Additional details about virtual memory

M1 MOSIG – Operating System Design

Renaud Lachaize

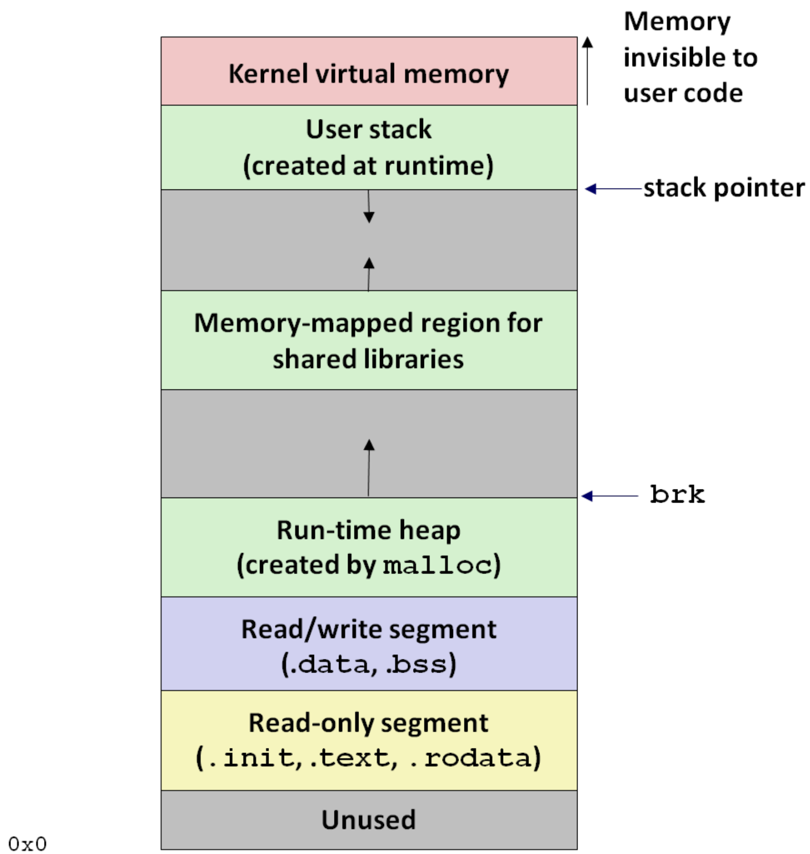
Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - (many slides/figures directly adapted from those of the CS140 class)
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)
    - Textbook: Computer Systems: A Programmer's Perspective (2^{nd} Edition)
    - CS 15-213/18-243 classes
  - Textbooks (Silberschatz et al., Tanenbaum)

Outline

- Systems calls related to virtual memory
- Copy-on-Write
- Paging in day-to-day use
- Hardware/OS paging extensions
- Exposing page faults to applications

Recall typical virtual address space



- Dynamically allocated memory goes in heap
- Top of heap called "breakpoint" (**`brk`**)
  - (Do not confuse with debugging breakpoints)

Early VM system calls

- OS keeps "breakpoint" – top of data segment (heap)
  - Memory addresses between breakpoint and next region trigger fault on access

- **`char *brk(const char addr);`**
  - Set and return new value of breakpoint

- **`char *sbrk(int incr);`**
  - Increment value of breakpoint and return old value

- On modern systems, applications should not directly use such calls
  - They will be called indirectly through invocations of **`malloc`** or the **`mmap`** system call (described next)

