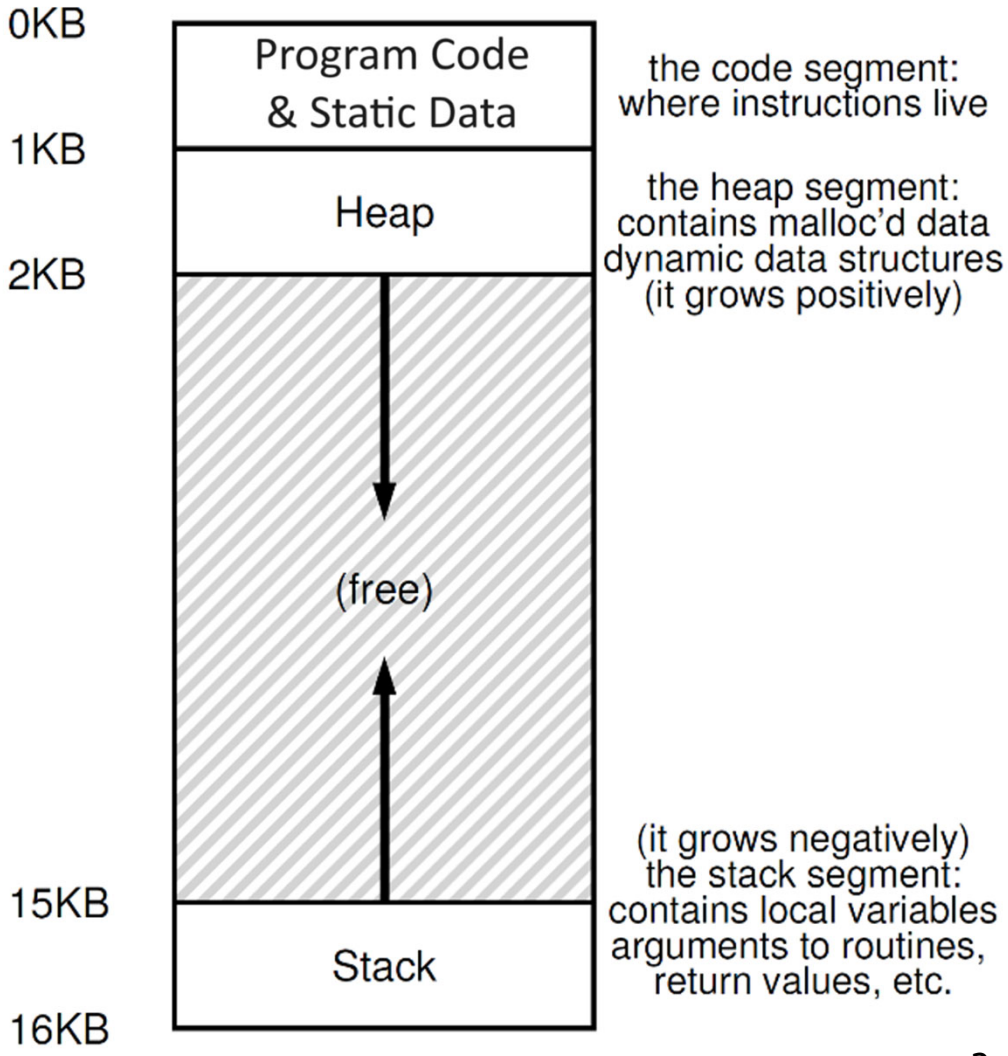
EECE7376: Operating Systems Interface and Implementation

Memory Virtualization

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

The Address Space

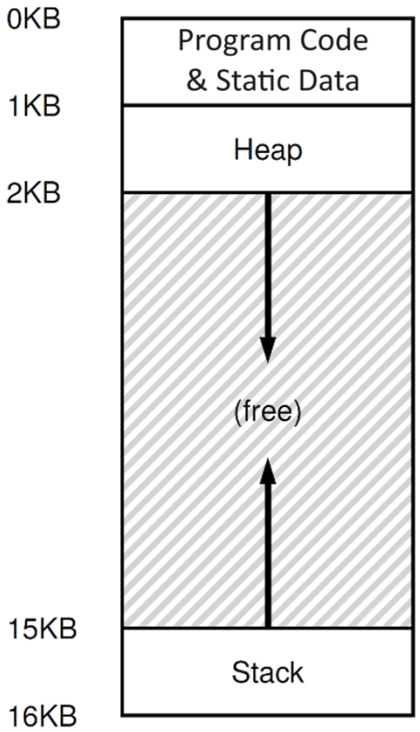
- The **address space** is an easy-to-use abstraction of physical memory, and it is the running program's view of memory in the system.
- The address space of a process contains all the memory state of the running program as shown.
- **Both code and static data** won't need more space as the program runs.
- The **heap** and the **stack** are the two regions of the address space that may **grow and shrink** while the program runs.

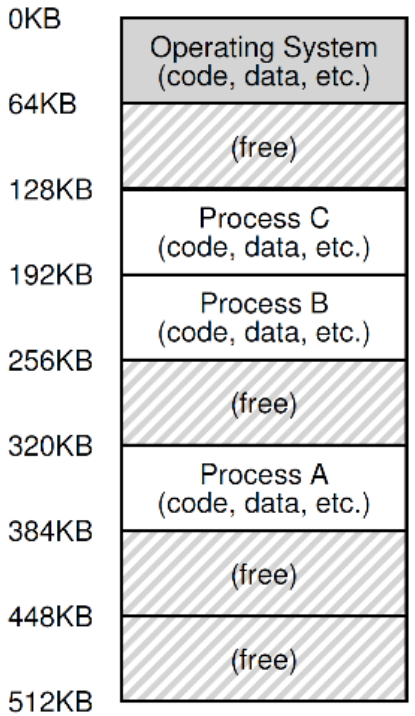


©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Address Space Abstraction

- What we are describing in left diagram below is the **abstraction** that the OS is providing to the running program.
- The program really isn't in memory at physical addresses 0 through 16KB; rather it is loaded at some arbitrary physical address(es).
- Examine processes A, B, and C in right diagram below; there you can see how each process is loaded into memory at a different address.

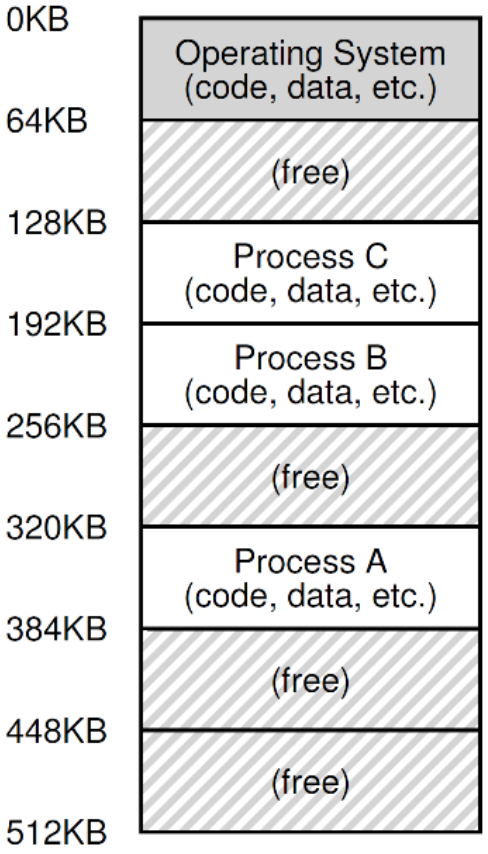




©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Memory Virtualization

- The running process thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space (say 32-bits or 64-bits); the reality is quite different.
- When the OS does this, we say the OS is **virtualizing** the memory.
- When, for example, the shown process ***A*** tries to perform a load at address 0 (which we will call a **virtual address**), somehow the OS, with some hardware support, has to make sure the load doesn't actually go to physical address 0 but rather to physical address 320KB (where *A* is loaded into memory).



Why are there free memory slots in the middle?

