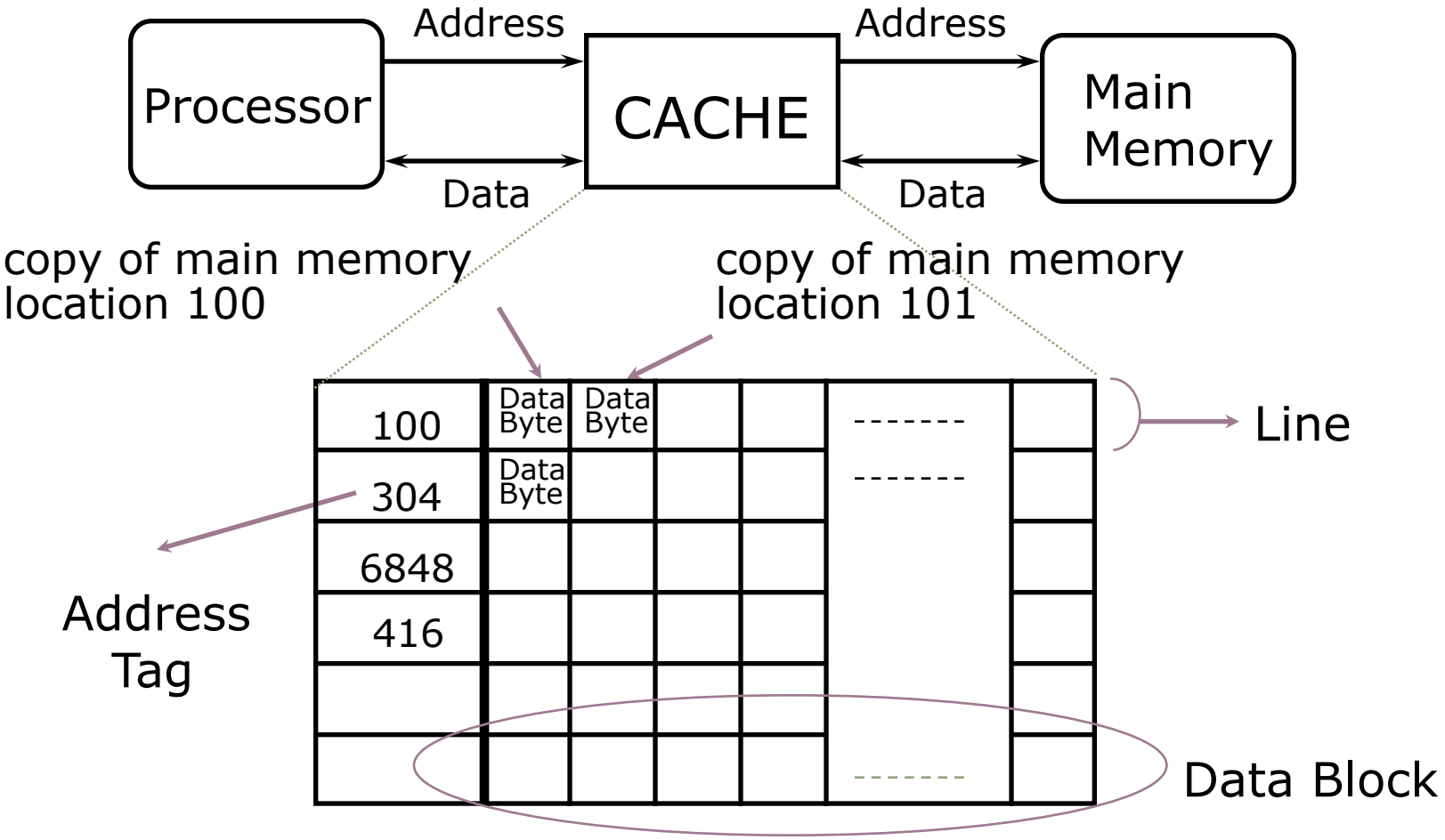
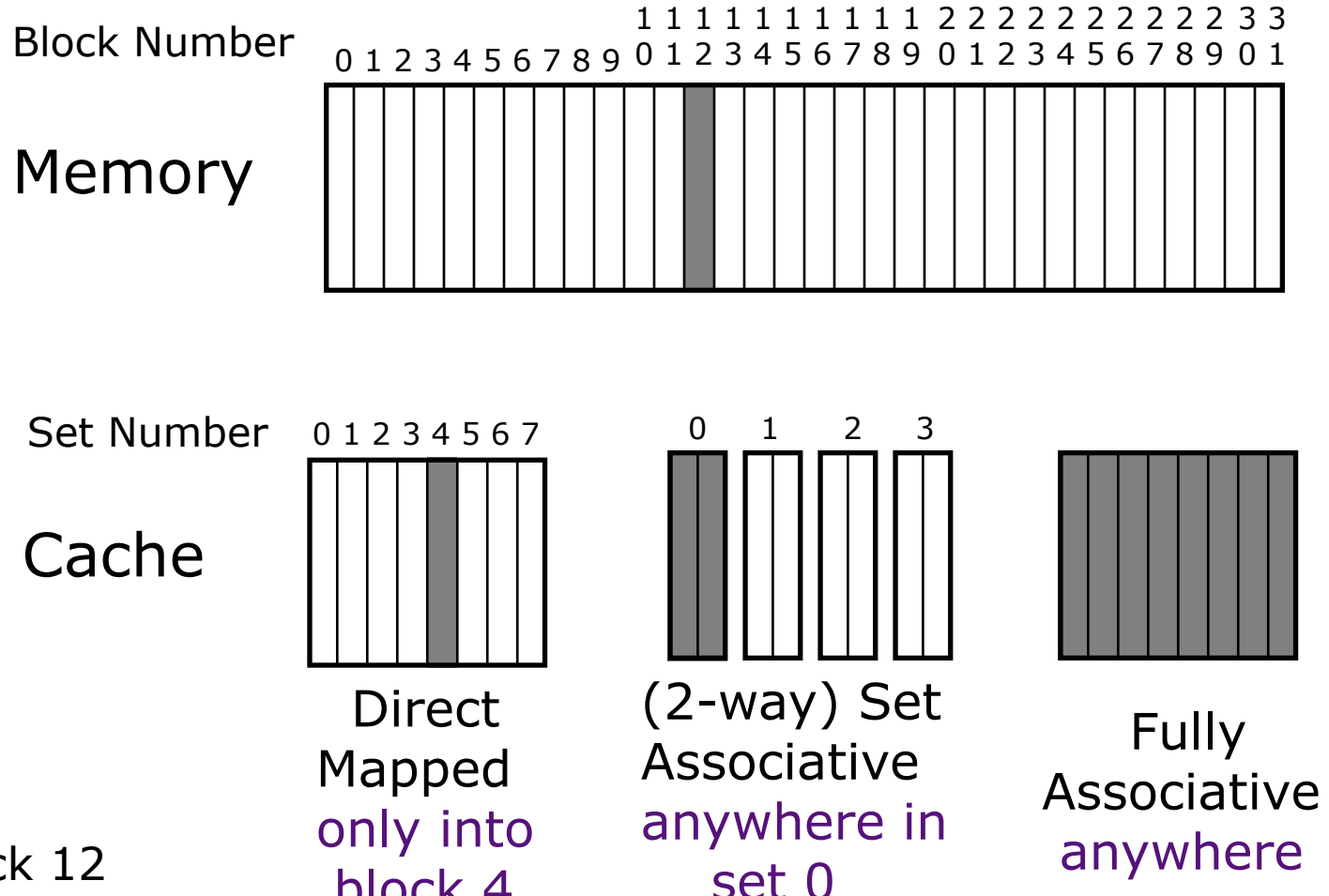
Caches (continued)

Mengjia Yan
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Recap: Inside a Cache



Recap: Placement Policy



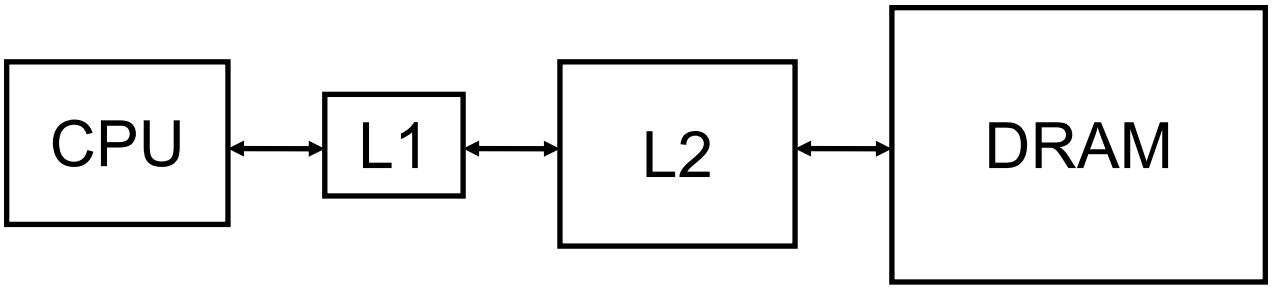
Effect of Cache Parameters on Performance

	Larger capacity cache	Higher associativity cache	Larger block size cache *
Compulsory misses	=	=	↓
Capacity misses	↓	=	↓
Conflict misses	↓	↓	?
Hit latency	↑	↑	=
Miss latency	=	=	↑↓

* Assume substantial spatial locality

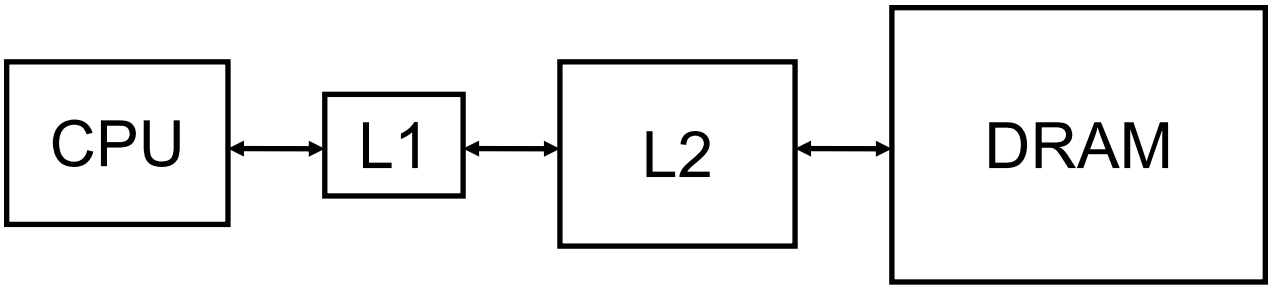
Multilevel Caches

- A memory cannot be large and fast
- Add level of cache to reduce miss penalty
  - Each level can have longer latency than level above
  - So, increase sizes of cache at each level



Multilevel Caches

- A memory cannot be large and fast
- Add level of cache to reduce miss penalty
  - Each level can have longer latency than level above
  - So, increase sizes of cache at each level



Metrics:

Local miss rate = misses in cache/ accesses to cache

Global miss rate = misses in cache / CPU memory accesses

Misses per instruction (MPI) = misses in cache / number of instructions

Block-level Optimizations

- Tags are too large, i.e., too much overhead
  - Simple solution: Larger blocks, but miss penalty could be large.