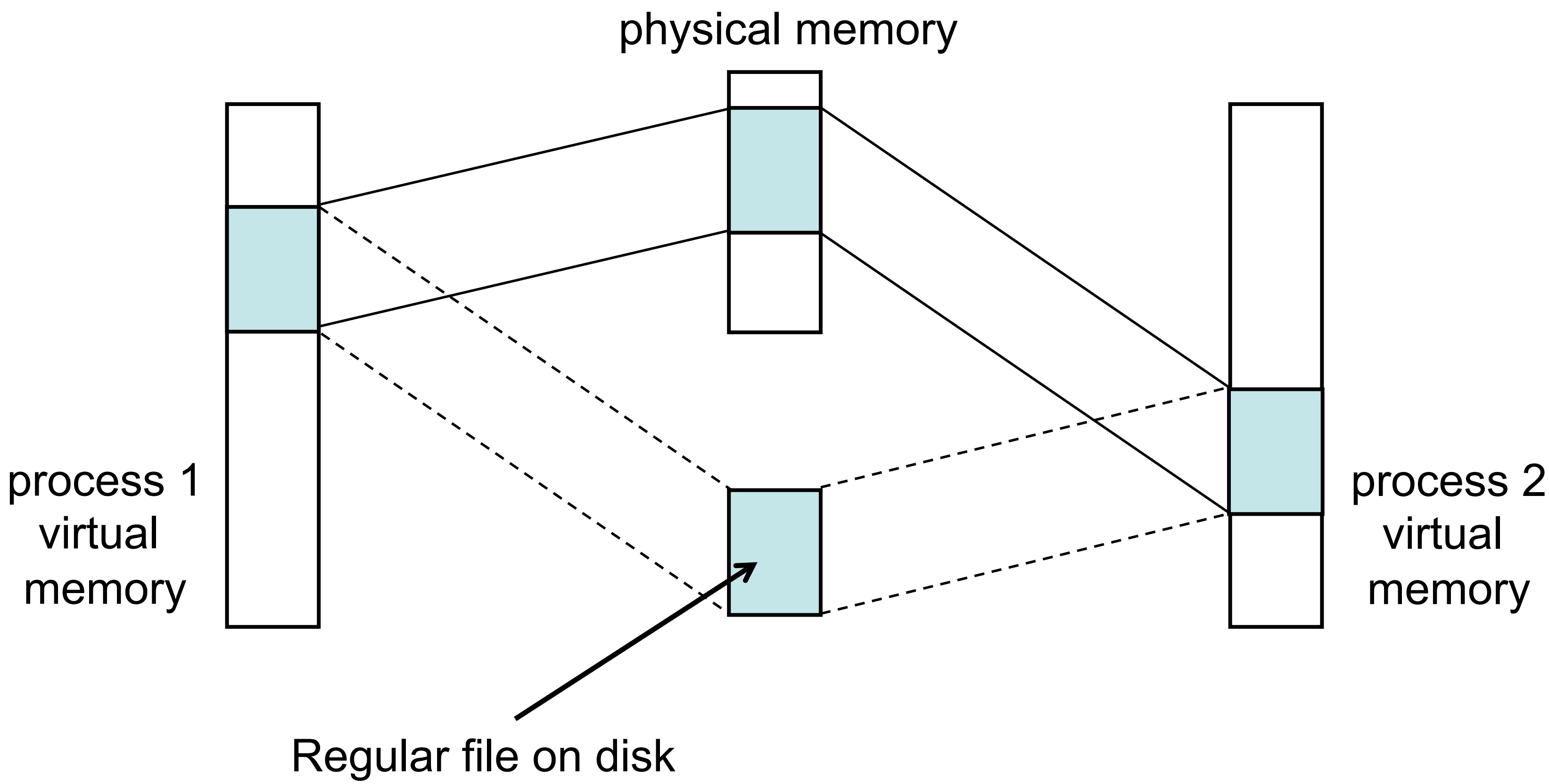
Memory-mapped files

- **Key idea**: associate an address range within an address space (a.k.a. *"memory area"/"region"/"zone"*, and sometimes *"segment"*) with the contents of a "backing" file (or a portion of a backing file)
- **Useful**
  - For the OS, when building the contents of an address space
  - For the application programmers (makes code simpler and/or more efficient)
  - See details in the next slides
- **Two different kinds of backing files**
  - **Regular (persistent) files**:
    - Initial page bytes come from this file
    - Updated bytes may (or may not, depending on settings) be propagated to the backing file (and become persistent)
  - **Fake file full of zeros, called "demand-zero" or "anonymous"**
    - Does not need to be read from disk
    - Once the page is modified (dirtied), treated like any other page
    - Updates are not persistent

Memory-mapped files (continued)

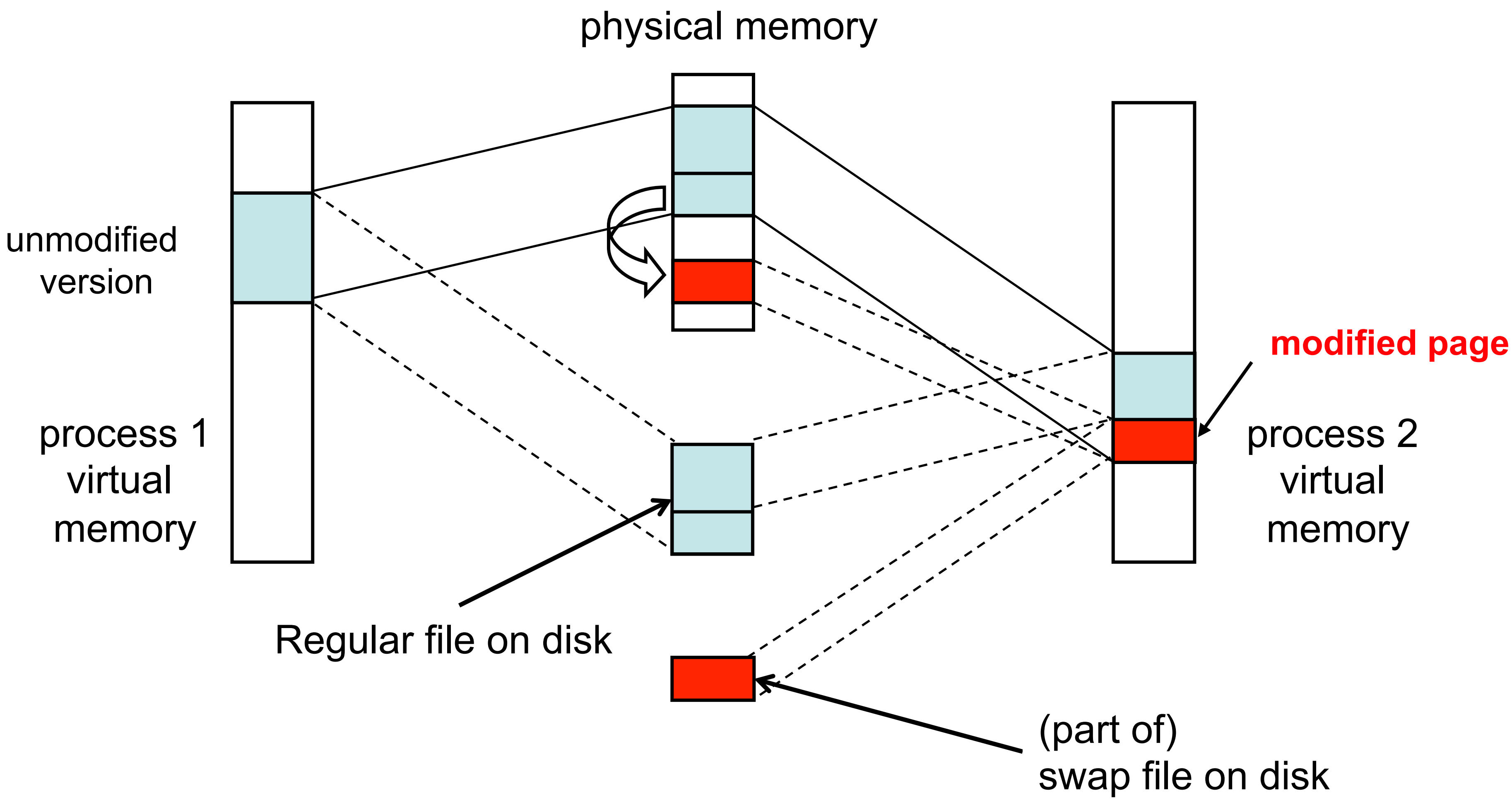
- Different levels of sharing/visibility
  - **Shared mapping**
    - Single copy in physical memory
    - Several processes can share it
    - Updates from a given process are visible by the other processes with the shared mapping
    - Updates are propagated to the backing regular file
  - **Private mapping**
    - Initially, only a single copy in memory
    - When a page is modified, a new page is allocated to store the new version
    - Updates from a given process are not visible by the other processes (with a shared or a private mapping)
    - Updates are <u>not</u> propagated to the backing regular file