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Memory Virtualization Goals

- **Transparency**: The OS should implement **virtual** memory in a way that is invisible to the running program (i.e., the program behaves as if it has its own private physical memory)
- **Efficiency**: The OS should make the memory virtualization efficient, both in terms of **time** (i.e., not making programs run much more slowly) and **space** (i.e., not using too much memory for structures needed to support virtualization).
- **Protection**: The OS should make sure to protect processes from one another as well as the OS itself from the running processes.
  - Protection thus enables us to deliver the property of **isolation** among processes.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Address Translation

- With address translation, the hardware transforms each memory access (e.g., an instruction **fetch**, **load**, or **store**), changing the virtual address provided by the instruction to a physical address where the desired information is actually located.

- The OS must get involved at key points to set up the hardware so that the **correct translations** take place; it must thus **manage memory**, keeping track of which locations are **free** and which are **in use**.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

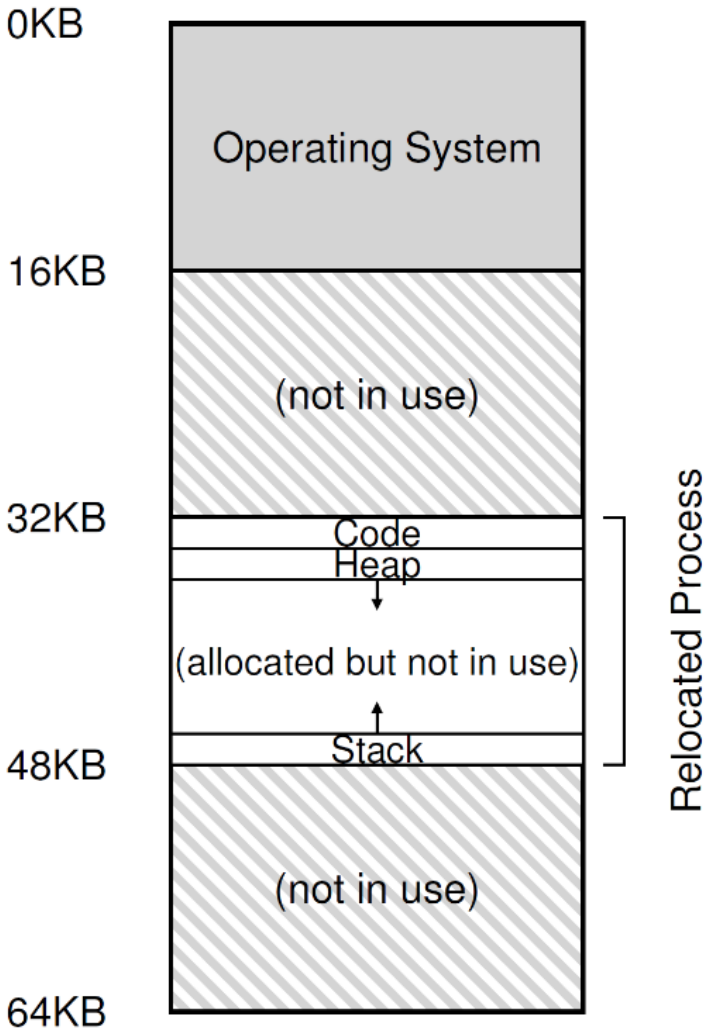
Simple Address Translation

- Initial assumptions for simplicity:
  1. The process address space must be placed **contiguously** in physical memory.
  2. The **size** of a process address space is less than the size of the physical memory.
  3. All processes address spaces are the **same size**.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Simple Address Translation Example

- The shown example shows the physical memory contents with a single **relocated** process following the previous assumption.
- In this example, the OS is using the first **slot** of physical memory for itself. It has relocated the process from a virtual address space starting at 0 into the slot starting at physical memory address 32 KB.
- Each slot is 16 KB.
- The other two slots are free (16 KB-32 KB and 48 KB-64 KB)



©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Dynamic Hardware-based Relocation

- To map from a process virtual address to a physical address after relocation, a simple idea referred to as **base** and **bound**.
- Two hardware **registers** within each CPU are associated with each **running** process: one is called the **base** register, and the other the **bound** (sometimes called the **limit**) register.
  - When a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value. The bound register is set to the max memory space allowed for the process.
  - For the process in the previous example base = 32KB and bound = 16KB.
  - As the process relocation happens at runtime, and the process might be relocated again after it has started running, the technique is often referred to as **dynamic relocation**.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Virtual to Physical Address

- When any memory reference is generated by the process (each memory reference generated by the process is a **virtual address**), it is translated by the processor in the following manner:

```
physical address = virtual address + base
```

- If a process generates a virtual address that is **greater** than the value in the **bound** register (assuming virtual address start at 0), or one that is **negative**, the CPU will raise an exception, and the process will likely be terminated by the OS exception handler.
- The bound register helps with **protection**.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Summary of Hardware Requirements for Dynamic Relocation

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

- A single bit, perhaps stored in some kind of **processor status word**, indicates which mode the CPU is currently running in.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Summary of OS Requirements for Dynamic Relocation

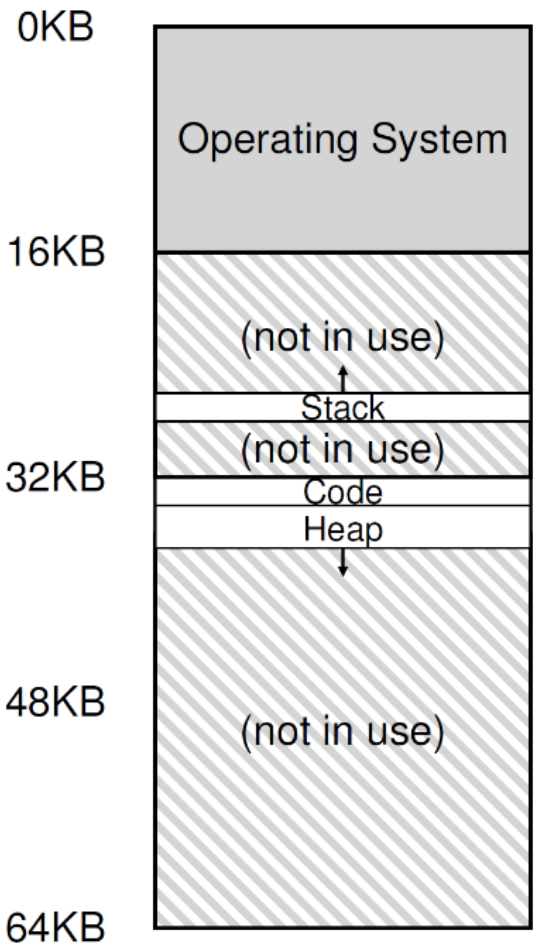
OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via **free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

- Given our assumptions that each address space is (a) smaller than the size of physical memory and (b) the same size, this is quite easy for the OS; it can simply view physical memory as an **array of slots**, and track whether each one of these slots is free or in use.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Memory Segmentation

- There is a waste of physical memory space between the Stack and the Heap when we use base/bound and put the entire address space of each process in the memory. This type of waste is usually called **internal fragmentation**.
- It also makes it quite hard to run a process when the entire address space doesn't fit into memory.
- With **segmentation**, instead of having just one base/bound pair for the entire process address space, we have a **base/bound pair per each logical segment** of the address space: *code*, *stack*, and *heap*.
- As shown, each segment is placed *independently* in physical memory.