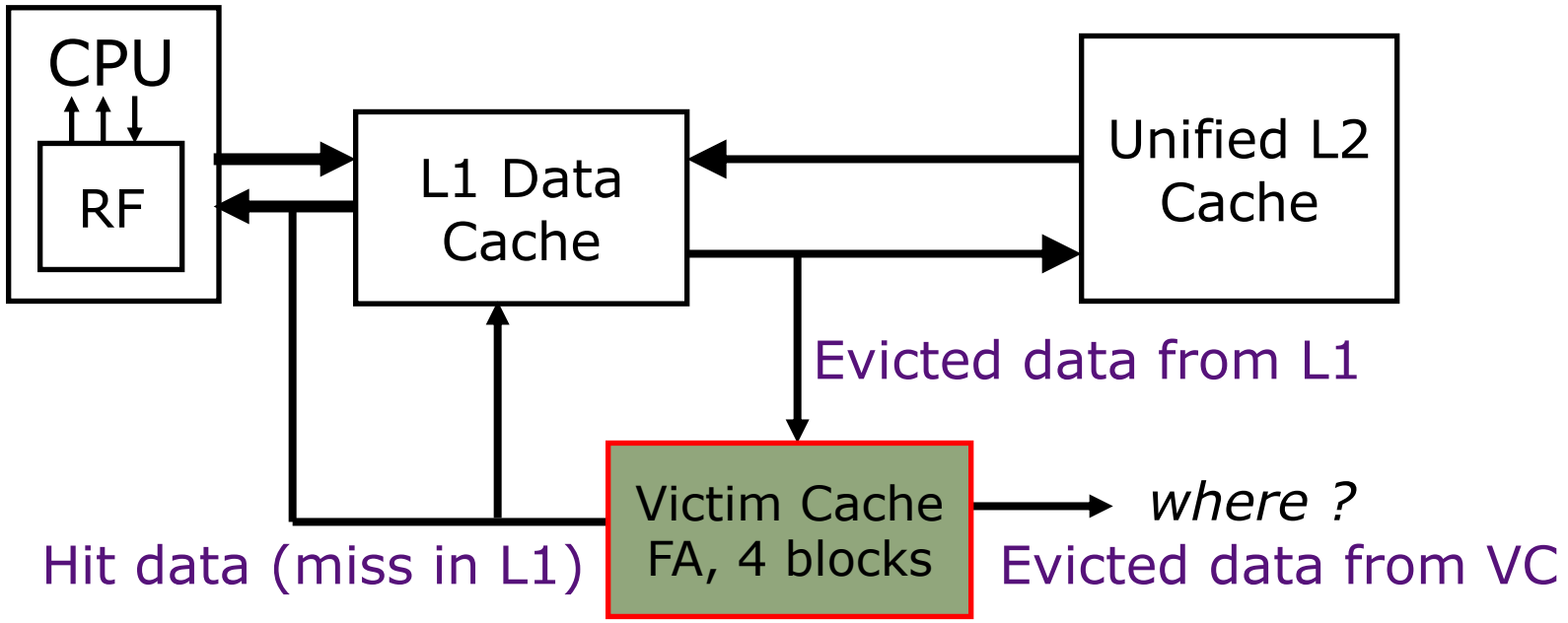
Block-level Optimizations

- Tags are too large, i.e., too much overhead
  - Simple solution: Larger blocks, but miss penalty could be large.
- Sub-block placement (aka sector cache)
  - A valid bit added to units smaller than the full block, called sub-blocks
  - Only read a sub-block on a miss
  - *If a tag matches, is the sub-block in the cache?*

100
300
204

1		***1***		***1***		***1***	
1		***1***		***0***		***0***	
0		***1***		***0***		***1***	

Victim Caches (HP 7200)



Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines

- First look up in direct mapped cache
- If miss, look in victim cache
- If hit in victim cache, swap hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim->?

Fast hit time of direct mapped but with reduced conflict misses
-> Nowadays, more general, L4 in Intel Haswell, L3 in IBM Power5

Inclusion Policy

- Inclusive multilevel cache:
  - Inner cache holds copies of data in outer cache
  - On miss, line inserted in inner and outer cache; replacement in outer cache invalidates line in inner cache
  - External accesses need only check outer cache
  - Commonly used (e.g., Intel CPUs up to Broadwell)
- Non-inclusive multilevel caches:
  - Inner cache may hold data not in outer cache
  - Replacement in outer cache doesn't invalidate line in inner cache
  - Used in Intel Skylake, ARM
- Exclusive multilevel caches:
  - Inner cache and outer cache hold different data
  - Swap lines between inner/outer caches on miss
  - Used in AMD processors

Inclusion Policy

- Inclusive multilevel cache:
  - Inner cache holds copies of data in outer cache
  - On miss, line inserted in inner and outer cache; replacement in outer cache invalidates line in inner cache
  - External accesses need only check outer cache
  - Commonly used (e.g., Intel CPUs up to Broadwell)
- Non-inclusive multilevel caches:
  - Inner cache may hold data not in outer cache
  - Replacement in outer cache doesn't invalidate line in inner cache
  - Used in Intel Skylake, ARM
- Exclusive multilevel caches:
  - Inner cache and outer cache hold different data
  - Swap lines between inner/outer caches on miss
  - Used in AMD processors

Why choose one type or the other?

Replacement Policy

Which block from a set should be evicted?

Replacement Policy

Which block from a set should be evicted?

- Random

Replacement Policy

Which block from a set should be evicted?

- Random

Replacement Policy

Which block from a set should be evicted?

- Random
- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree was often used for 4-8 way

Replacement Policy

Which block from a set should be evicted?

- Random
- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree was often used for 4-8 way
- First In, First Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches

Replacement Policy

Which block from a set should be evicted?

- Random
- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree was often used for 4-8 way
- First In, First Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches
- Not Least Recently Used (NLRU)
  - FIFO with exception for most recently used block or blocks

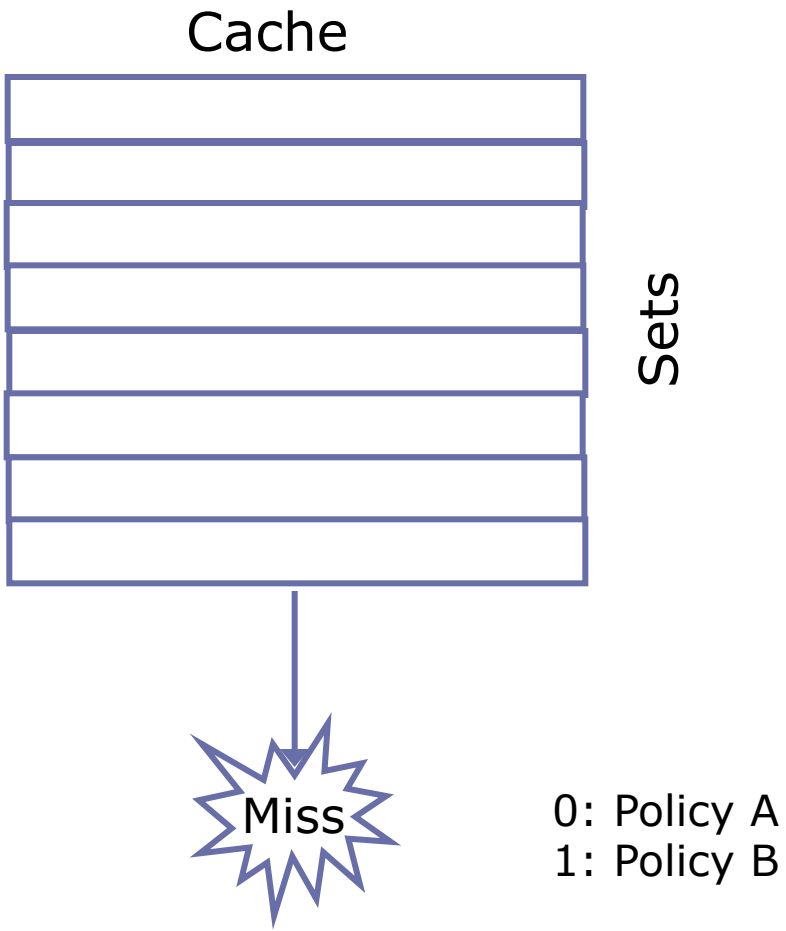
Replacement Policy

Which block from a set should be evicted?

- Random
- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree was often used for 4-8 way
- First In, First Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches
- Not Least Recently Used (NLRU)
  - FIFO with exception for most recently used block or blocks
- One-bit LRU
  - Each way represented by a bit. Set on use, replace first unused.

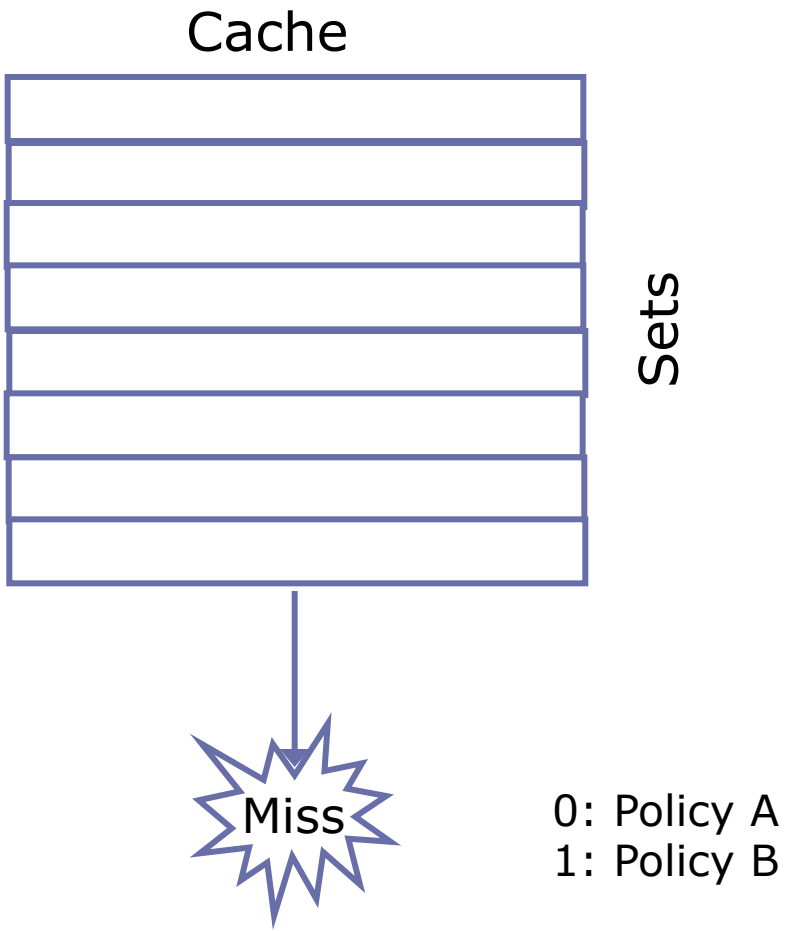
Multiple replacement policies

Use the best replacement policy for a program



Multiple replacement policies

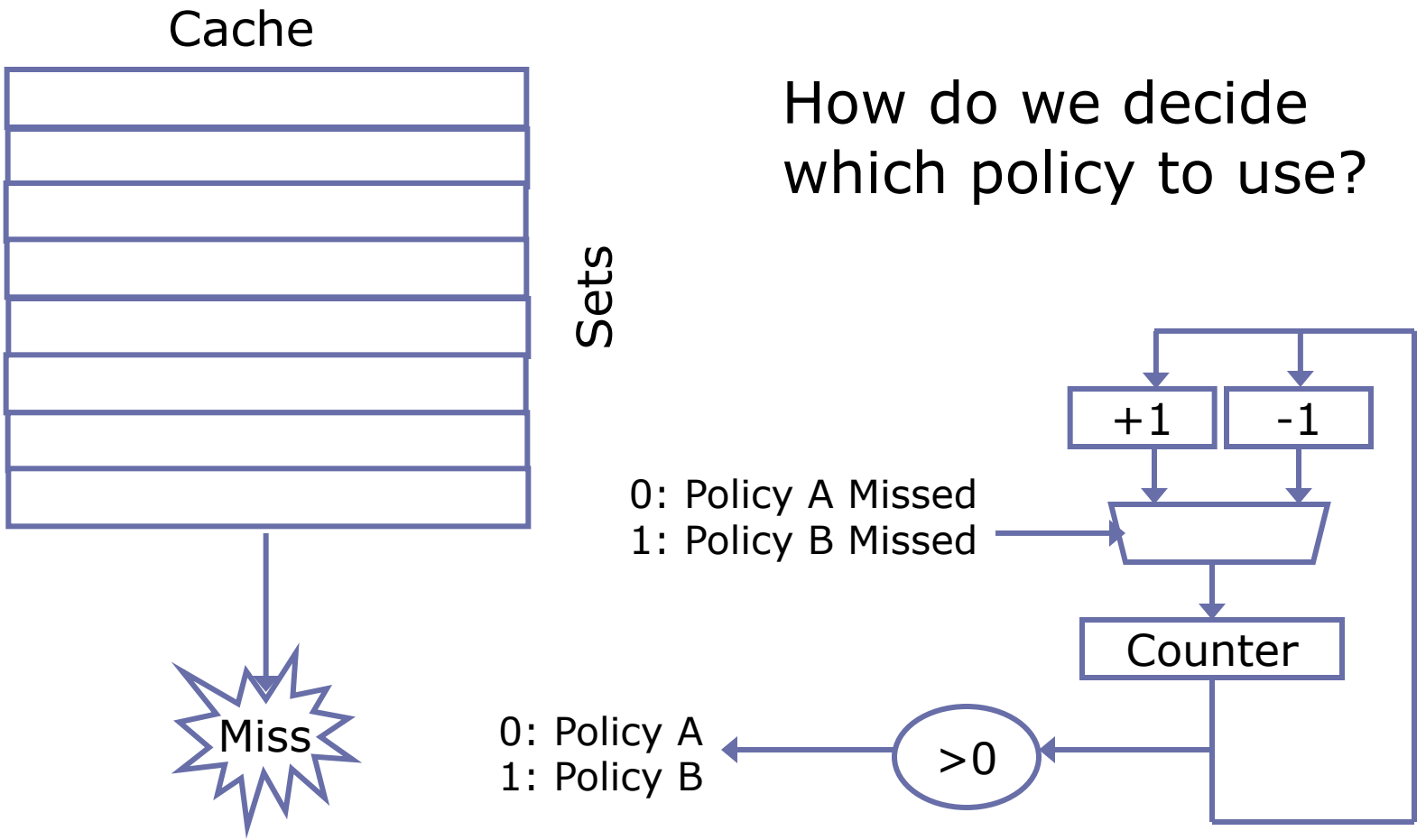
Use the best replacement policy for a program



How do we decide which policy to use?