Memory-mapped file
Shared mapping



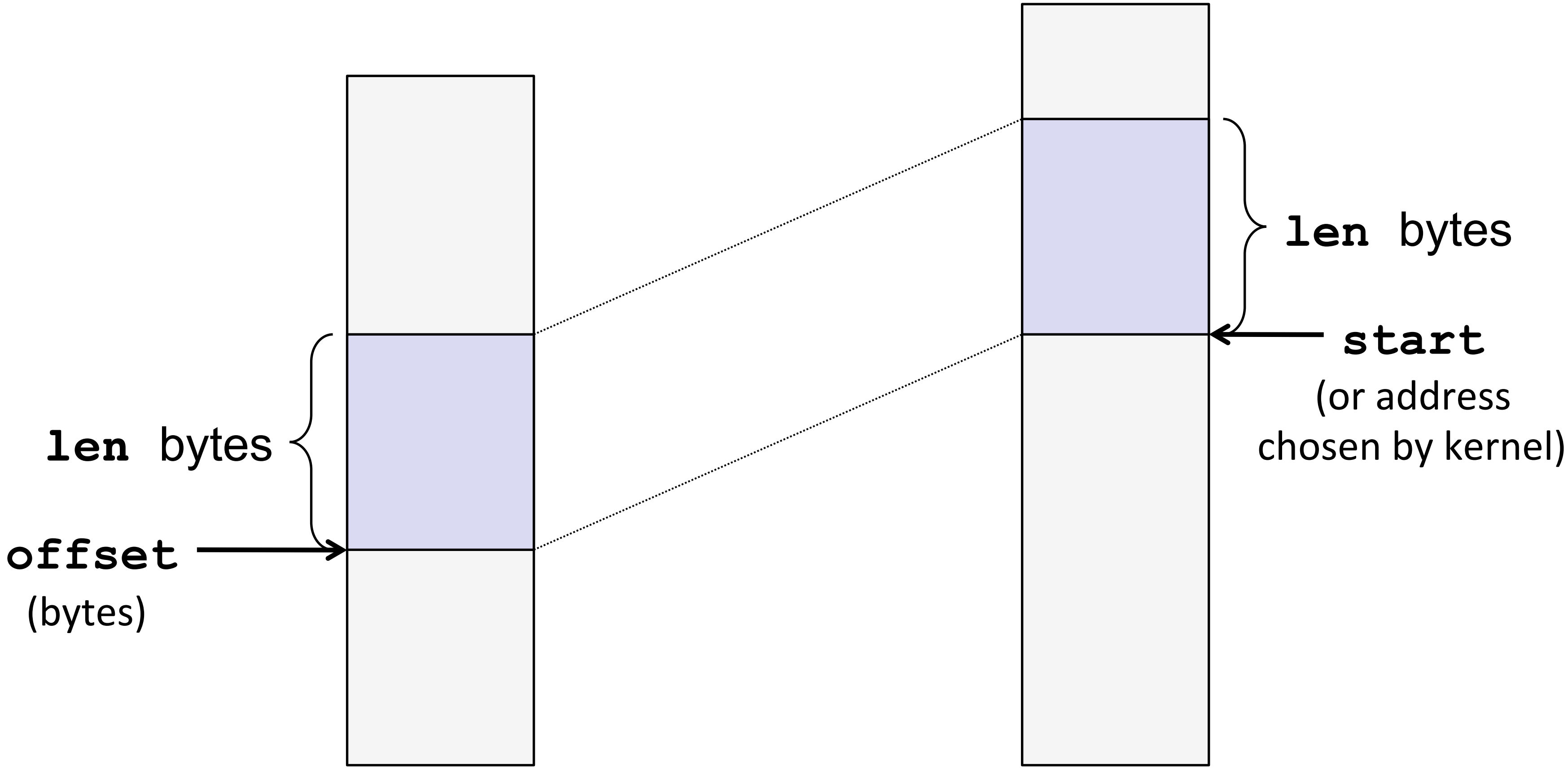
- Notice that different processes can map the file at different addresses

Memory-mapped file
Private mapping



The mmap system call

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Disk file specified by file descriptor **fd**

Process virtual memory

The mmap system call (continued)

```
void *mmap(void *start, int len, int prot, int flags,
           int fd, int offset)
```

- return value: starting address of mapping (**`NULL`** if error)
- **`fd`**: open file descriptor corresponding to the file to be mapped
- **`start`**: hint for the starting address of the mapping
  - The kernel may choose a different address
  - Typically set to **`NULL`**, to let kernel choose address
- **`len`**: size of the mapping (in bytes)
- **`offset`**: offset relative to the start of the file (in bytes)
- **`prot`**: protection rights (for whole mapped region):
  - **`PROT_READ`**, **`PROT_WRITE`**, **`PROT_EXEC`**, **`PROT_NONE`**
  - Can combine several rights using bitwise OR (e.g., **`PROT_READ | PROT_WRITE`**)
- **`flags`**:
  - **`MAP_PRIVATE`**: private mapping
  - **`MAP_SHARED`**: shared mapping
  - **`MAP_ANONYMOUS`**: anonymous memory (**`fd`** should be -1), i.e. “demand-zero” mapping
    - Option that can be combined (bitwise OR) with either **`MAP_PRIVATE`** or **`MAP_SHARED`**

The mmap system call

Purposes of the various types of memory mappings

Visibility of modifications	*Mapping type*	
	File	**Anonymous**
Private	Initializing memory from contents of file	Memory allocation
Shared	Sharing data between processes or Memory-mapped file I/O (accessing a file without explicit **`read`**/ **`write`** calls)	Sharing memory between processes (of the same family)

The mmap system call

Purposes of the various types of memory mappings (cont.)