©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Address Translation in Segmentation

- A virtual address reference in a procedure needs first to be translated into an offset address within the corresponding segment then added to the base address of that segment.
- One way for the hardware to determine which segment a particular address is in is the **implicit** way where the segment is determined by noticing how the address is formed:
  - If the address is generated from the program counter, then the address is within the code segment.
  - If the address is based off the stack or frame pointer, it must be in the stack segment.
  - Any other address must be in the heap.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

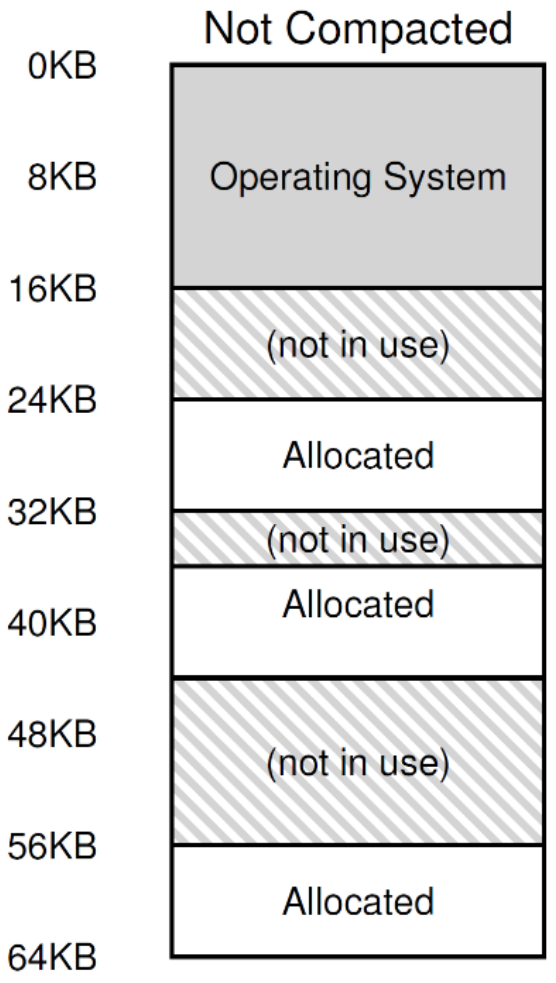
Segmentation OS Support

- The OS must save/restore the segment registers on a context switch.
- When segments (heap or stack) **grow** (or perhaps **shrink**), the OS provides an updated space, updating the segment size (bound) register to the new (bigger or smaller) segment.
- With segmentation, not every address space has the same size (our previous assumption). Now, we have a number of segments per process, and each segment might be **a different size**. The OS needs to allocate/deallocate physical memory spaces for these segments.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

External Fragmentation

- The general problem that arises from allocating/deallocating physical memory spaces is that physical memory quickly becomes **full of little holes** of free space (as shown), making it difficult to allocate new segments, or to grow existing ones. We call this problem **external fragmentation**.
  - In the example, a new process wishes to allocate a 20KB segment. There is a total of about 24KB free, but not in one contiguous segment. Thus, the OS cannot satisfy the 20KB request.



©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

External Fragmentation Solutions

- One solution to this problem would be to **compact** physical memory by rearranging the existing segments.
  - Compaction is expensive, as copying segments is memory-intensive and generally uses a fair amount of processor time.
  - Compaction also makes requests to grow existing segments hard to serve.
- A simpler approach might instead be to use a free-list management algorithm that tries to keep large extents of memory available for allocation.
  - Unfortunately, though, no matter how smart the algorithm, external fragmentation will still exist; thus, a good algorithm simply attempts to minimize it.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Free-Space Management

- Memory free-space management becomes more difficult when the free space consists of **variable-sized units**.
- This happens when using segmentation to implement virtual memory, which results in **external fragmentation**.
- Some basic algorithms for managing free space are: **Best Fit**, **Worst Fit**, **First Fit**, and **Next Fit**.
- These algorithms maintain a free list of elements that describe the free space remaining in the heap.
  - *Example*: a free linked list that has three elements of sizes 10, 30, and 20 (the starting memory address of each element is omitted for simplicity):

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Best Fit

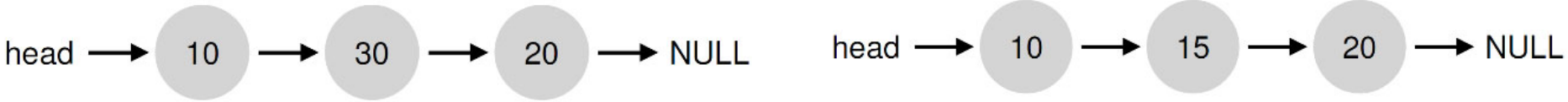
- Best fits searches through the free list and find chunks of free memory that are as big or bigger than the requested size.
- Then, return the one that is the smallest in that group of candidates; this is the so called **best-fit** chunk (it could be called **smallest fit** too).
- *Example:* Assume an allocation request of size 15 from the left list. Best fit will result in the free list on the right:

- *Pros*: It tries to reduce wasted space.
- *Cons*: It pays a heavy performance penalty by performing an exhaustive search for the smallest free block that fits.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Worst Fit

- Worst fits searches through the free list and find chunks of free memory that are as big or bigger than the requested size.
- Then, return the **largest** chunk in that group of candidates.
- *Example:* Assume an allocation request of size 15 from the left list. Worst fit will result in the free list on the right:

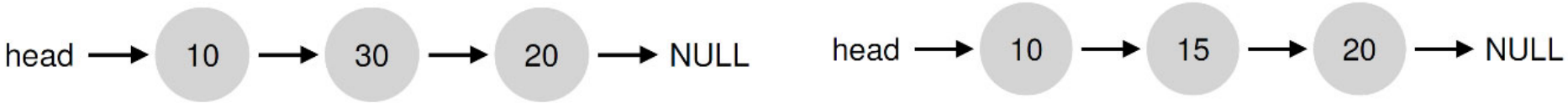


- *Pros*: It tries to leave big chunks free instead of lots of small chunks.
- *Cons*: It pays a heavy performance penalty by keeping track of the largest free block (less penalty than the best fit though)

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

First Fit

- First fit simply finds the first block that is big enough and returns the requested amount to the user.
- *Example:* Assume an allocation request of size 15 from the left list. First fit will result in the free list on the right (same result as worst fit in this example):



- *Pros*: It has the advantage of speed — no exhaustive search of all the free spaces are necessary.
- *Cons*: It sometimes pollutes the beginning of the free list with many small objects. Next memory allocation will need to go through these many objects until it finds the "first fit".

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Next Fit

- Next fit is like first fit but instead of always beginning the search at the beginning of the list, the next fit algorithm keeps an extra pointer to the location within the list where one was looking last.

- *Pros*: It has the advantage of speed, and it spreads the searches for free space throughout the list more uniformly.
- *Cons*: The need to maintain that extra pointer.

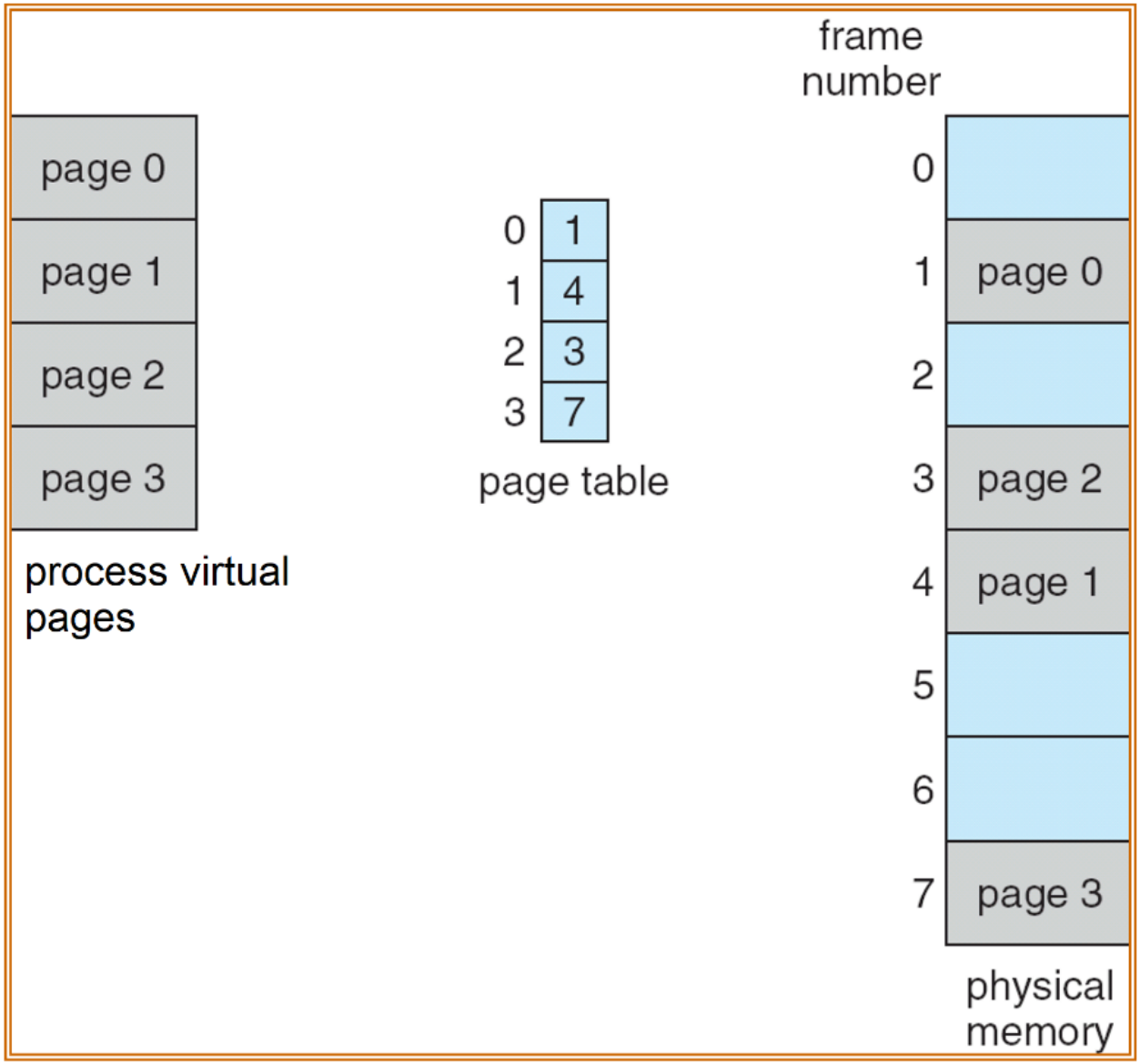
©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Paging