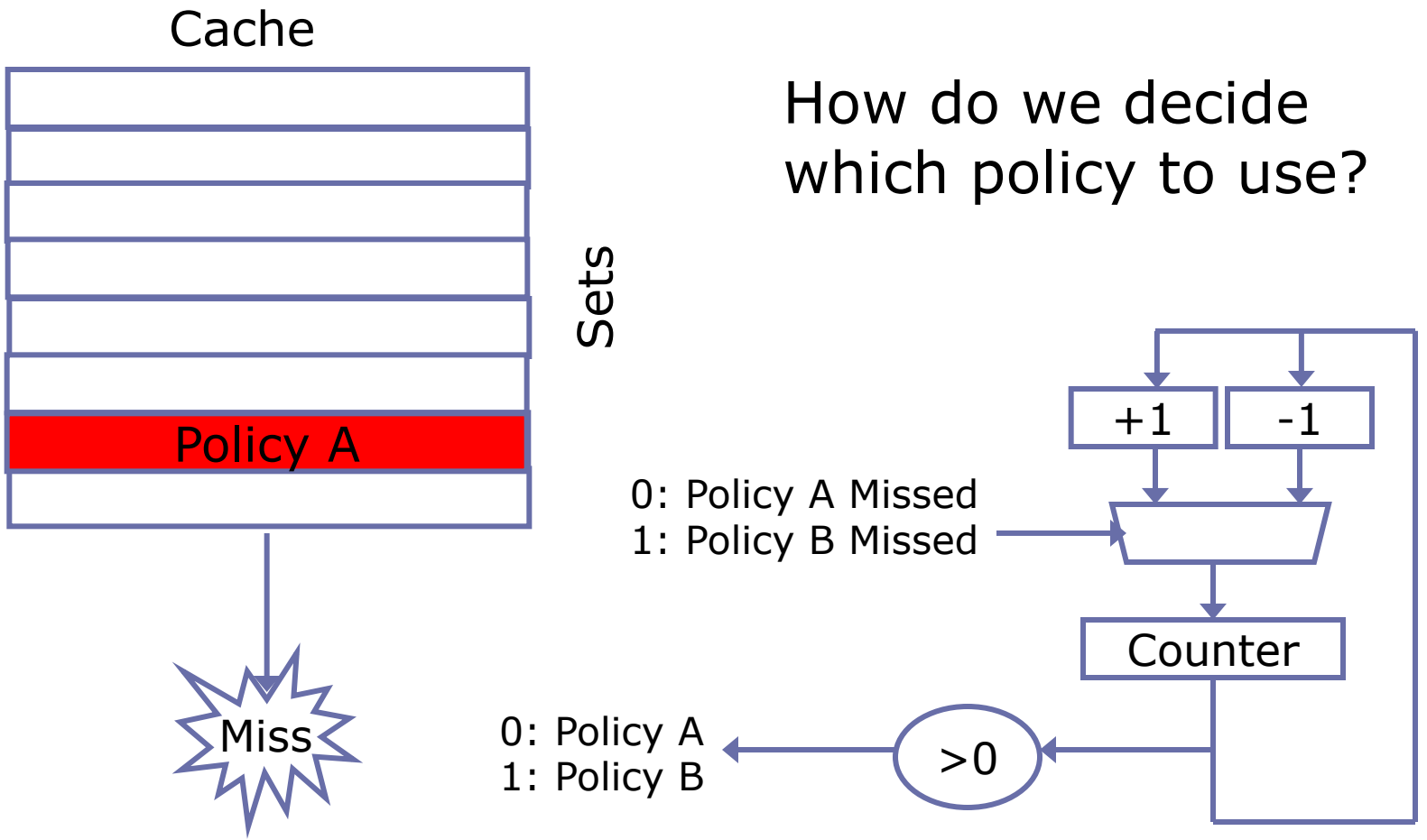
Multiple replacement policies

Use the best replacement policy for a program



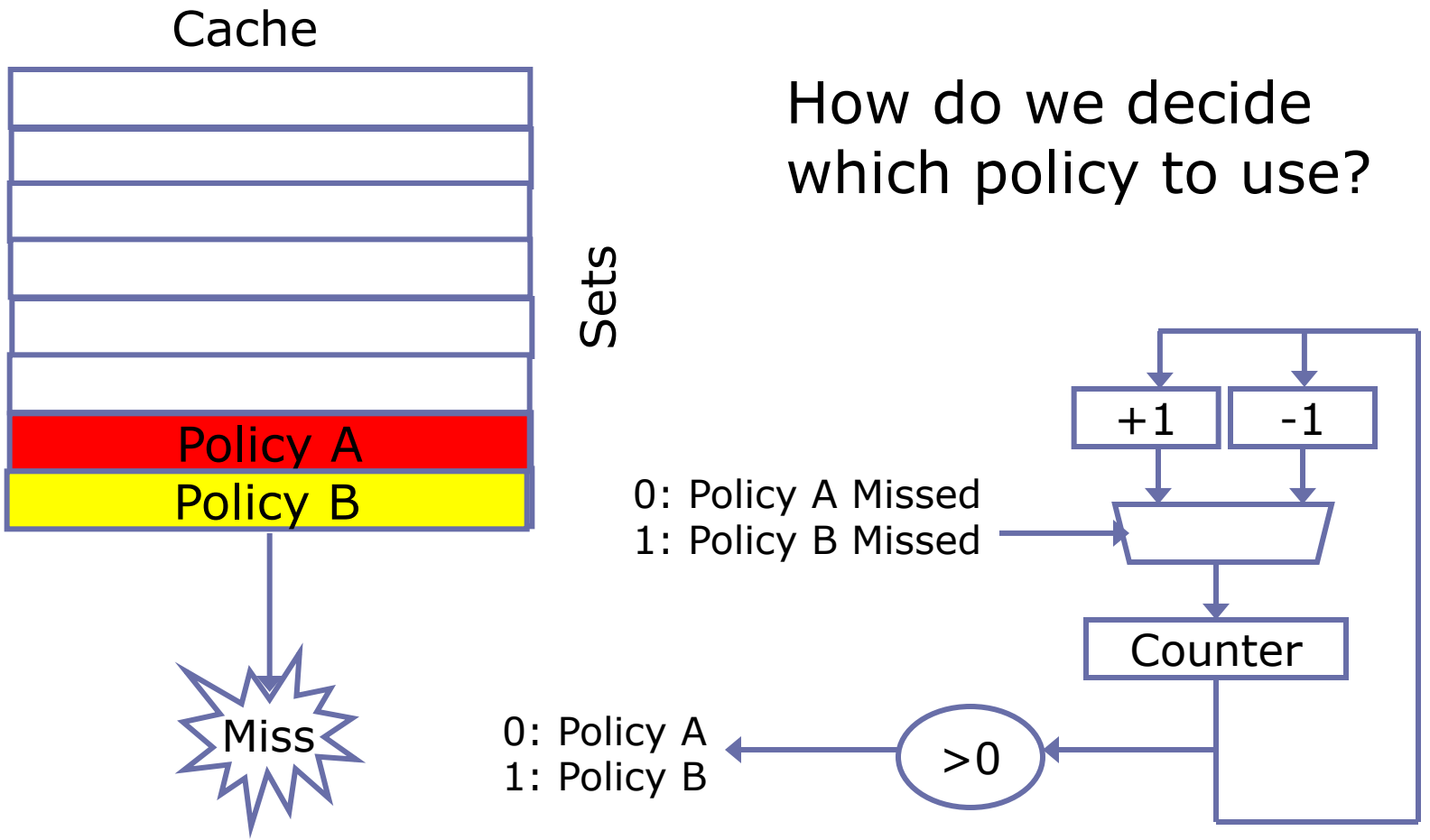
Multiple replacement policies

Use the best replacement policy for a program

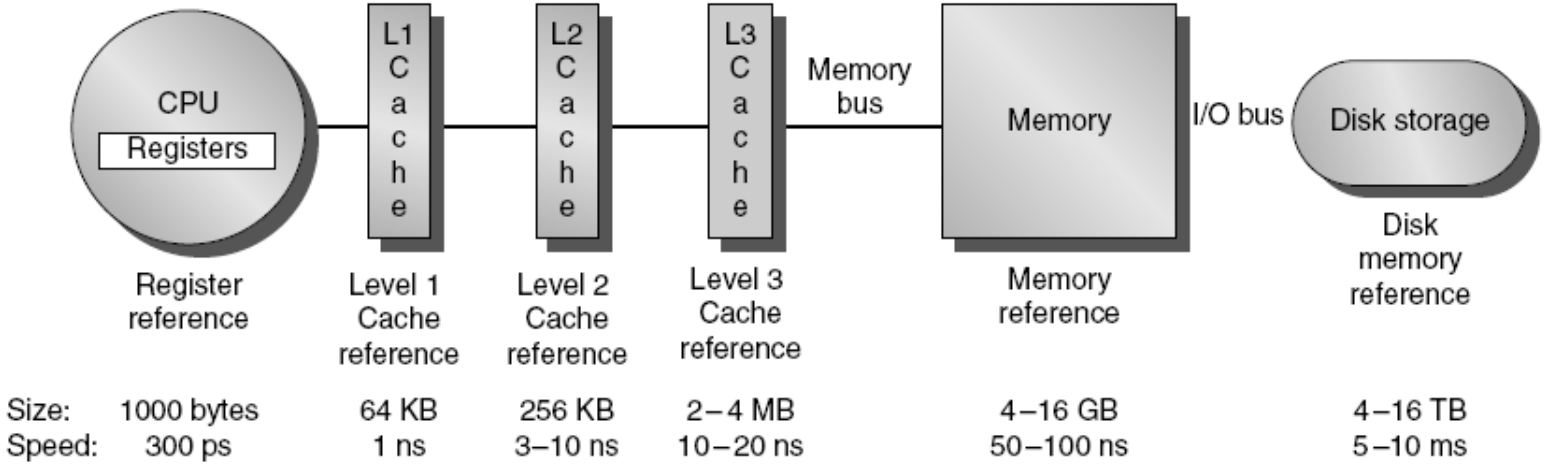


Multiple replacement policies

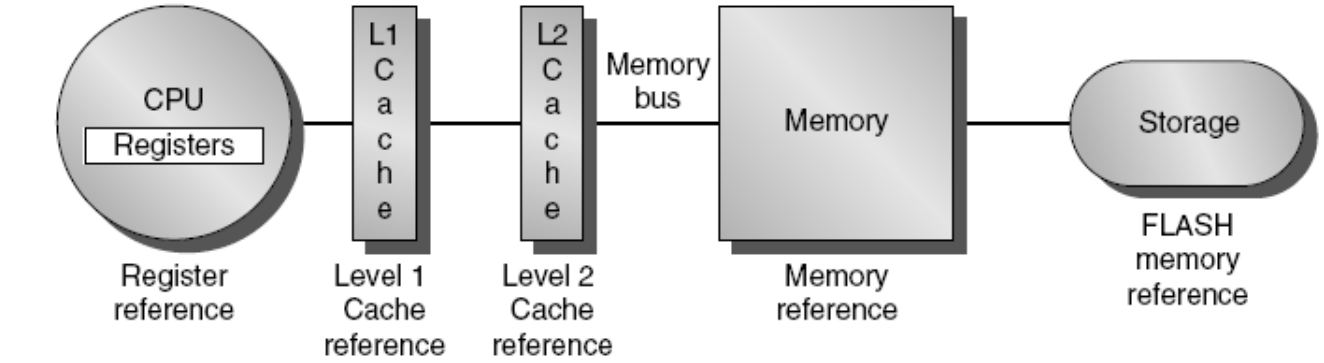
Use the best replacement policy for a program



Typical memory hierarchies



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

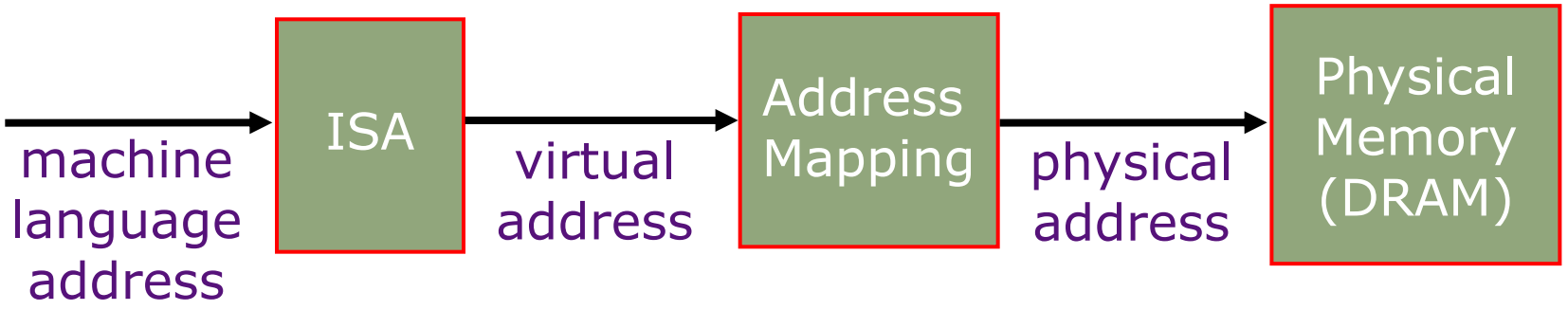
Memory Management: *From Absolute Addresses to Demand Paging*

Mengjia Yan
Computer Science and Artificial Intelligence Laboratory
M.I.T.