- **Private-file**: initializing memory from contents of file
  - Example: program/library data (global static variables)
    - Modifications must not be visible from other processes (each process has its own copy)
- **Private-anonymous**
  - Used to allocate new, zero-filled memory region, with private modifications (e.g., memory heap)
- **Shared-file**
  - Memory mapped I/O: e.g., reading and (persistently) modifying a file without having to explicitly use `read`/`write`/`fread`/`fwrite` ...
  - (Persistent) shared buffer for data exchange between (arbitrary) processes
- **Shared-anonymous**
  - (Non persistent) shared buffer for data exchange between related processes (e.g., parent-child) – such a mapping can only be transmitted via “family inheritance” (through `fork`)

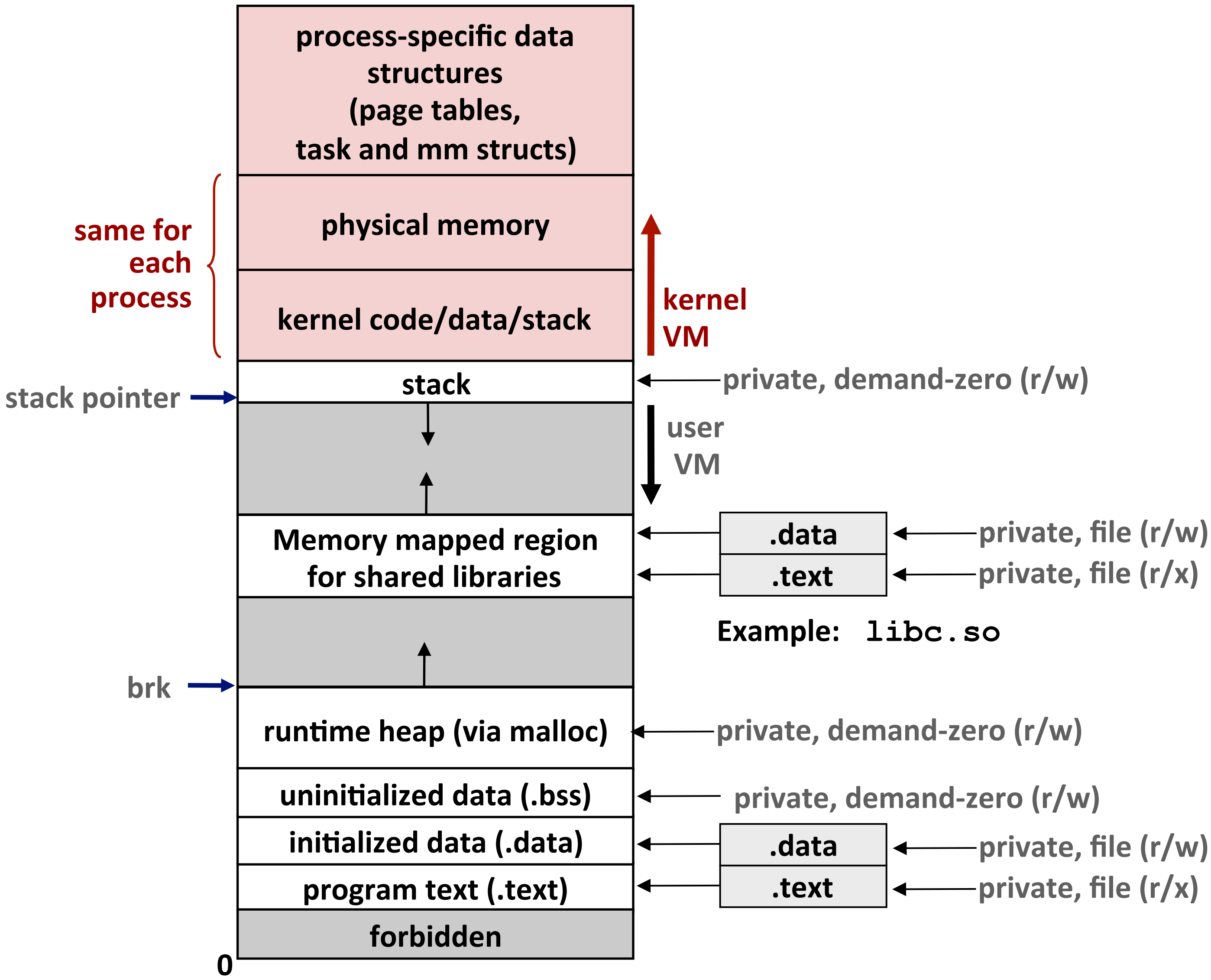
The mmap system call
Details on swapping

- **What happens when a dirty page within a memory mapped region must be swapped out (to disk)?**

- **The location on disk depends on the type of mapping**
  - **File-shared**: update the corresponding (regular) file
  - **File-private**: store the modified page in the swap file
  - **Anonymous-shared**: store the modified page in the swap file
  - **Anonymous-private**: store the modified page in the swap file

- Note:
  - The size of the swap file (on disk) + the total size of the physical memory provide an upper bound on the maximum (global) amount of virtual memory that can be allocated by the OS
  - The swap file is stored on disk (and is thus persistent) but its contents are discarded upon each reboot

Address space initialization via memory mappings



Picture from: Bryant & O'Hallaron, *Computer systems: a programmer's perspective* / CMU lectures