- Instead of splitting up a process's address space into variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a **page**.
- Correspondingly, we view physical memory as an array of fixed-sized slots called **page frames**; each of these frames can contain a single virtual-memory page.
- Paging makes **free-space management** simpler.
  - The OS keeps a **free list** of all free physical frames, and just grabs the first free pages needed for a process off of this list.
- To record where each virtual page of the address space is placed in physical memory, the OS keeps a *per-process* data structure known as a **page table**.
  - The page table stores address translations for each of the virtual pages, thus letting us know where in physical memory each page resides.

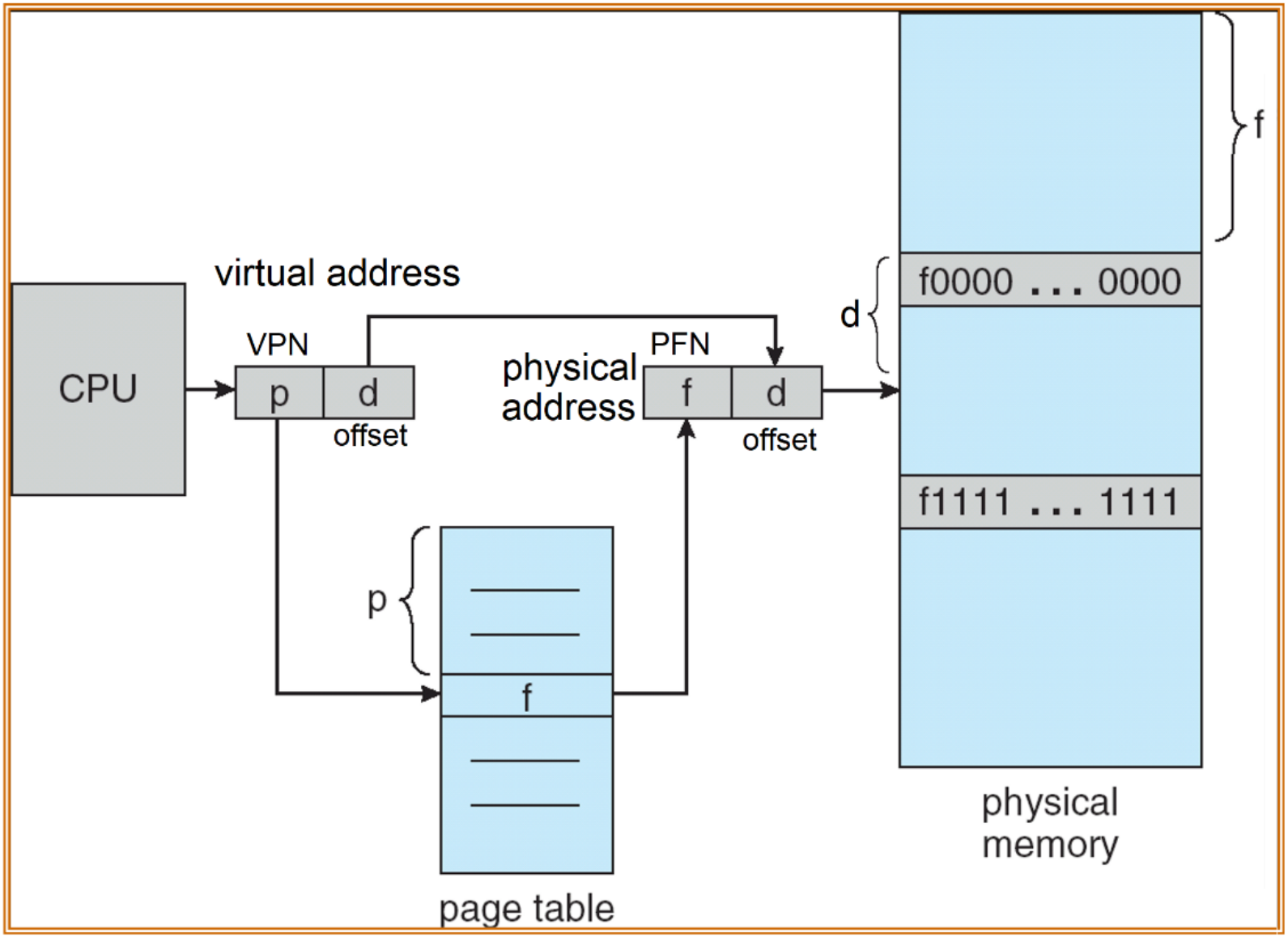
© 2005 Silberschatz, Galvin, and Gagne "Operating System Concepts", 7/e John Wiley & Sons

Page Table Example



© 2005 Silberschatz, Galvin, and Gagne "Operating System Concepts", 7/e John Wiley & Sons

Address Translation Example



virtual page number (VPN) physical frame number (PFN)

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

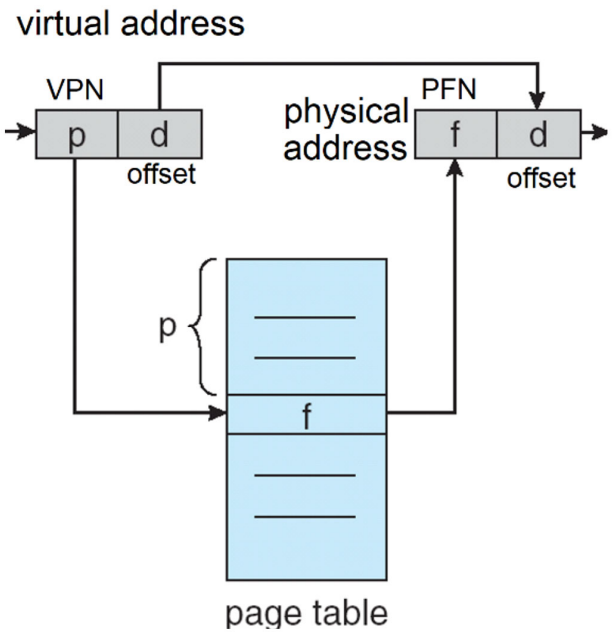
Where Are Page Tables Stored?

- Page tables can get terribly large.
- *Example:*
  - 32-bit virtual address space with 4KB pages → 12-bit offset (to address bytes within each page) → The 20 remaining bits are for the VPN.
  - A 20-bit VPN implies that there are 2^{20} translations that the OS would have to manage for each process.
  - Assume we need 4 bytes per **page table entry** (**PTE**) , we get an immense 4MB of memory needed for each page table.
  - Now imagine there are 100 processes running, this means the OS would need 400MB of memory just for all those address translations!