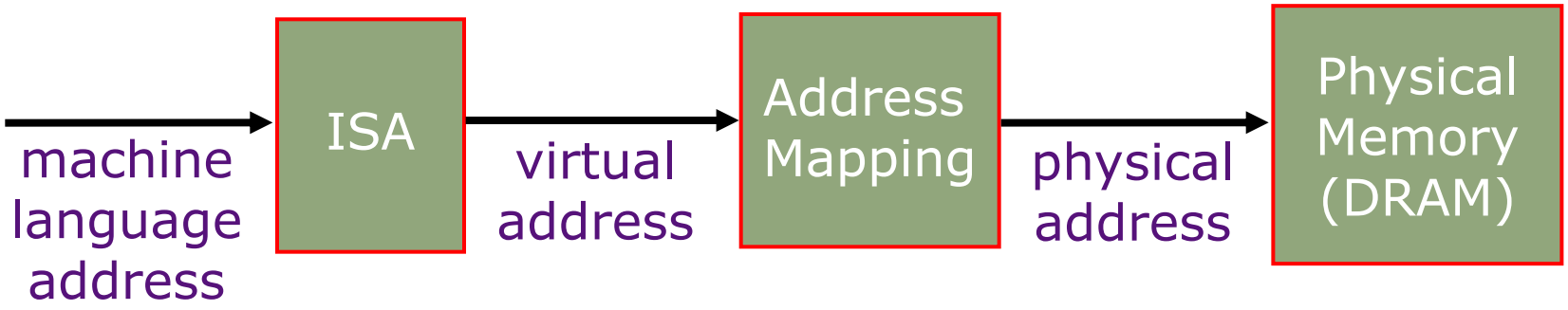
Memory Management

- The Fifties
  - Absolute Addresses
  - Dynamic address translation
- The Sixties
  - Atlas and Demand Paging
  - Paged memory systems and TLBs
- Modern Virtual Memory Systems

Names for Memory Locations



Names for Memory Locations



- Machine language address
  - as specified in machine code
- Virtual address
  - ISA specifies translation of machine code address into virtual address of program variable (sometimes called *effective* address)
- Physical address
  - Operating system specifies mapping of virtual address into name for a physical memory location

Absolute Addresses

EDSAC, early 50's

virtual address = physical memory address

Absolute Addresses

EDSAC, early 50's

virtual address = physical memory address

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory

Absolute Addresses

EDSAC, early 50's

virtual address = physical memory address

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But* it was more convenient for programmers to write location-independent subroutines

Absolute Addresses

EDSAC, early 50's

virtual address = physical memory address

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But* it was more convenient for programmers to write location-independent subroutines

How could location independence be achieved?

Absolute Addresses

EDSAC, early 50's

virtual address = physical memory address

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But* it was more convenient for programmers to write location-independent subroutines

How could location independence be achieved?

Linker and/or loader modify addresses of subroutines and callers when building a program memory image

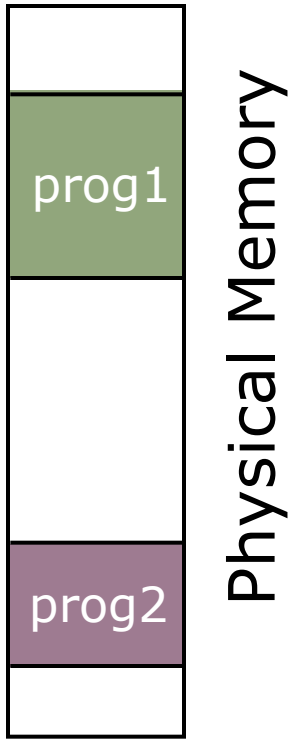
Multiprogramming

Motivation

In the early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of 2 or more programs were overlapped. *How?*

⇒ *multiprogramming*



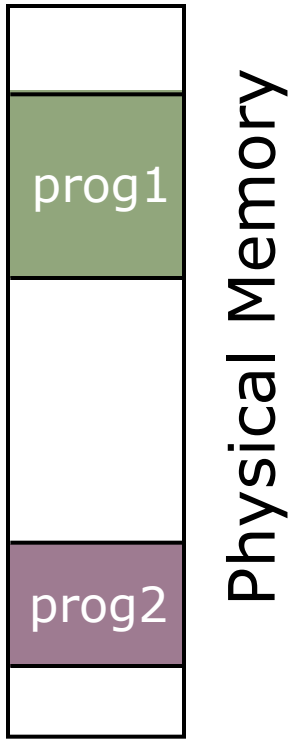
Multiprogramming

Motivation

In the early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of 2 or more programs were overlapped. *How?*

⇒ *multiprogramming*



Location-independent programs

Programming and storage management ease

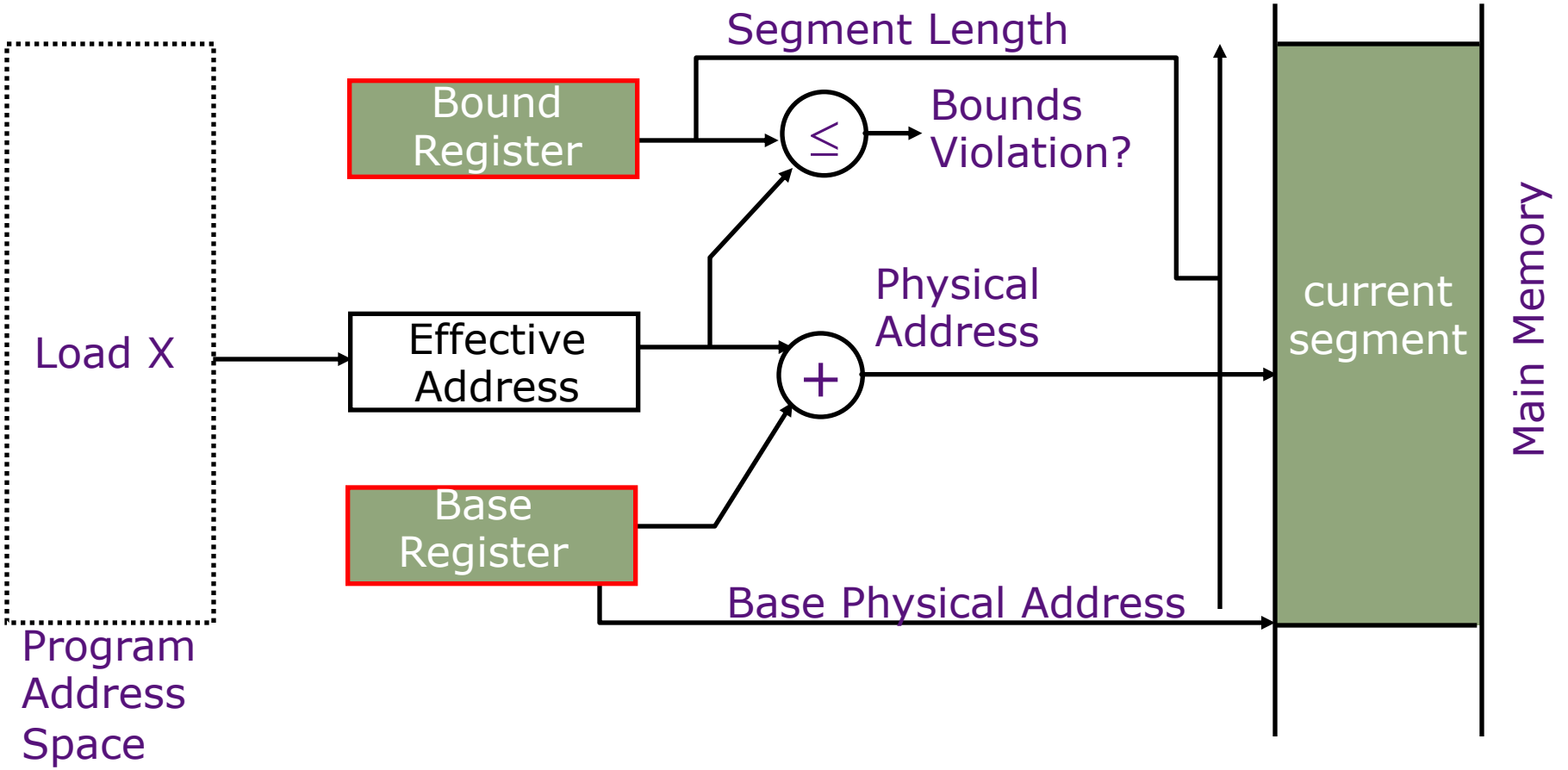
⇒ need for a *base register*

Protection

Independent programs should not affect each other inadvertently

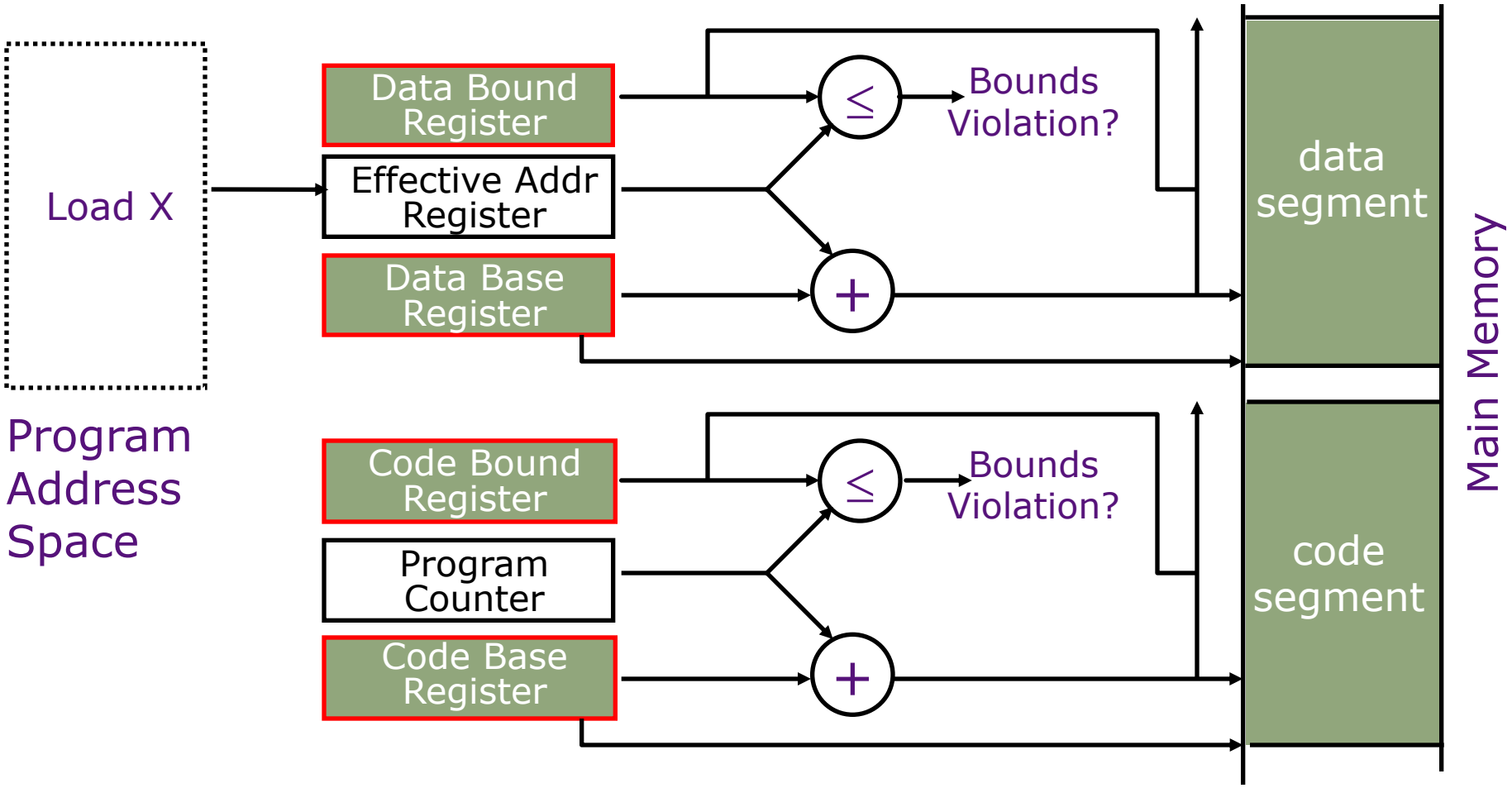
⇒ need for a *bound register*

Simple Base and Bound Translation



Base and bounds registers are visible/accessible only when processor is running in *supervisor mode*