Examples of user-level memory mapping

- Fast/simple file copy:

```
int main() {
  struct stat stat;
  int i, fd, size;
  char *bufp;

  /* open the file & get its size*/
  fd = open("./input.txt", O_RDONLY);
  fstat(fd, &stat);
  size = stat.st_size;

  /* map the file to a new VM area */
  bufp = mmap(0, size, PROT_READ,
    MAP_PRIVATE, fd, 0);

  /* write the VM area to stdout */
  write(1, bufp, size);
  exit(0);
}
```

Picture from: Bryant & O'Hallaron, *Computer systems: a programmer's perspective* / CMU lectures

More VM system calls

- **`int msync(void *addr, size_t len, int flags);`**
  - Flushes changes of mmapped files to backing store
  - Ensures that updates are visible by other processes that access the file via **`read`**
- **`int munmap(void *addr, size_t len)`**
  - Destroys a virtual memory mapping
- **`int mprotect(void *addr, size_t len, int prot)`**
  - Changes protection on pages
- **`int mincore(void *addr, size_t len, char *vec)`**
  - Returns in **`vec`** which pages are present in RAM

Outline

- Systems calls related to virtual memory
- **Copy-on-Write**
- Paging in day-to-day use
- Hardware/OS paging extensions
- Exposing page faults to applications

Copy-on-write (CoW)

- **A technique that allows minimizing the (space) cost of maintaining two (or more) copies of a given data item**
- Used in many different contexts (memory, storage, ...). Here, we focus on virtual memory.

- **Example: CoW is used to efficiently manage private memory mappings.** General principle:
  - Initially, keep a single copy of the pages of the memory-mapped region. Configure all the pages as read-only.
  - A write access to such a page will trigger a protection fault.
  - In the trap handler: notice that the trap was caused by CoW semantics, allocate a new frame, copy the original page into it and remap the corresponding page (for the process that issued the write instruction)
  - Restart the instruction that caused the write access (like in the case of a "normal" page fault)

Outline

- Systems calls related to virtual memory
- Copy-on-Write
- **Paging in day-to-day use**
- Hardware/OS paging extensions
- Exposing page faults to applications

Paging in day-to-day use

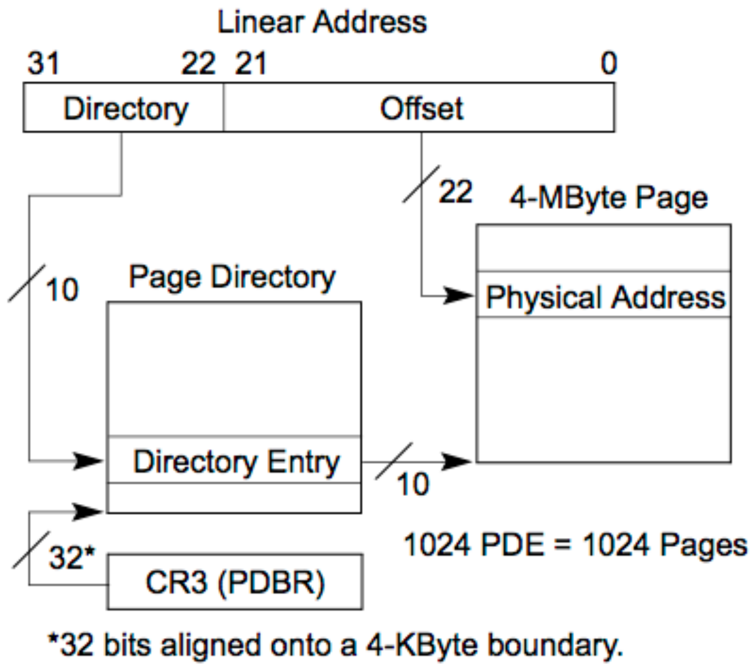
- Demand paging
- BSS page allocation
- Shared text
- Shared libraries
- Shared memory
- Copy-on-write (mmap, fork, etc.)
- Growing the stack

Outline

- Systems calls related to virtual memory
- Copy-on-Write
- Paging in day-to-day use
- **Hardware/OS paging extensions**
- Exposing page faults to applications

x86 paging extensions

- PSE: Page size extensions
  - Setting bit 7 in a PDE makes a 4MB translation (no page table)
  - Note that 4kB pages can coexist with 4MB pages
  - (more details later – see discussion about “superpages”)



x86 paging extensions (continued)

- PAE: Physical address extensions
  - Newer 64-bit PTE format allows 36 bits of physical address
  - (But virtual addresses are still 32-bit long)