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Page Table Structure

- The simplest structure of a page table is called a **linear page table**, which is just an array.
- The OS indexes the array by the virtual page number (**VPN**) and looks up the page-table entry (**PTE**) at that index in order to find the desired physical frame number (**PFN**).
- The PTE has an entry called the **valid bit** to indicate whether the particular translation is valid. Its task corresponds to the bound register in segmentation.
- *Example:* when a process starts running, it will have code and heap at one end of its address space, and the stack at the other. All the **unused space** in-between will be marked **invalid**, and if the process tries to access such space, a trap will be generated to the OS, which will likely terminate the process.
  - It also means that there is no **need to allocate physical frames** for those unused pages in the virtual space.



©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

More PTE Contents

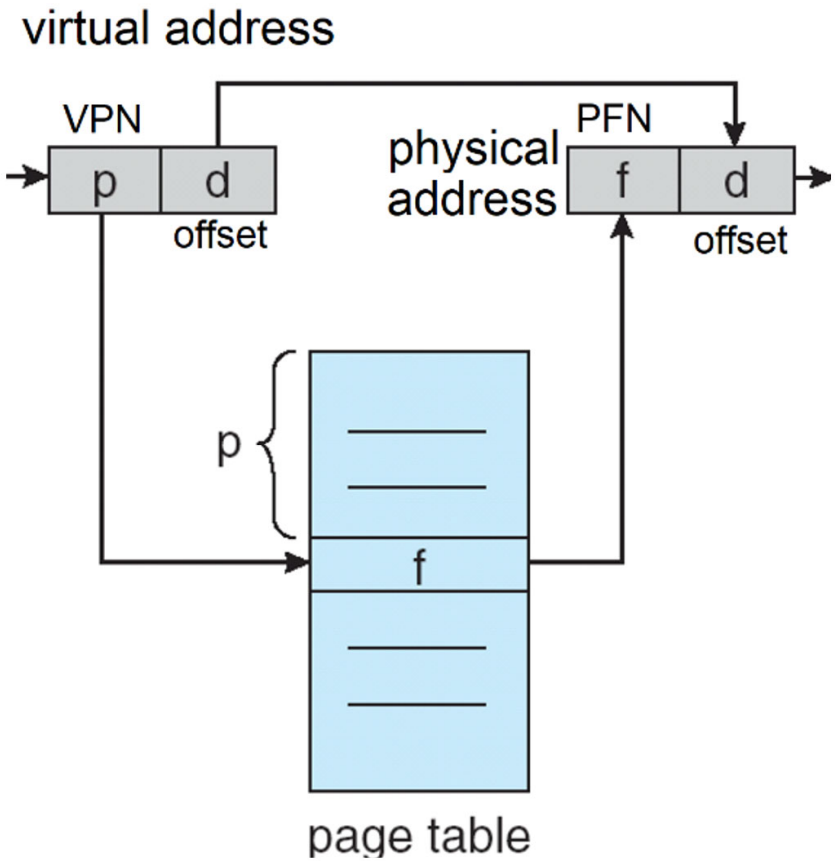
- The **protection bits**, indicating whether the page could be read from, written to, or executed from.
- A **present bit** indicates whether this page is in physical memory or on disk (i.e., it has been **swapped out**).
- A **dirty bit** indicates whether the page has been modified since it was brought into memory from the disk.
- A **reference bit** (a.k.a. **accessed bit**) is used to track whether a page has been accessed. It is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during **page replacement**.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Paging is Too Slow

- The following are the calculations needed **by hardware** to find the physical address with paging. Assume a single **page-table base register** (**PTBR**) contains the physical address of the page table.

```
// Loading Register with data from VirtualAddress
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))
// Fetch the PTE
PTE = AccessMemory(PTEAddr)
// Check if process can access the page
if (PTE.Valid == False)
RaiseException(SEGMENTATION_FAULT)
  else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else  // Access is OK: form physical address and fetch it
              offset = VirtualAddress & OFFSET_MASK
              PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
              Register = AccessMemory(PhysAddr)
```



©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

How to Speed up Address Translation

- With paging, going to memory for translation information before every instruction fetch or explicit load or store is too slow.
- To speed address translation, a **translation-lookaside buffer**, or **TLB** is added to the chip's memory-management unit (MMU) and is simply a hardware **cache** of **frequently** used **valid** virtual-to-physical address translations.
- Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein; if so, the translation is performed (quickly) without having to consult the page table in the main memory (which keeps all translations).

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Hardware-Managed TLB Basic Algorithm