Separate Areas for Code and Data

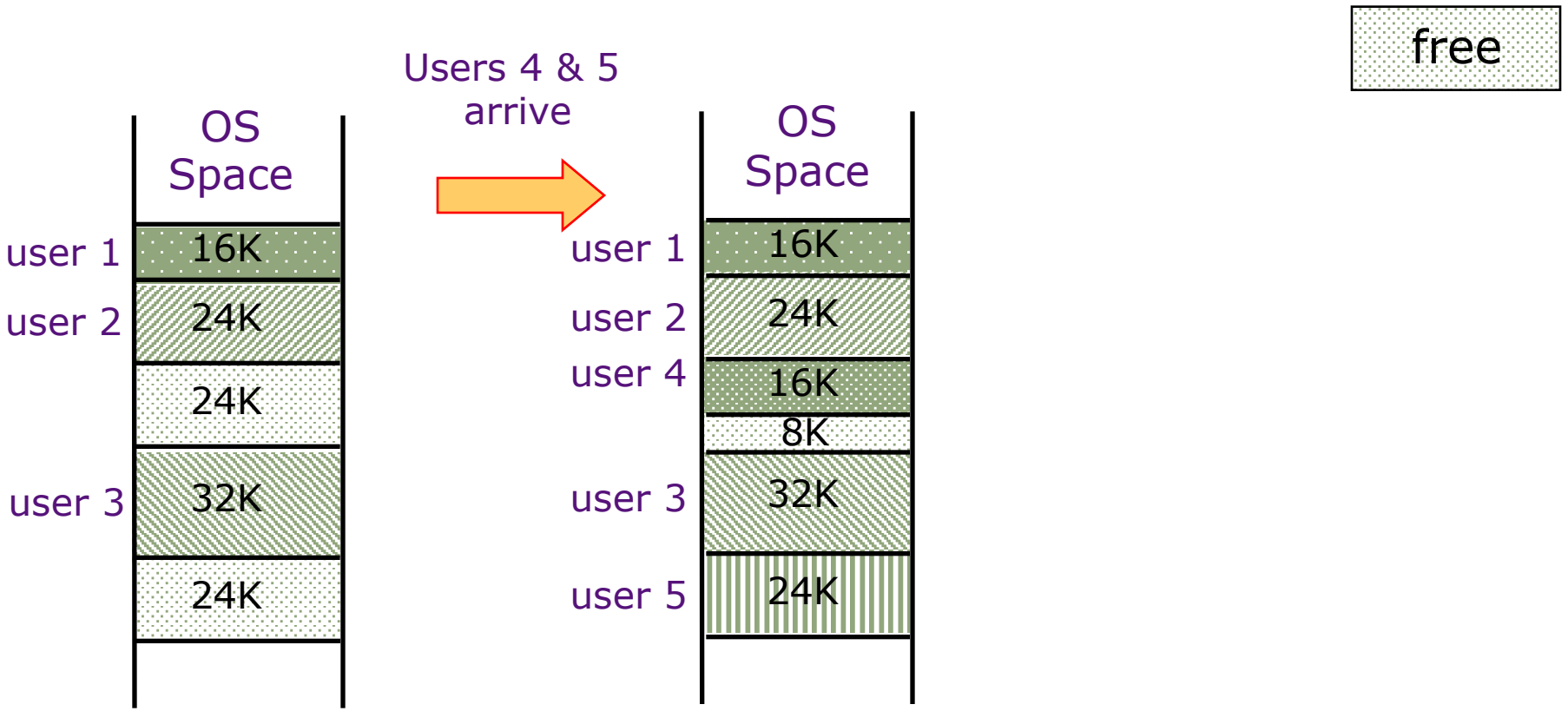


What is an advantage of this separation?

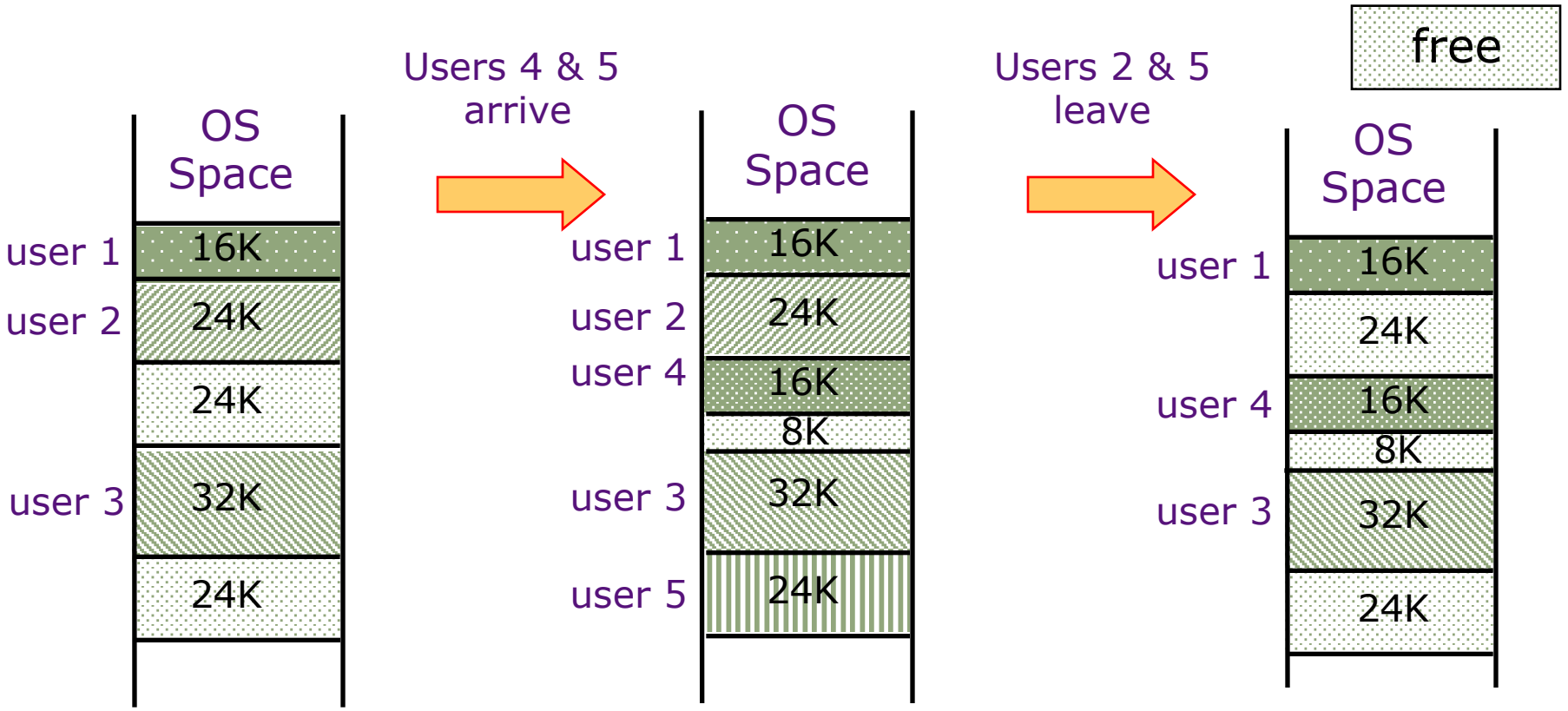
(Scheme used on all Cray vector supercomputers prior to X1, 2002)

