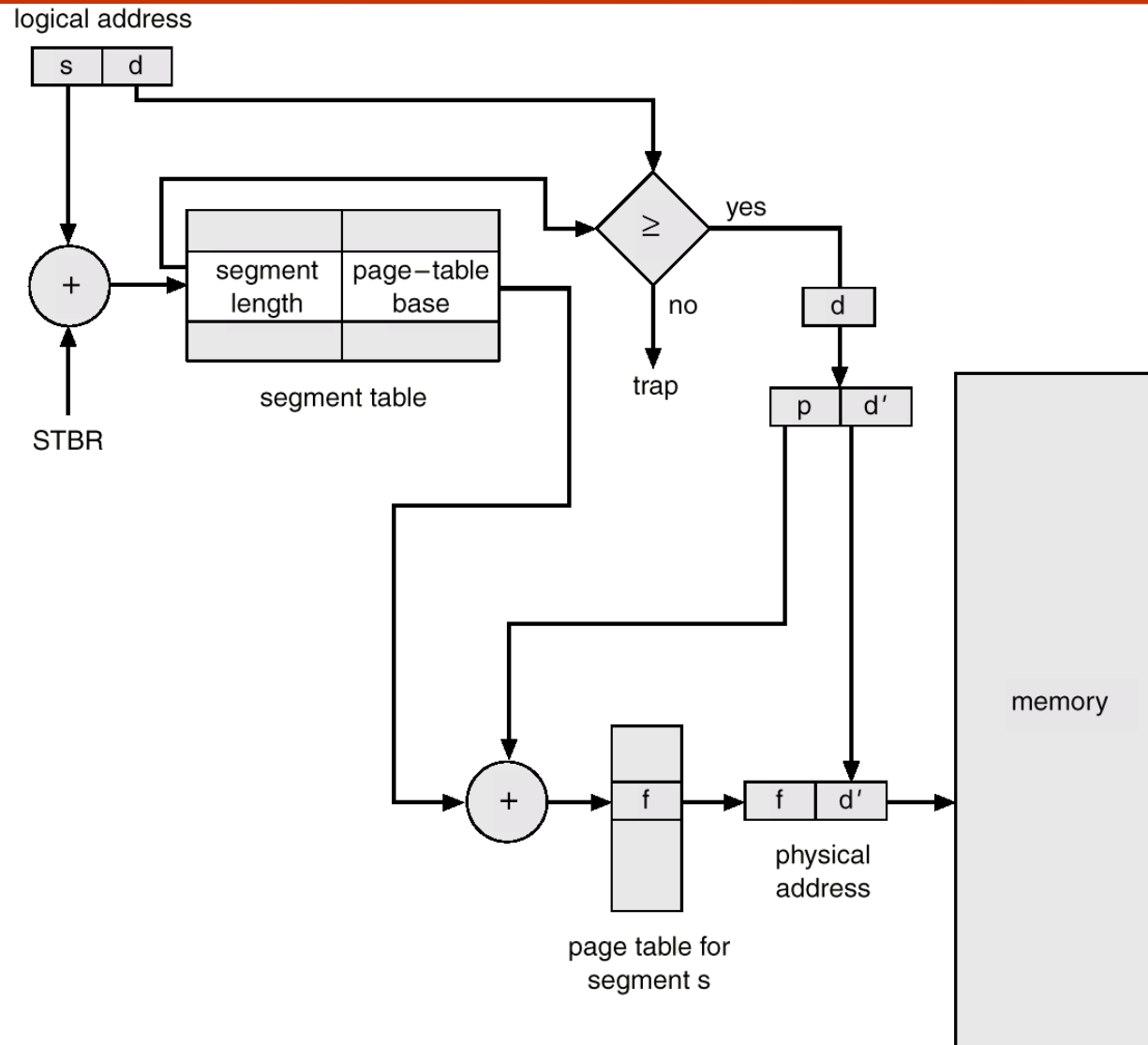


Segmentation with Paging – MULTICS

The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

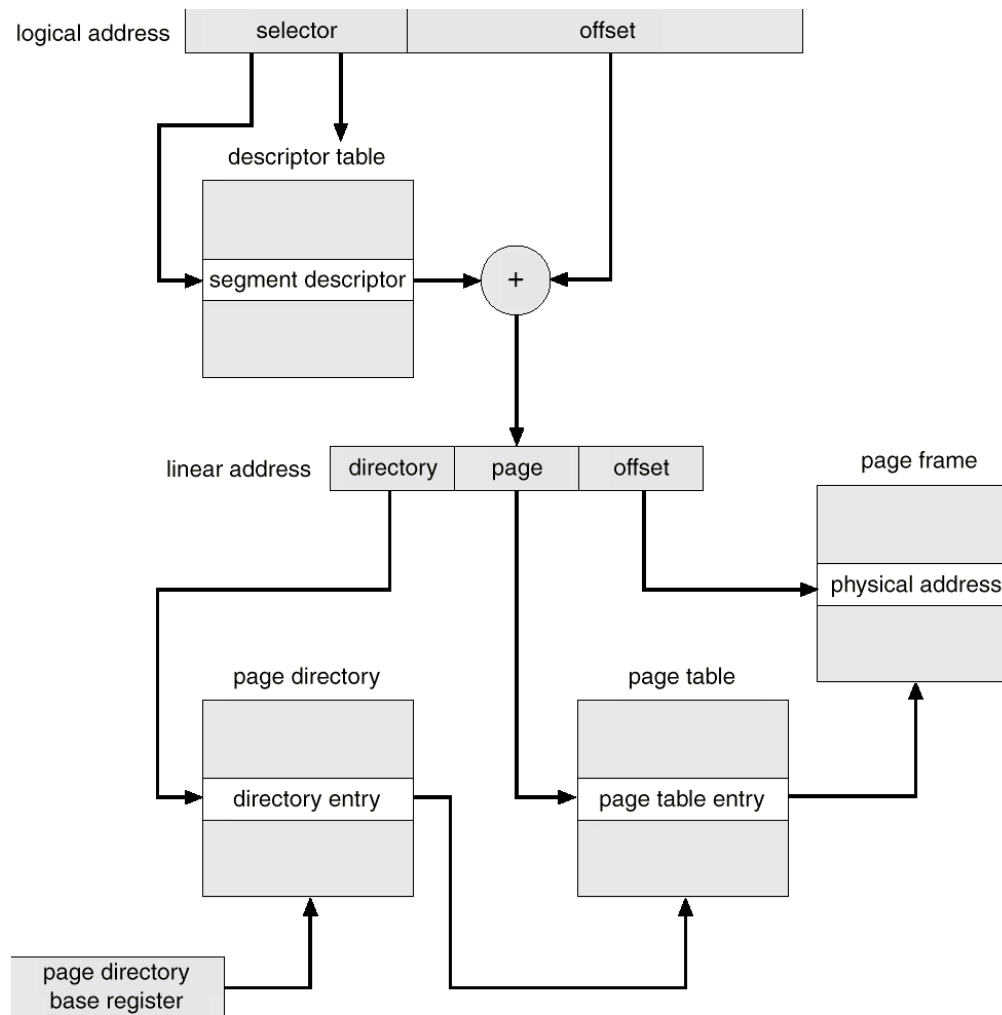
MULTICS Address Translation Scheme



Segmentation with Paging – Intel 386

As shown in the following diagram, the Intel 386 uses segmentation with two-level paging for memory management.

Intel 80386 address translation



Summary

Virtual memory is a way of introducing another level in our memory hierarchy in order to abstract away the amount of memory actually available in a particular system

This is incredibly important for ease of programming

Imagine having to explicitly check for size of physical memory and manage it in each and every one of your programs

It's also useful to prevent fragmentation in multiprogramming environments

Can be implemented using paging (sometimes segmentation or both)

Page faults are expensive so can't have too many of them

Important to implement a good page replacement policy

Have to watch out for thrashing!!