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess (TlbEntry.ProtectBits) == True) // Note: only Valid entries are in TLB
      Offset = VirtualAddress & OFFSET MASK
      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
      Register = AccessMemory(PhysAddr)
    else    RaiseException(PROTECTION_FAULT)
 else // TLB Miss
     PTEAddr = PTBR + (VPN * sizeof(PTE))
     PTE = AccessMemory(PTEAddr)
     if (PTE .Valid == False)  RaiseException(SEGMENTATION_FAULT)
     else
         if (CanAccess(PTE.ProtectBits) == False) RaiseException(PROTECTION_FAULT)
         else // updates the TLB with the translation
             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
             RetryInstruction()
```

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

TLB Caching Principles

- A **TLB hit** happens when the CPU finds the address translation in the TLB. Otherwise, it is a **TLB miss**.
  - When a miss occurs, the hardware updates the TLB with the missed translation.
  - The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache (i.e., are hits). If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast.
  - When a miss occurs, the high cost of paging is incurred as it leads to more memory accesses.
- Like all caching, the goal is to increase the **hit rate**, and this is achieved by taking advantage of the **temporal locality** and **spatial locality** of instruction and data references.
- The TLB is a **fully-associative cache** (i.e., any given translation can be anywhere in the TLB). The hardware searches the entries in parallel to see if there is a match.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Software-Managed TLB

- RISC, reduced-instruction set computers, have what is known as a **software-managed TLB**.
  - On a TLB miss, the hardware simply raises an exception, which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a trap handler.
- This trap handler is a code within the OS that is written with the express purpose of handling TLB misses.
  - When run, the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction (resulting in a TLB hit).
  - When returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that caused the trap (**not the following instruction** like in regular trap).
  - Thus, depending on how a trap or exception was caused, the hardware must save the appropriate PC (program counter) when trapping into the OS

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Software-Managed TLB Advantages

- The primary advantage of the software-managed approach is *flexibility*: the OS can use **any data structure** it wants to implement the page table, without necessitating hardware change.
- Another advantage is *simplicity*: the **hardware is simpler** as it doesn't do much on a miss: just raise an exception and let the OS TLB miss handler do the rest.
- Finally, the OS can handle the segmentation and protection **faults directly**. In case of segmentation fault, the OS terminates the process without the need to return from the TLB miss-handling trap.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

TLB Issues

- **Context Switches**:
  - One approach is to simply **flush the TLB** on context switches, thus emptying it before running the next process.
  - Each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high
- **Replacement Policy**:
  - As with any cache, when we are inserting a new entry in a full TLB, we must replace an old one, and thus the question: *which one to replace?*
  - One common approach is to evict the **least-recently-used** or **LRU** entry.
  - Another typical approach is to use a **random** policy, which evicts a TLB mapping at random

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Paging: Smaller Tables

- Page tables are too big and thus consume too much memory.
- A 32-bit address space, with 4KB pages and a 4-byte page-table entry will need 4MB per table.
- We usually have one page table for every process in the system.
- With a hundred active processes, we will be allocating hundreds of megabytes of memory just for page tables.
- Simple solution: use bigger pages.
  - A 32-bit address space, with 16KB pages and a 4-byte page-table entry will need $2^{18} \times 2^2 = 1$MB per table.
  - *Problem*: big pages lead to waste within each page → **internal fragmentation**.
  - Thus, most systems use relatively small page sizes in the common case: 4KB (as in x86) or 8KB (as in SPARCv9).

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

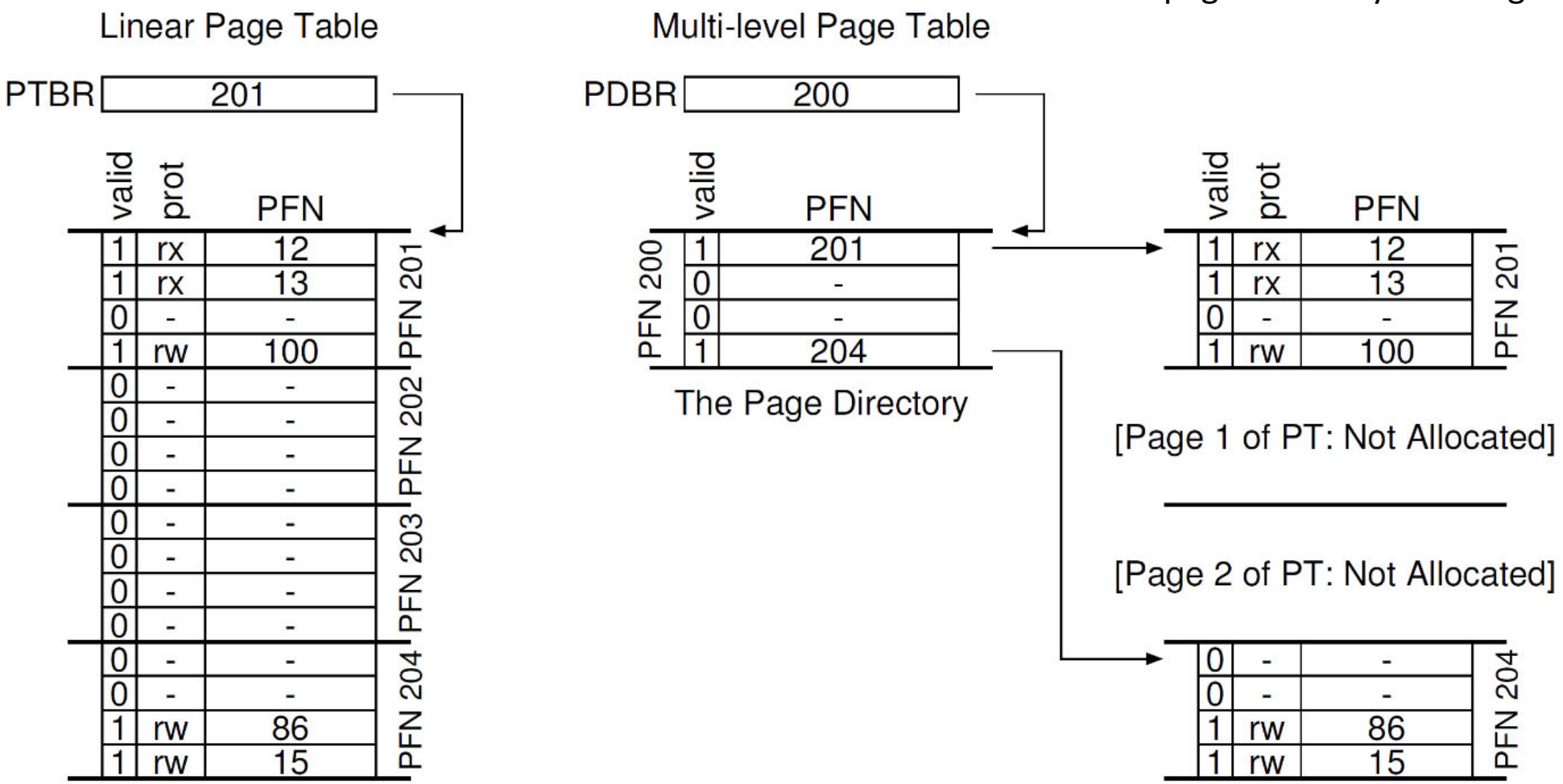
Multi-level Page Tables

- The **multi-level page table** tries to get rid of all those invalid regions in the page table by turning the linear page table into something like a **tree** (a hierarchy of tables).
- The multi-level page table is formed as follows:
  - Assume the original linear page table itself is divided into **units**.
  - If a unit has **all** of its **page-table entries** (**PTE**s) invalid, the unit is marked as **invalid** and is not allocated in the memory.