Memory Fragmentation

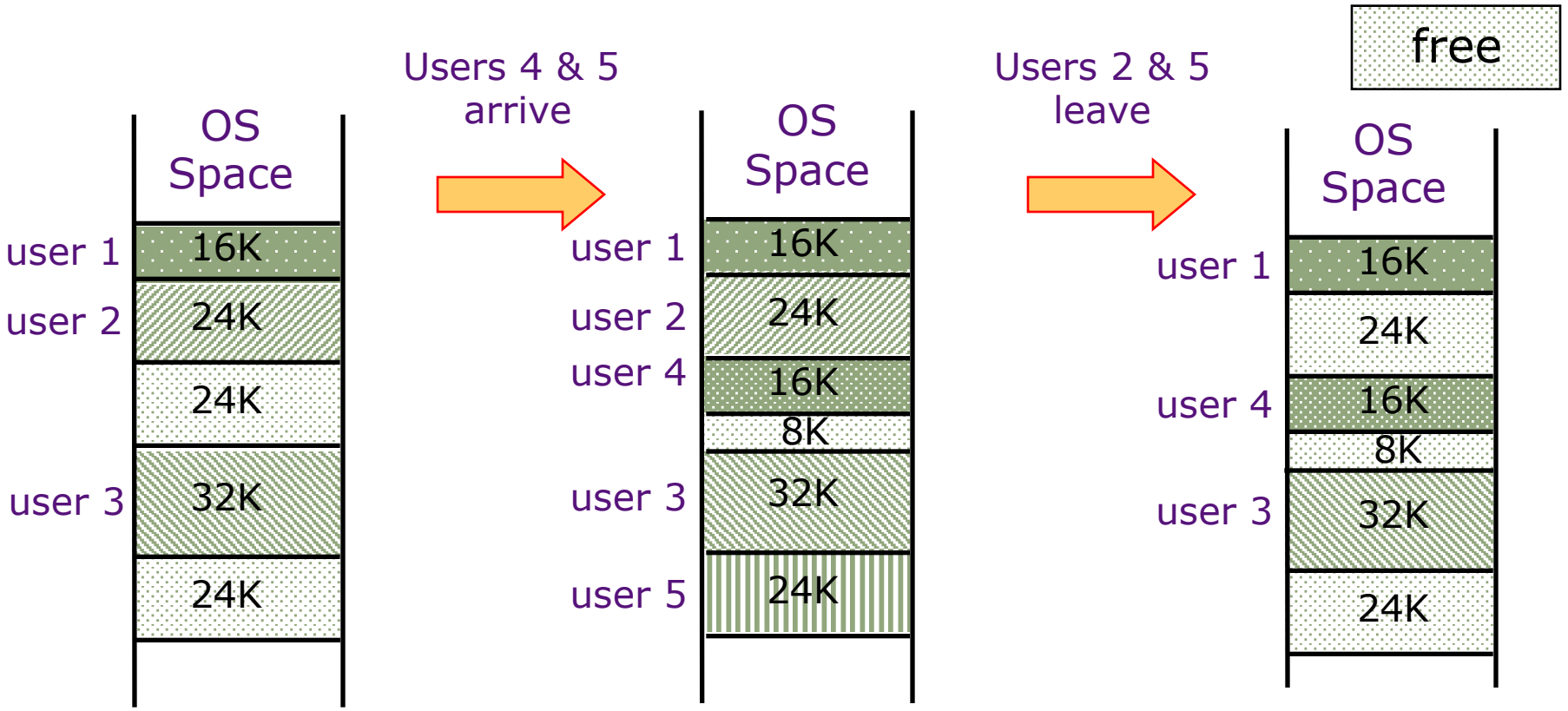
Memory Fragmentation



Memory Fragmentation

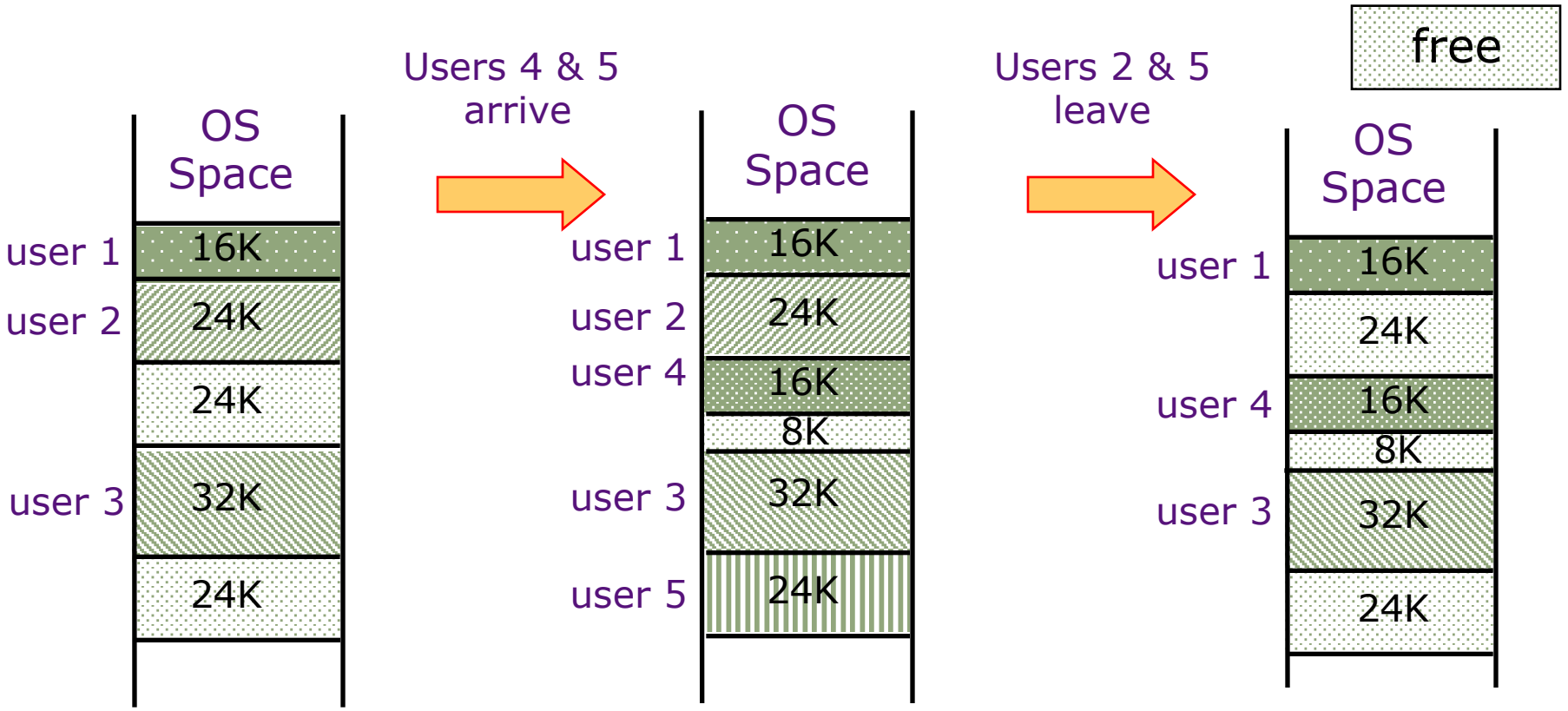


Memory Fragmentation



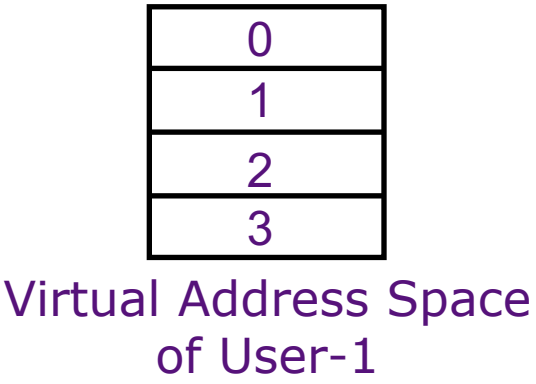
As users come and go, the storage is "fragmented".

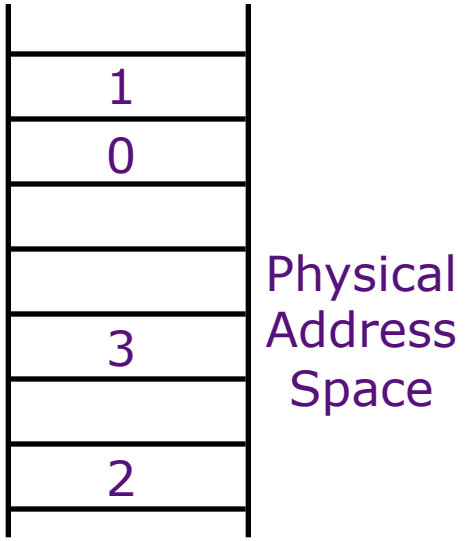
Memory Fragmentation



As users come and go, the storage is "fragmented". Therefore, at some stage programs have to be moved around to compact the storage.

Paged Memory Systems

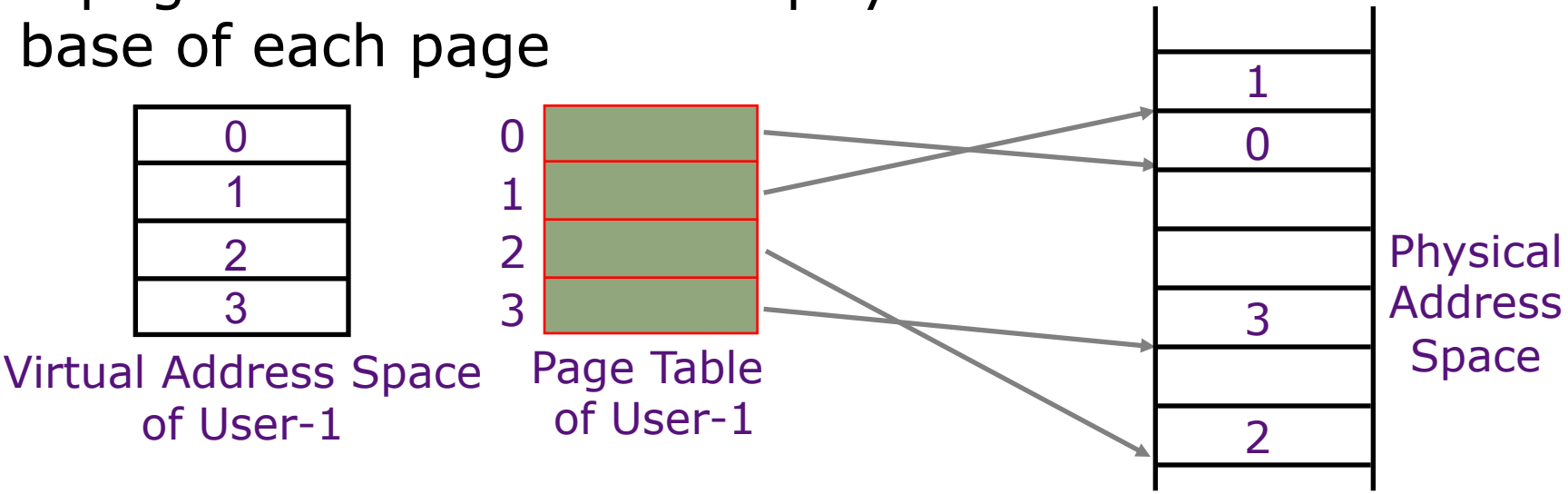




Paged Memory Systems

- Processor-generated address can be interpreted as a pair <page number, offset>

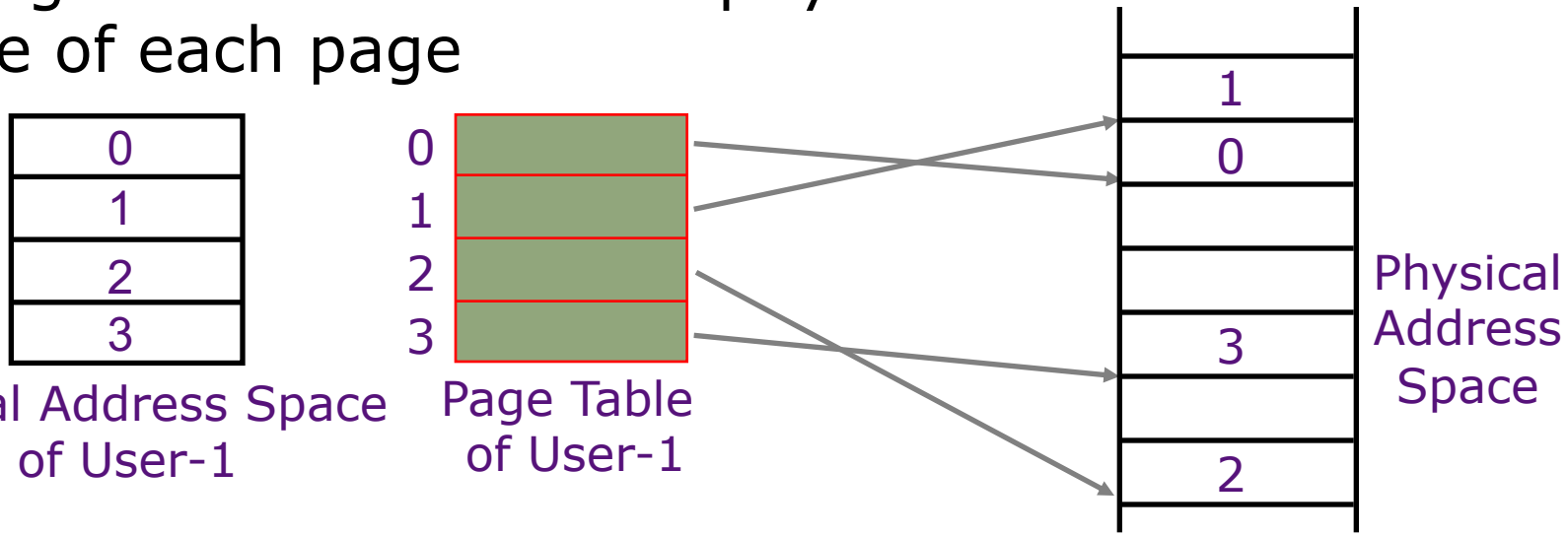
- A page table contains the physical address of the base of each page



Paged Memory Systems

- Processor-generated address can be interpreted as a pair <page number, offset>

- A page table contains the physical address of the base of each page



Page tables make it possible to store the pages of a program ***non-contiguously****.*

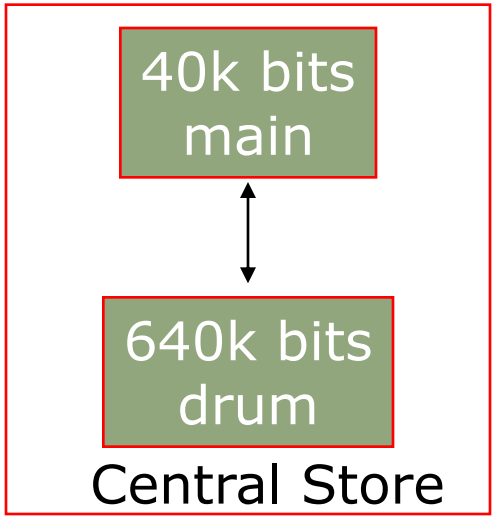
A Problem in Early Sixties

- There were many applications whose data could not fit in the main memory, e.g., payroll
  - *Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory*

- Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

tricky programming!

Manual Overlays

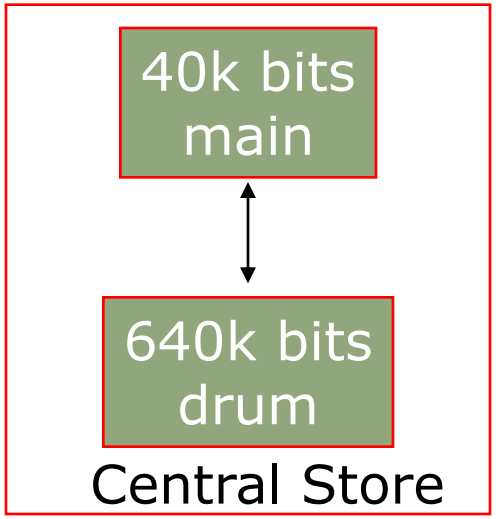


Ferranti Mercury
1956

Manual Overlays

- Assume an instruction can address all the storage on the drum
- *Method 1:* programmer keeps track of addresses in the main memory and initiates an I/O transfer when required
- *Method 2:* automatic initiation of I/O transfers by software address translation

Brooker's interpretive coding, 1960

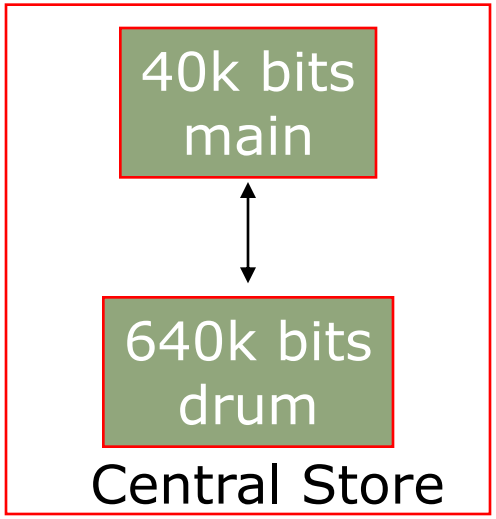


Ferranti Mercury
1956

Manual Overlays

- Assume an instruction can address all the storage on the drum
- *Method 1:* programmer keeps track of addresses in the main memory and initiates an I/O transfer when required
- *Method 2:* automatic initiation of I/O transfers by software address translation

Brooker's interpretive coding, 1960



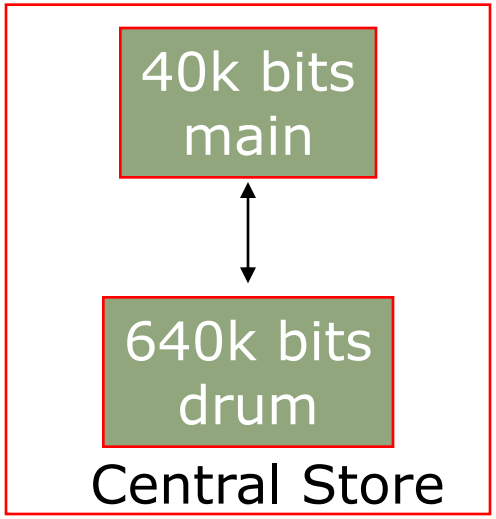
Ferranti Mercury
1956

Problems?