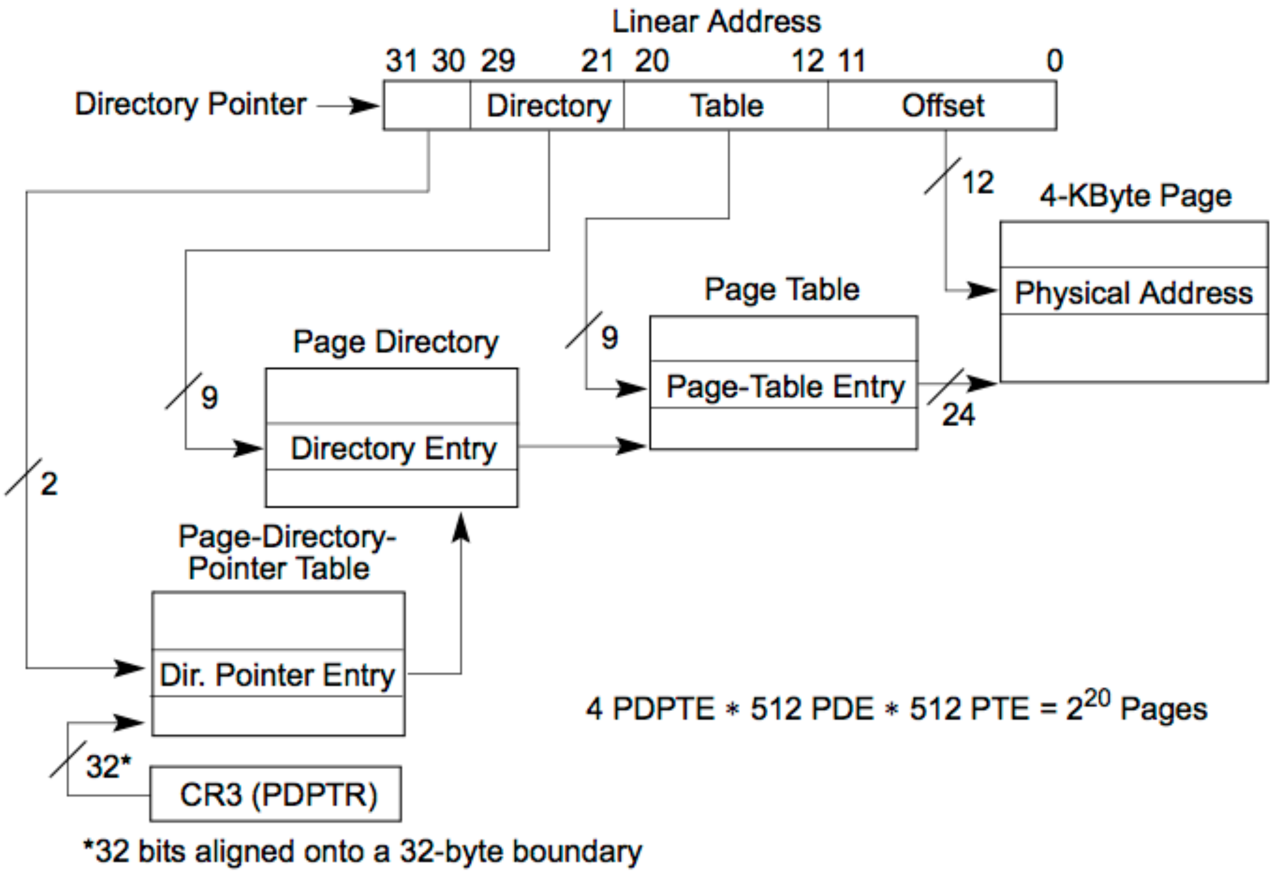
  - Page directories and page tables have only 512 entries
    - Each entry is stored on 64 bits
    - The size of a page directory or page table is still 4 kB

  - CR3 register now points to “page directory pointer table”, which contains pointers to 4 page directories
    - This allows regaining 2 lost bits

  - PDE bit 7 allows (optional) 2MB translation: same principle as PSE but with smaller page size – since there are only 21 remaining bits for the offset (compared to 22 bits with “basic PSE”)

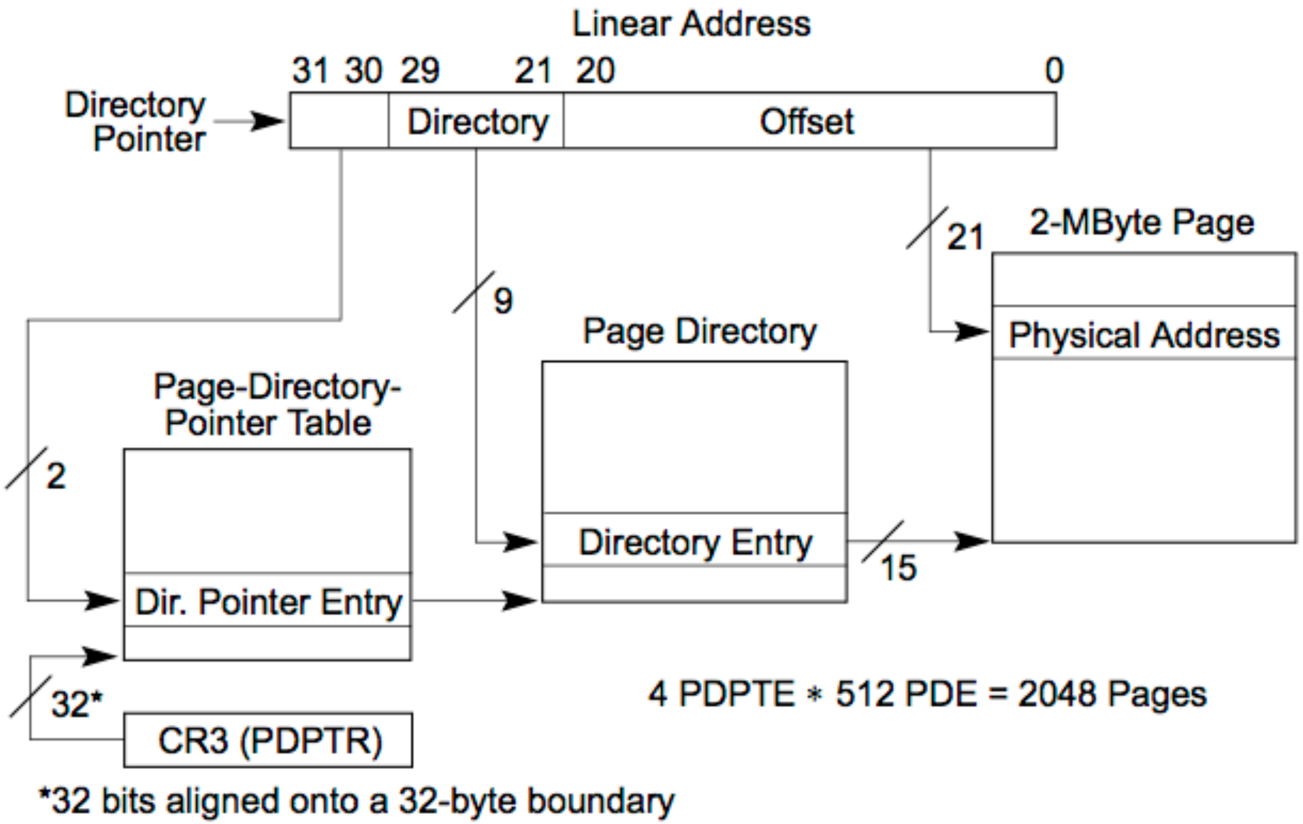
x86 paging extensions (continued)