  - If a unit has **at least one** of its page-table entries (PTEs) valid, the unit is marked as **valid** and is allocated in the memory.
  - A **page directory** contains entries (**page directory entries**, **PDE**) for all units. A PDE contains a bit for each unit to indicate whether the unit is valid or invalid. For a valid unit, the PDE provides its **PFN** (physical frame number).

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Linear vs. Multi-Level Page Tables

PTBR: page-table base register
PDBR: page-directory base register



Contiguous space in physical memory

No need to be in a contiguous space in physical memory

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Advantages of Multi-level Page Tables

- The multi-level table only allocates page-table space in proportion to the amount of address space used by the process; thus, it is generally compact and supports sparse address spaces.
- Each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table.
  - Contrast this to a simple (non-paged) linear page table, which is just an array of PTEs indexed by VPN; with such a structure, the entire linear page table must **reside contiguously** in physical memory.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Disadvantages of Multi-level Page Tables

- On a TLB miss, two loads from memory will be required to get the right translation information from the page table (one for the page directory, and one for the PTE itself), in contrast to just one load with a linear page table. Thus, the multi-level table is a small example of a **time-space trade-off**.
- Another disadvantage is **complexity**. Whether it is the hardware or OS handling the page-table lookup (on a TLB miss), doing so is undoubtedly more involved than a simple linear page-table lookup.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

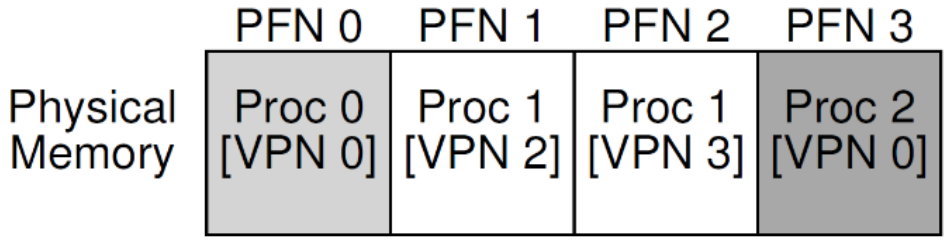
Swap Space

- We've been assuming that every address space of every running process fits into the physical memory.
- However, to support large address spaces, the OS will need a place to **swap** away portions of address spaces that currently aren't in great demand. In modern systems, this swap place is usually the **hard disk drive**.
- The addition of swap space allows the OS to support the illusion of **a large virtual memory** for multiple concurrently running processes.
- The first thing we will need to do is to reserve some space on the disk for moving pages back and forth. In operating systems, we generally refer to such space as **swap space**.

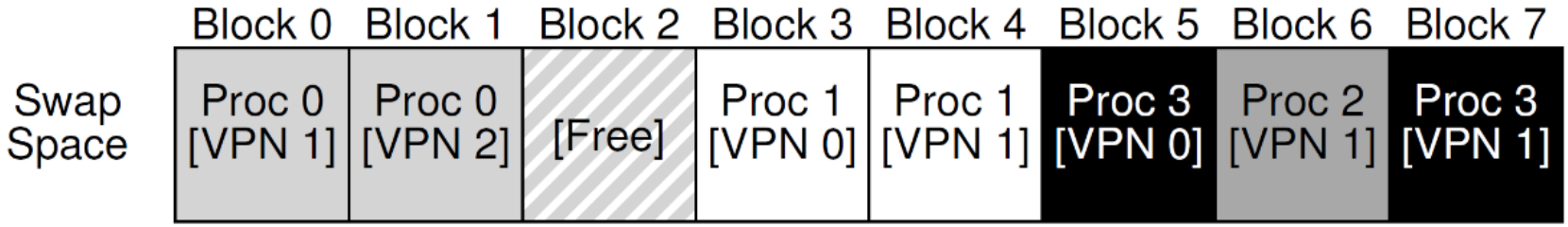
©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Physical Memory and Swap Space Example

- The following is a little example of a 4-page physical memory and an 8-page swap space.
- Proc 0, Proc 1, and Proc 2 are actively sharing physical memory; each of the three processes only have some of their valid pages in memory, with the rest located in swap space on disk.
- A fourth process (Proc 3) has all its pages swapped out to disk, and thus clearly isn't currently running.
- Using swap space allows the system to pretend that memory is larger than it actually is.



VPN: virtual page number



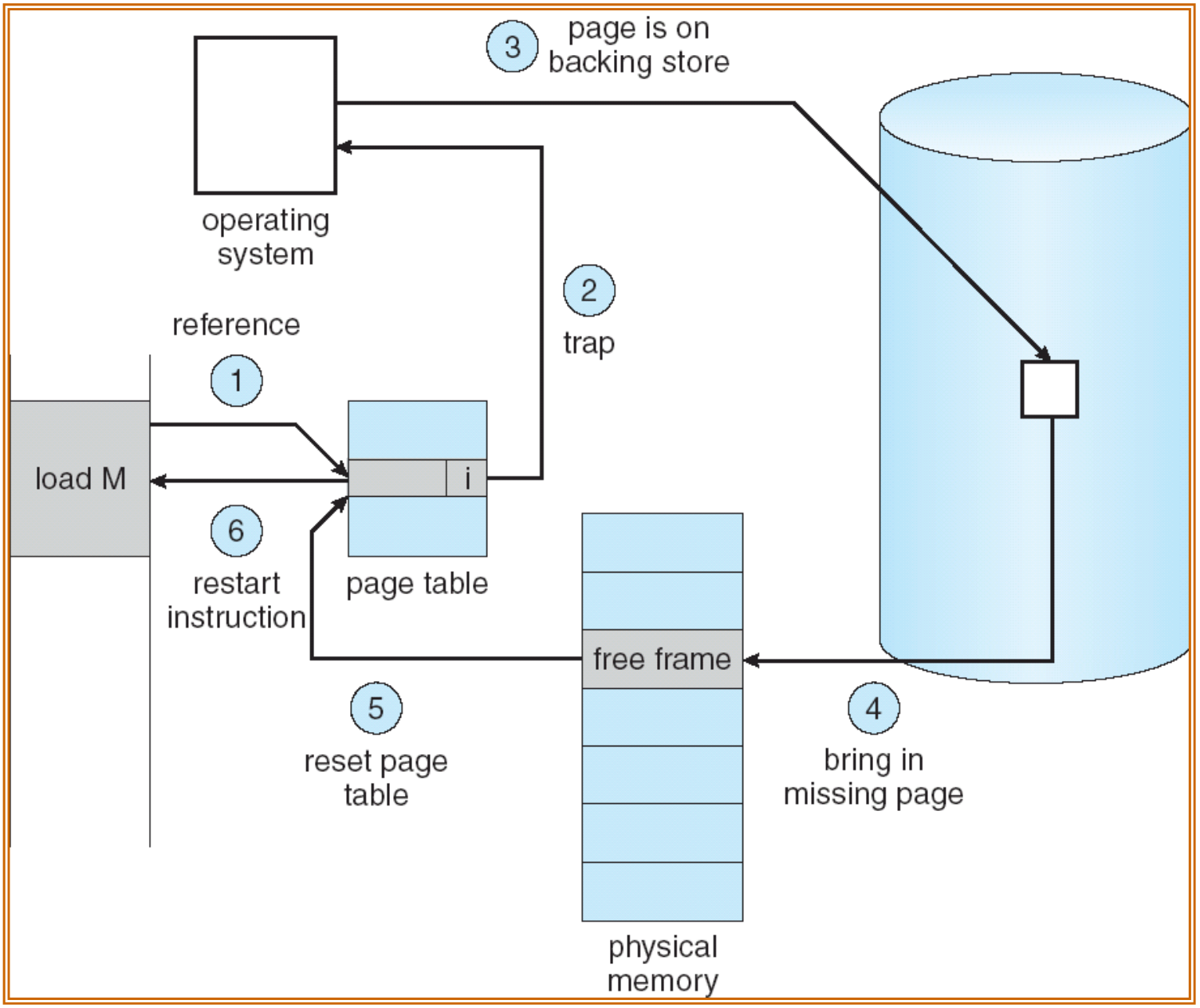
©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

The Present Bit and Page Fault

- To allow pages to be swapped to disk, the system needs to determine when it looks in the **PTE** whether the page presents in physical memory or not.
- The way the hardware (or the OS, in a software-managed TLB approach) determines this is through a new piece of information in each page-table entry, known as the **present bit**.
  - If the present bit is set to one, it means the page is present in physical memory and everything proceeds as before; if it is set to zero, the page is not in memory but rather on disk somewhere.
  - The **page disk address** can be stored in the PTE bits that are normally used for the page PFN (physical frame number) .
- The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.
- Upon a page fault, the OS runs the **page-fault handler** code to service the page fault.

© 2005 Silberschatz, Galvin, and Gagne "Operating System Concepts", 7/e John Wiley & Sons

Steps in Handling a Page Fault



©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Memory High and Low Watermark

- Most operating systems keep a small amount of physical memory pages free all the time by having some kind of **high watermark** (*HW*) and **low watermark** (*LW*).
- When there are fewer than LW pages available, the OS runs a background thread, sometimes called the **swap daemon** or **page daemon**, that is responsible for swapping pages out until there are *HW* pages available.
- The process of picking the pages to swap out, or replace, is known as the **page-replacement policy**.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Page Replacement Policies