Manual Overlays

- Assume an instruction can address all the storage on the drum
- *Method 1:* programmer keeps track of addresses in the main memory and initiates an I/O transfer when required
- *Method 2:* automatic initiation of I/O transfers by software address translation

Brooker's interpretive coding, 1960



Ferranti Mercury
1956

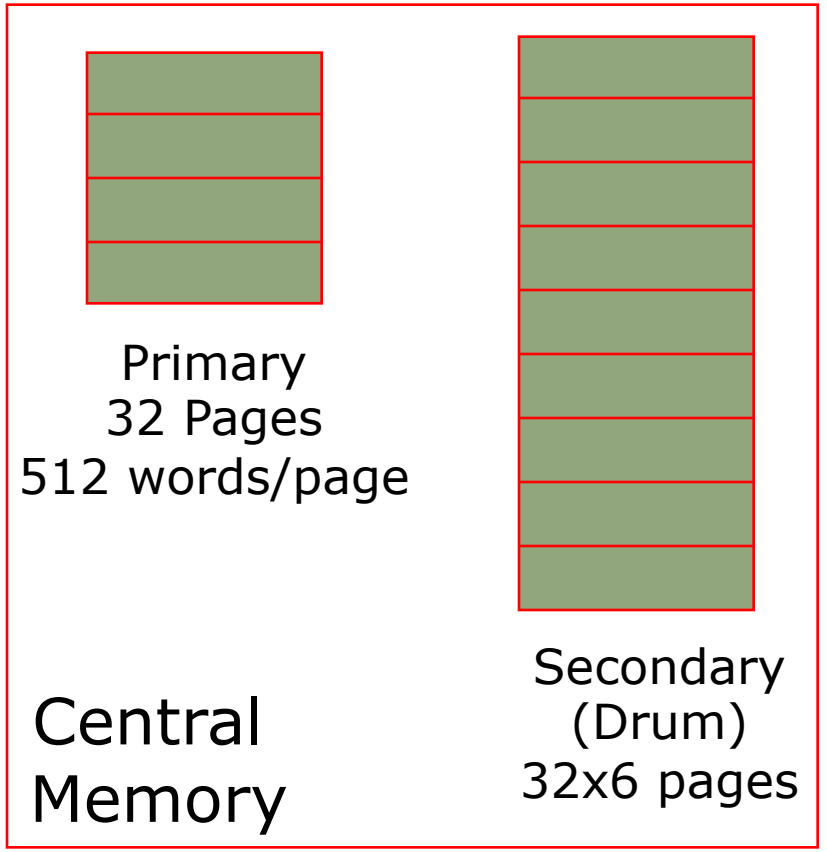
Problems?

Method1: Difficult, error prone
Method2: Inefficient

Demand Paging in Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

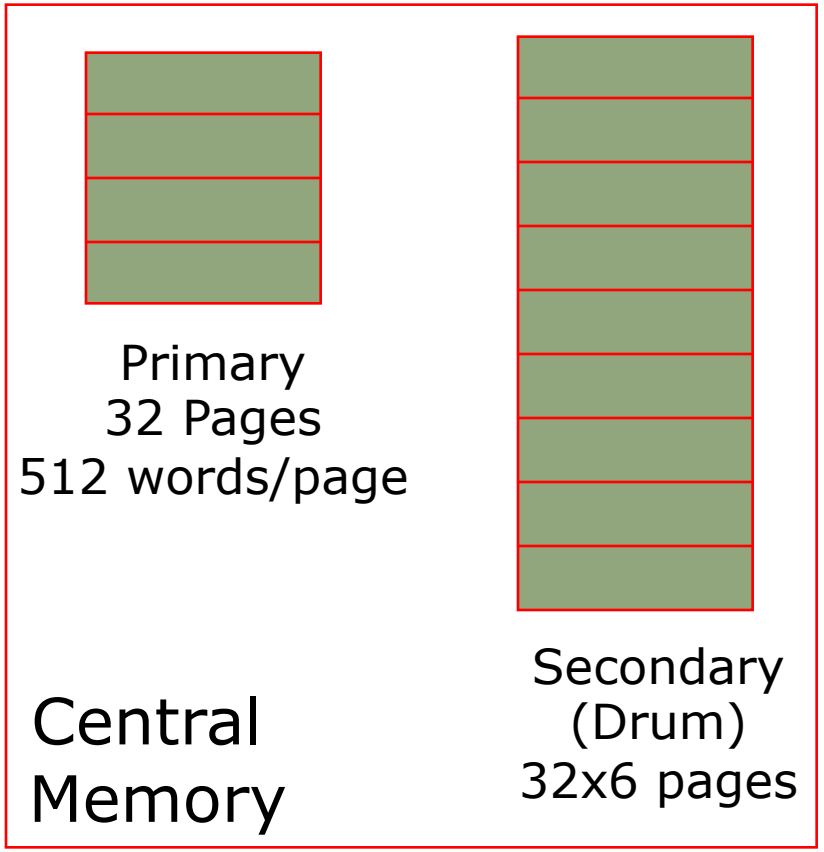
Tom Kilburn



Demand Paging in Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

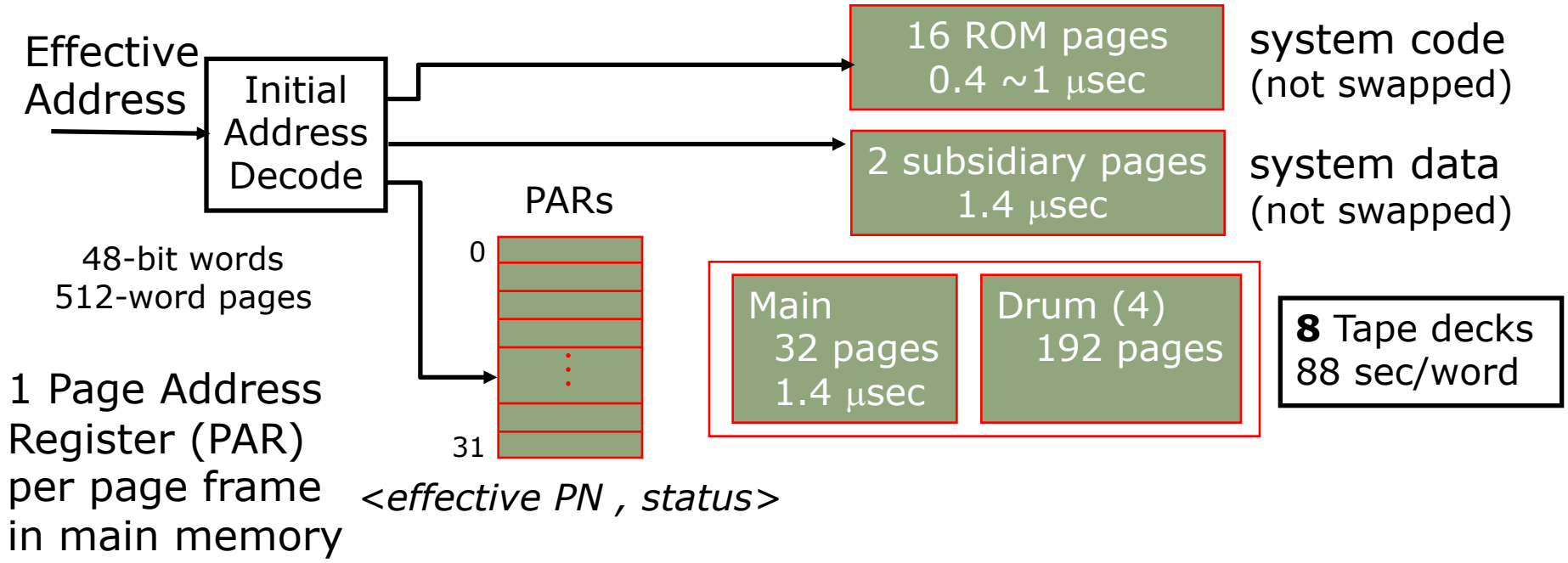
Tom Kilburn



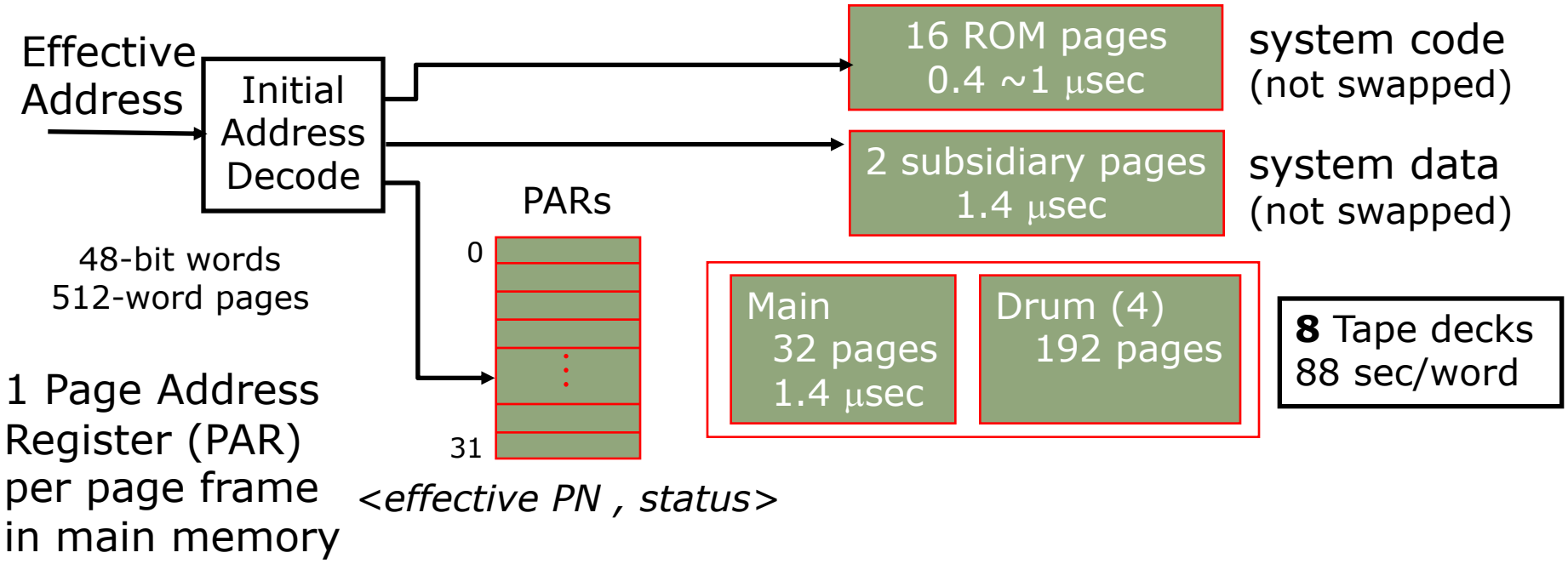
Primary memory as a *cache* for secondary memory

User sees the storage size of the secondary storage, since data transfer happens automatically

Hardware Organization of Atlas



Hardware Organization of Atlas



Compare the effective page address against all 32 PARs

match ⇒ normal access

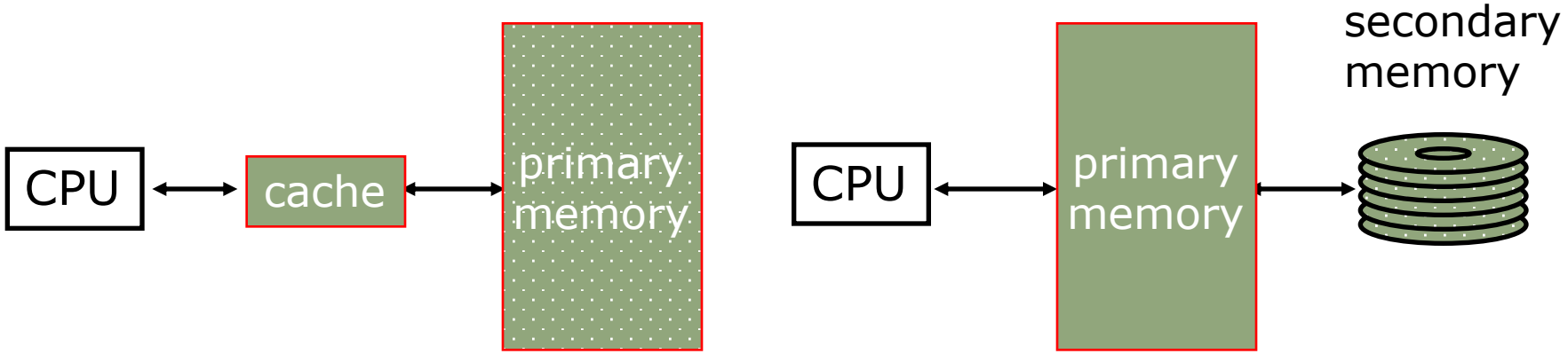
no match ⇒ *page fault*

save the state of the partially executed instruction

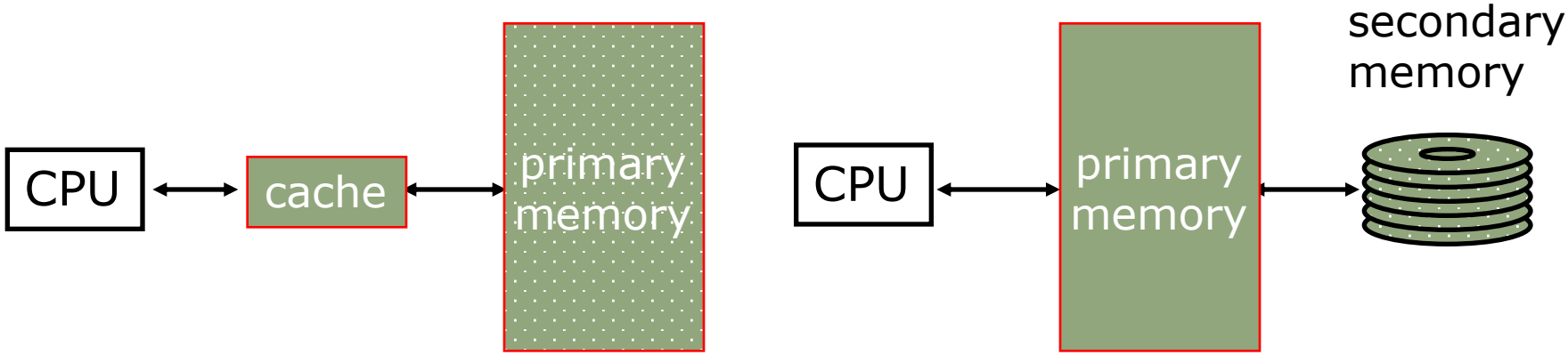
Atlas Demand Paging Scheme

- On a page fault:
  - Input transfer into a free page is initiated
  - The Page Address Register (PAR) is updated
  - If no free page is left, a *page is selected to be replaced* (based on usage)
  - The replaced page is written on the drum
    - to minimize the drum latency effect, the first empty page on the drum was selected
  - The *page table is updated* to point to the new location of the page on the drum