PAE with 4-kB pages



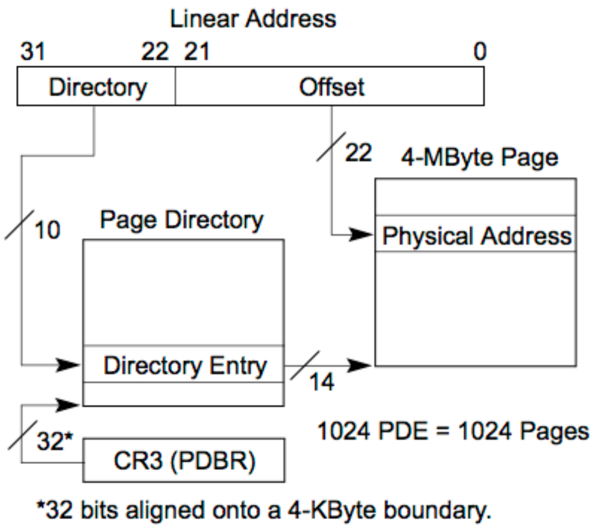
x86 paging extensions (continued)

PAE with 2-MB pages



x86 paging extensions (continued)

- PSE-36
  - An alternative to PAE
  - Uses the Page Size Extension (PSE) mode and a modified page directory to map 4MB (virtual) pages into a 64GB physical address space
  - Advantages (over PAE): hierarchy of page tables not modified, entries still use 32-bit format (not 64-bit)
  - Disadvantage: only large pages can be located in 64 GB of physical memory, and small pages can still be located only in the first 4 GB of physical memory



x86-64

- x86-64: a 64-bit processor architecture (an evolution of the x86 architecture)
  - With a 64-bit virtual address format
  - (Here, we focus on the operating mode named *"long mode"*. In contrast, *"legacy mode"* is for backwards compatibility with x86)

- However, current implementations:
  - Do not allow the entire address space of 2^{64} addresses to be used
  - Instead, define a mechanism for translating 48-bit virtual addresses to 48-bit physical addresses
    - Only the least significant 48 bits of a virtual address are considered
    - Bits 48 through 63 of any virtual address must be copies of bit 47
    - The current format is extensible up to 52-bit physical addresses

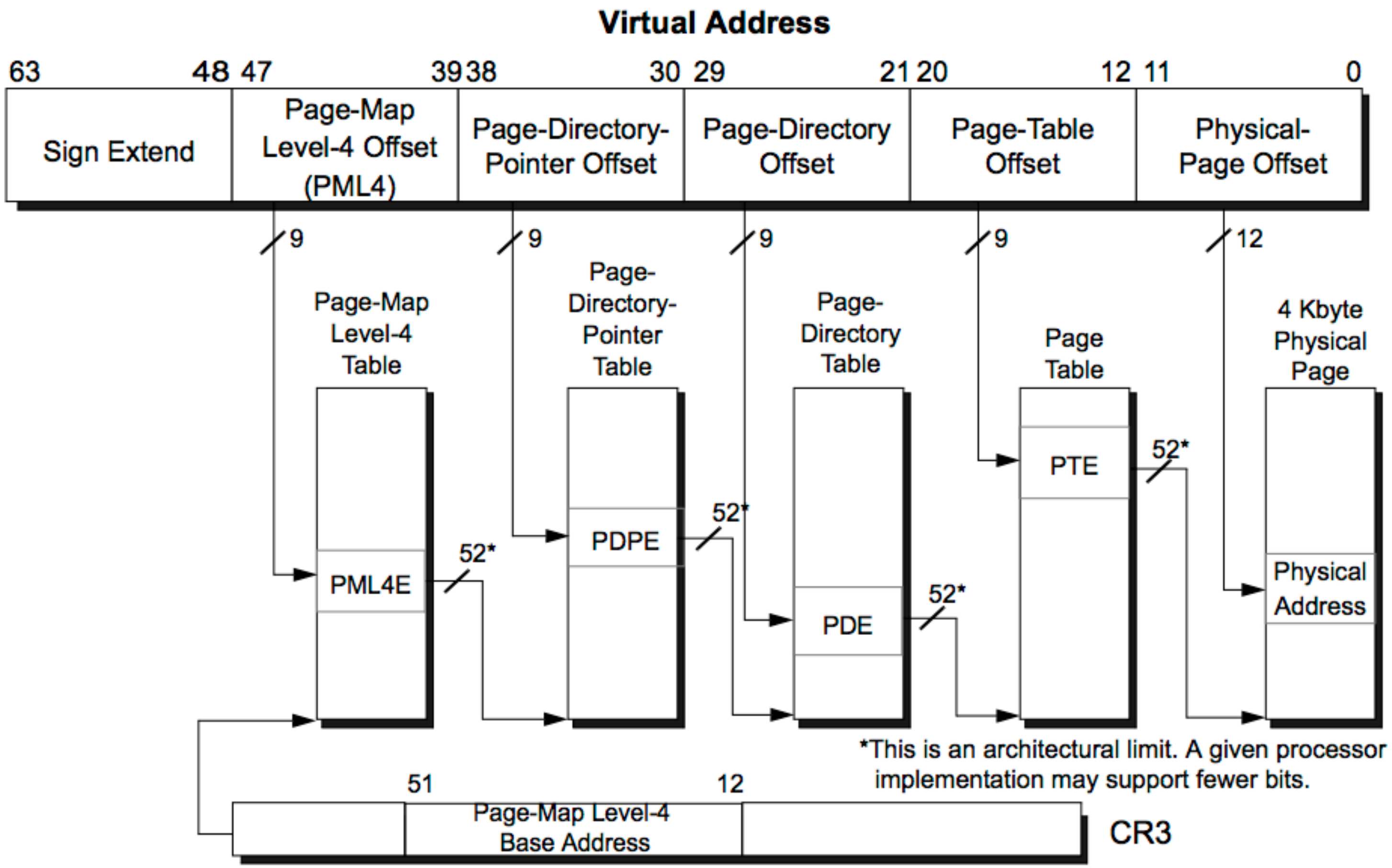
Picture from Wikipedia: https://en.wikipedia.org/wiki/X86-64