- The **optimal** replacement policy replaces the page that will be accessed ***furthest in the future*** as it results in the fewest-possible page faults. Such policy is difficult to implement!
- Other practical, but not optimal, replacement policies include:
  - **First-In, First-Out** (**FIFO**)
  - **Random**: it is simple to implement, but it might replace a page that is about to be referenced again.
  - **Least-Recently-Used** (**LRU**)
  - **Least-Frequently-Used** (**LFU**)
  - **Most Frequently Used** (**MFU**): based on the argument that the page that was least used was probably just brought in.
- One way to implement the above policies is to associate a **timer** or **counter** field with every page. Every time page is referenced, copy the clock into its time field or increment the counter field (this can be done by the hardware). When a page needs to be replaced, the OS looks at these fields to determine which pages to be replaced following one of the above policies.

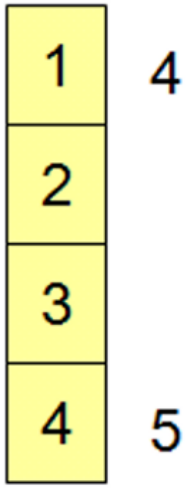
© 2005 Silberschatz, Galvin, and Gagne "Operating System Concepts", 7/e John Wiley & Sons

Optimal Page Replacement Example

- Replace page that will be accessed furthest in the future.
- Assume having 4 physical memory frames with the following 5-page process references:

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

In the shown trace, a page number is written when a fault happens.
Recall that we start with empty four frames.
When we have a tie case, we can apply FIFO.



6 page faults

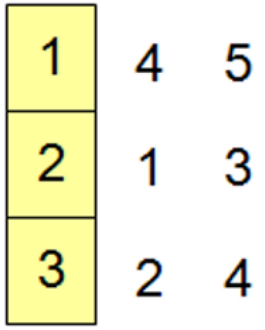
- As this approach is not practical, it is used mainly to measure how close any other algorithm to the optimal solution.

© 2005 Silberschatz, Galvin, and Gagne "Operating System Concepts", 7/e John Wiley & Sons

FIFO Page Replacement Examples

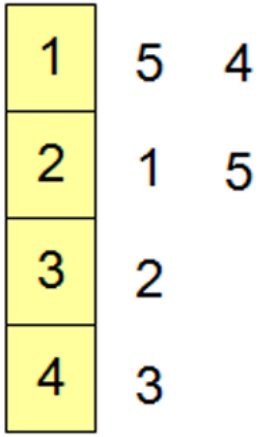
- Replace page that first entered the physical memory.
- Assume having 3 physical memory frames and a 5-page process with the following page references:

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



9 page faults

- Now assume 4 physical memory frames for the same above page references:



10 page faults

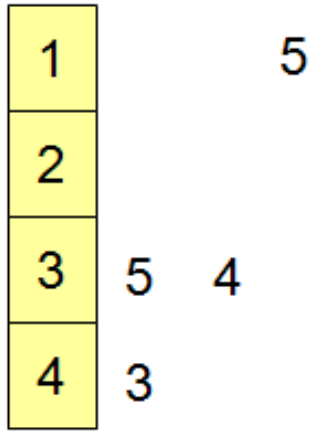
- FIFO Replacement – Belady's Anomaly more frames but more page faults !!

© 2005 Silberschatz, Galvin, and Gagne "Operating System Concepts", 7/e John Wiley & Sons

LRU Page Replacement Example

- Replace page that was least-recently-used.
- Assume having 3 physical memory frames and a 5-page process with the following page references:

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

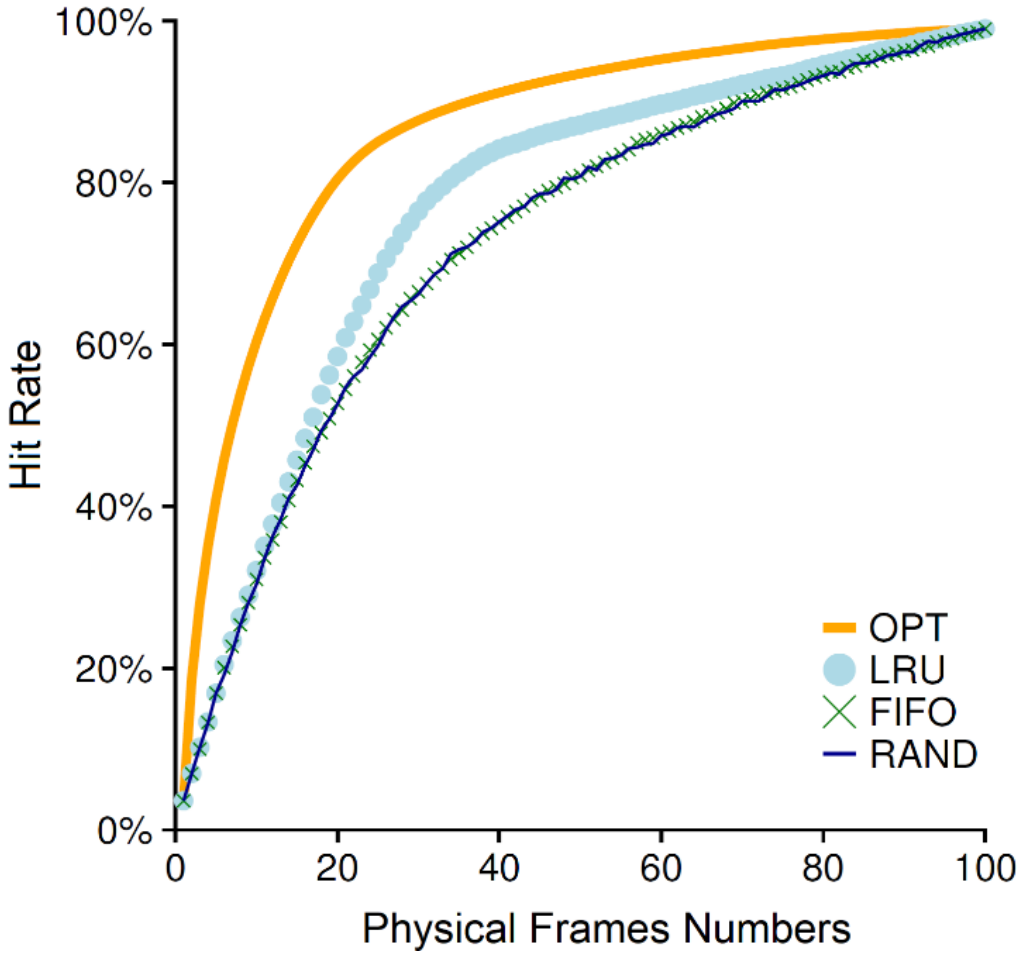


8 page faults

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

The 80-20 Workload Example

- A process with 100 unique pages and an overall 10,000-page references.
- The workload exhibits locality: 80% of the references are made to 20% of the pages; the remaining 20% of the references are made to the remaining 80% of the pages.
- While both random and FIFO do reasonably well, LRU does better.

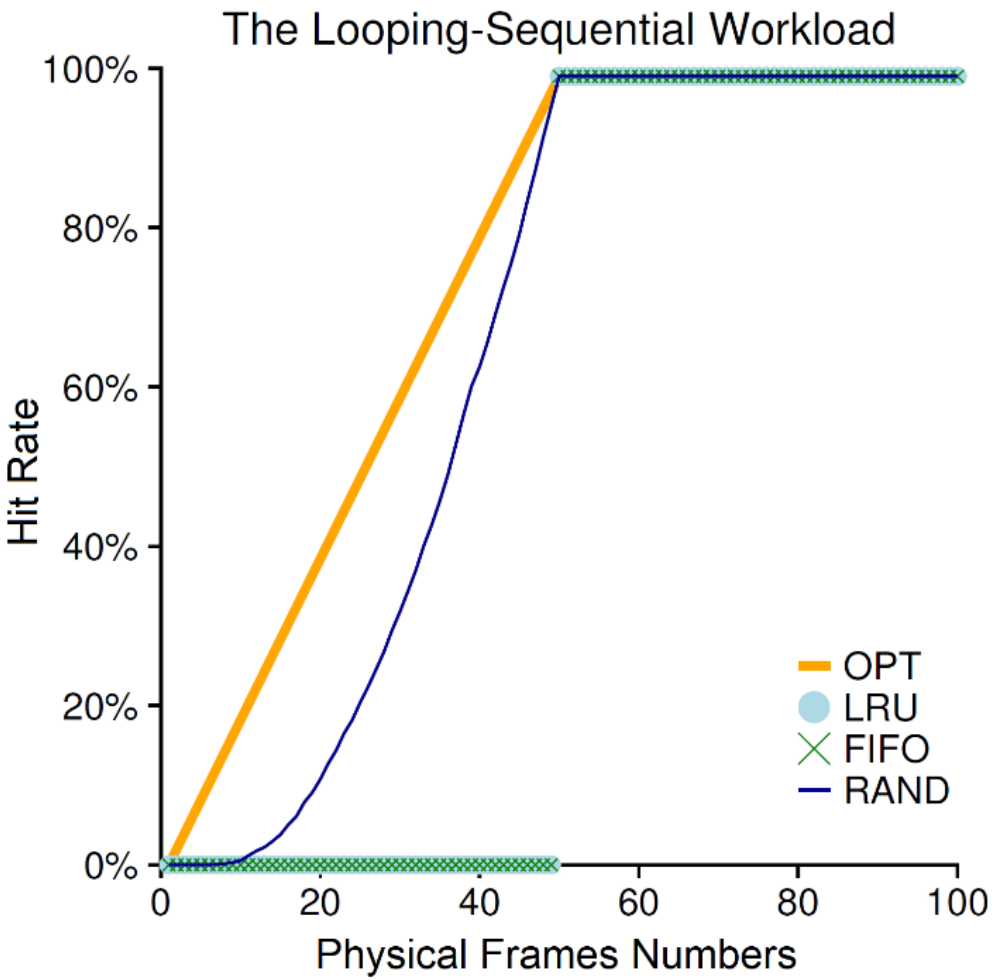


The physical frame numbers vary from 1 to enough to hold all the 100 unique pages.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

The Looping-Sequential Workload Example

- A process with 50 unique pages and an overall 10,000-page references.
- The workload exhibits a sequential reference to the 50 pages in a loop.
- Up to physical memory size of 49 frames, both LRU and FIFO have 0% hit rate as they replace older pages that are going to be accessed sooner than the pages that the policies prefer to keep in cache.
- Random performs notably better for this workload.



The physical frame numbers vary from 1 to enough to hold all the 50 unique pages.

©2021 R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau., "Operating Systems: Three Easy Pieces"

Considering Dirty Pages

- If a page has been **modified** while in the memory and is thus **dirty**, it must be written back to disk when it is replaced out of memory, which is an expensive operation.
- If the page has not been modified (and is thus **clean**), the replacement is free; the physical frame can simply be reused.
- The hardware should include a **modified bit** (a.k.a. **dirty bit**). This bit is set any time a page is modified, and thus can be incorporated into the page-replacement algorithm.
- Instead of writing dirty pages one at a time, many systems collect a number of pending dirty pages and write them to disk in one (more efficient) write. This behavior is usually called **clustering** or simply **grouping** of writes.
- Some replacement policies prefer to **replace clean pages** over dirty pages.

Readings and Resources List

- From Arpaci-Dusseau textbook:
  - Chapter 14: Interlude: Memory API
  - Chapter 20:
    - A Detailed Multi-Level Example
    - The Translation Process: Remember the TLB
  - Chapter 21: Page Fault Control Flow