Caching vs. Demand Paging



Caching vs. Demand Paging



Caching

- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled
  - in *hardware*

Demand paging

- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled
  - mostly in *software*

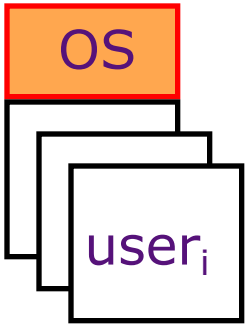
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

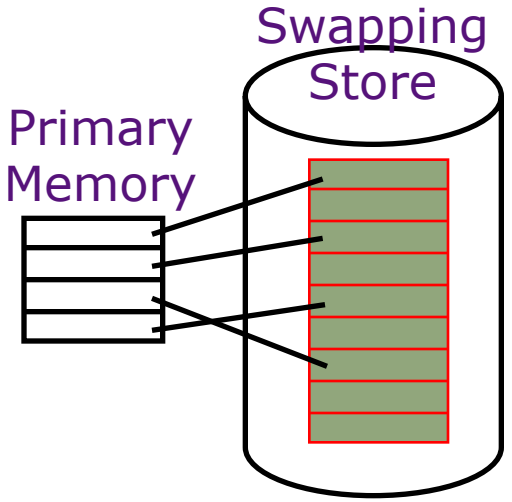
page table ≡ name space



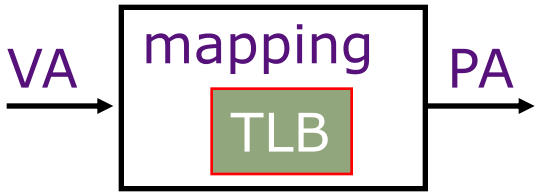
Demand Paging

Provides the ability to run programs larger than the primary memory

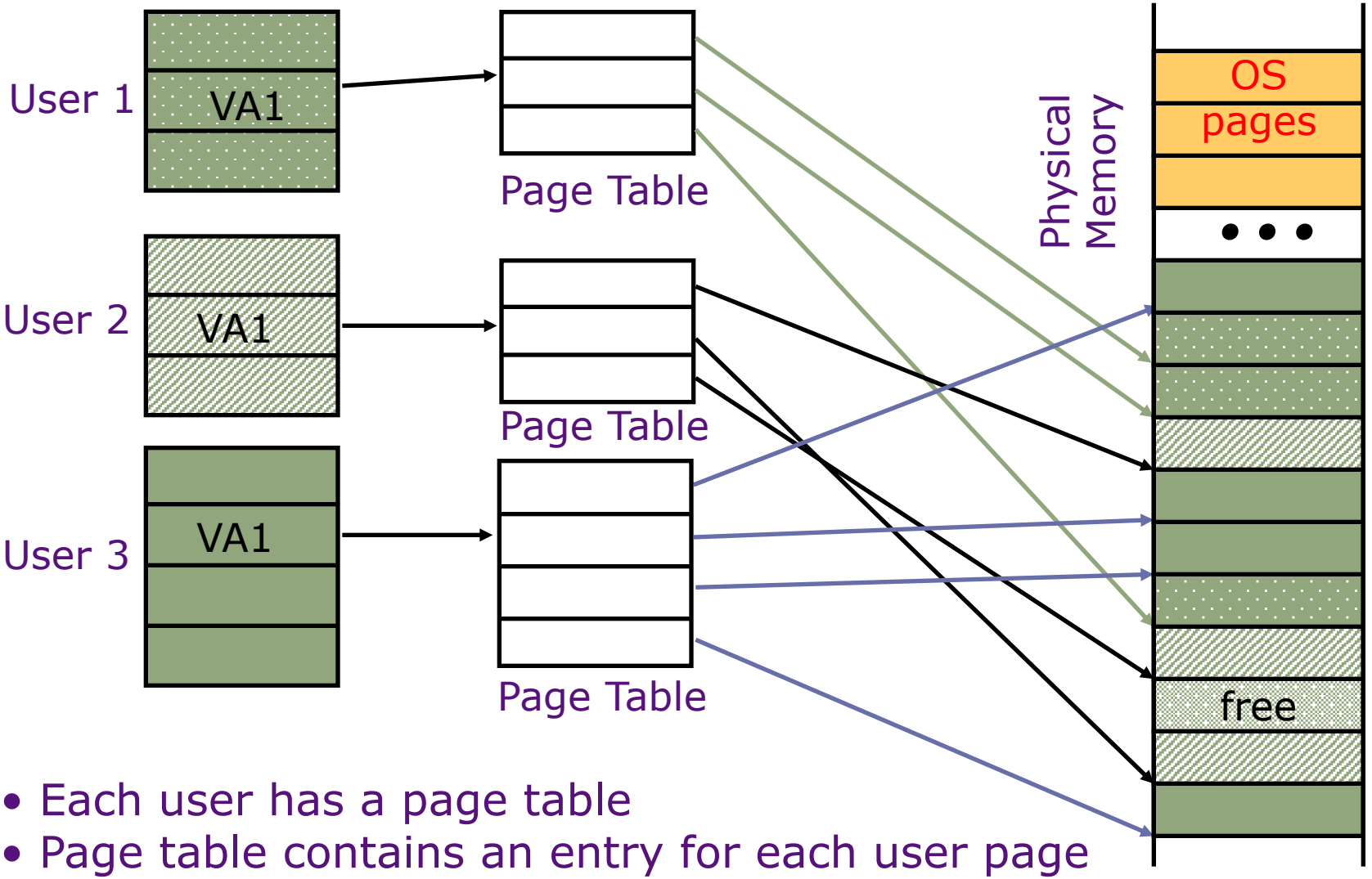
Hides differences in machine configurations



The price is address translation on each memory reference



Private Address Space per User



- Each user has a page table
- Page table contains an entry for each user page

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - ⇒ Space requirement is large
  - ⇒ Too expensive to keep in registers

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - ⇒ Space requirement is large
  - ⇒ Too expensive to keep in registers

- Idea: Keep PT of the current user in special registers
  - may not be feasible for large page tables
  - Increases the cost of context swap

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - ⇒ Space requirement is large
  - ⇒ Too expensive to keep in registers

- Idea: Keep PT of the current user in special registers
  - may not be feasible for large page tables
  - Increases the cost of context swap

- Idea: Keep PTs in the main memory
  - needs one reference to retrieve the page base address and another to access the data word

Where Should Page Tables Reside?

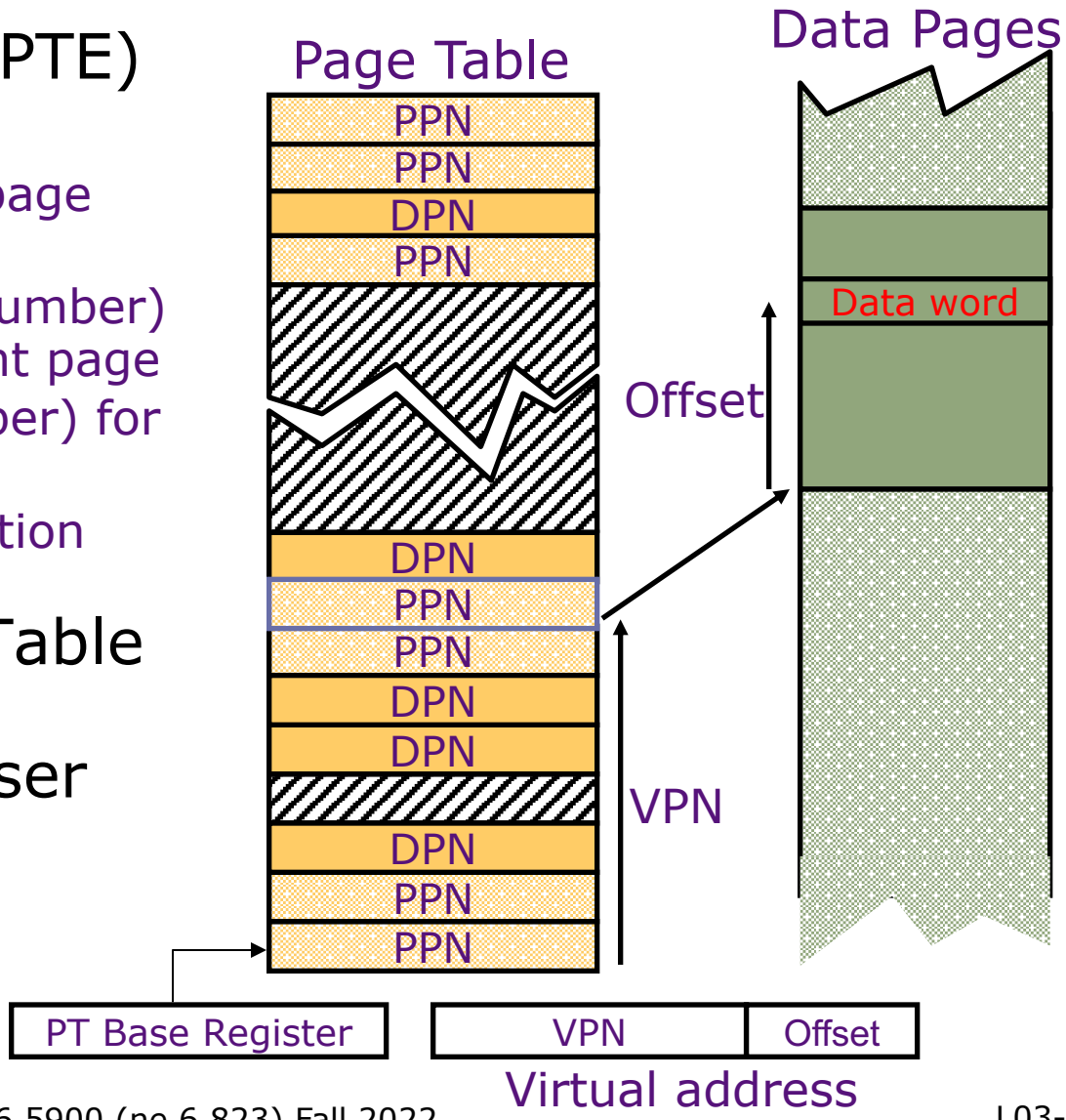
- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - ⇒ Space requirement is large
  - ⇒ Too expensive to keep in registers

- Idea: Keep PT of the current user in special registers
  - may not be feasible for large page tables
  - Increases the cost of context swap

- Idea: Keep PTs in the main memory
  - needs one reference to retrieve the page base address and another to access the data word
    - ⇒ *doubles the number of memory references!*

Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage
- OS sets the Page Table Base Register whenever active user process changes



Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:

- $\Rightarrow$ 2^{20} PTEs, i.e, 4 MB page table per user
- $\Rightarrow$ 4 GB of swap space needed to back up the full virtual address space

Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:

- $\Rightarrow$ 2^{20} PTEs, i.e, 4 MB page table per user
- $\Rightarrow$ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:

- $\Rightarrow$ 2^{20} PTEs, i.e, 4 MB page table per user
- $\Rightarrow$ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

- Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)

Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:

- $\Rightarrow$ 2^{20} PTEs, i.e, 4 MB page table per user
- $\Rightarrow$ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

- Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)

What is the "saving grace"?

Next lecture:

Modern Virtual Memory Systems