x86-64 paging

- Long mode is a superset of x86’s PAE mode
  - Page sizes can be 4 kB, 2 MB or 1 GB
  - 4-level page table (unlike PAE, which has 3 levels)
    - Page directory pointer table is extended from 4 entries to 512
    - A new level is introduced: Page Map Level 4 (PML4)
      - Contains 512 entries in implementations with 48-bit virtual addresses

x86-64 paging (continued)

4kB page translation – long mode



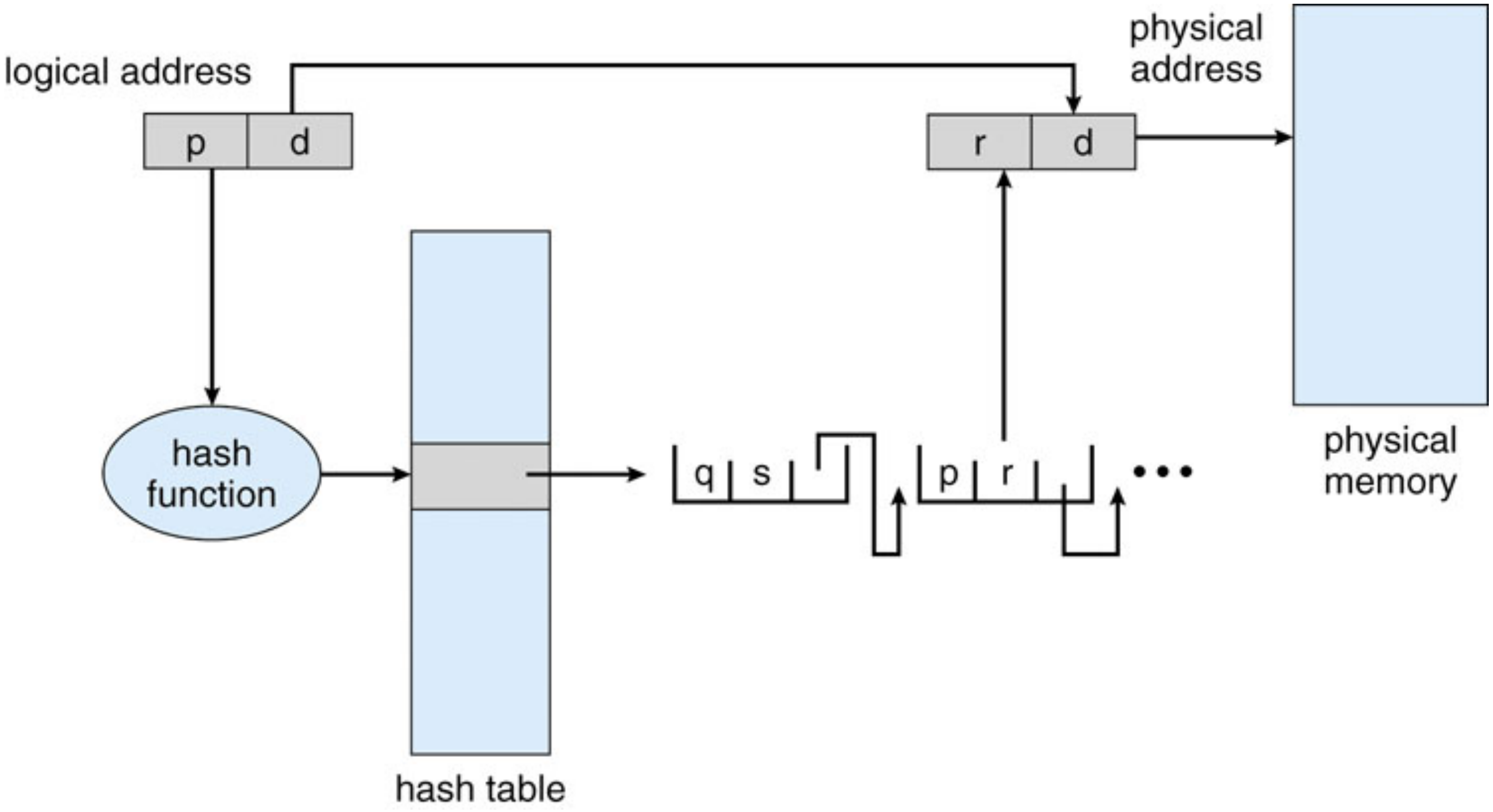
64-bit address spaces

- x86-64 has currently only 48-bit virtual address space
- **What if you want a 64-bit virtual address space?**
  - **Straight hierarchical page tables not efficient (esp. not space efficient)**
  - We will study two other approaches: *hashed page tables* and *inverted page tables*

- **Hashed page table**
  - Hash value: virtual page number
  - Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions)
  - Each element in a list contains 3 fields: (1) virtual page number, (2) physical page frame (+ details such as protection information), (3) pointer to next element in linked list
  - Variant: clustered page tables
    - Similar to hashed page table except that each element refers to several consecutive pages (e.g., 16) rather than a single page

64-bit address spaces (continued)

Hashed page table:



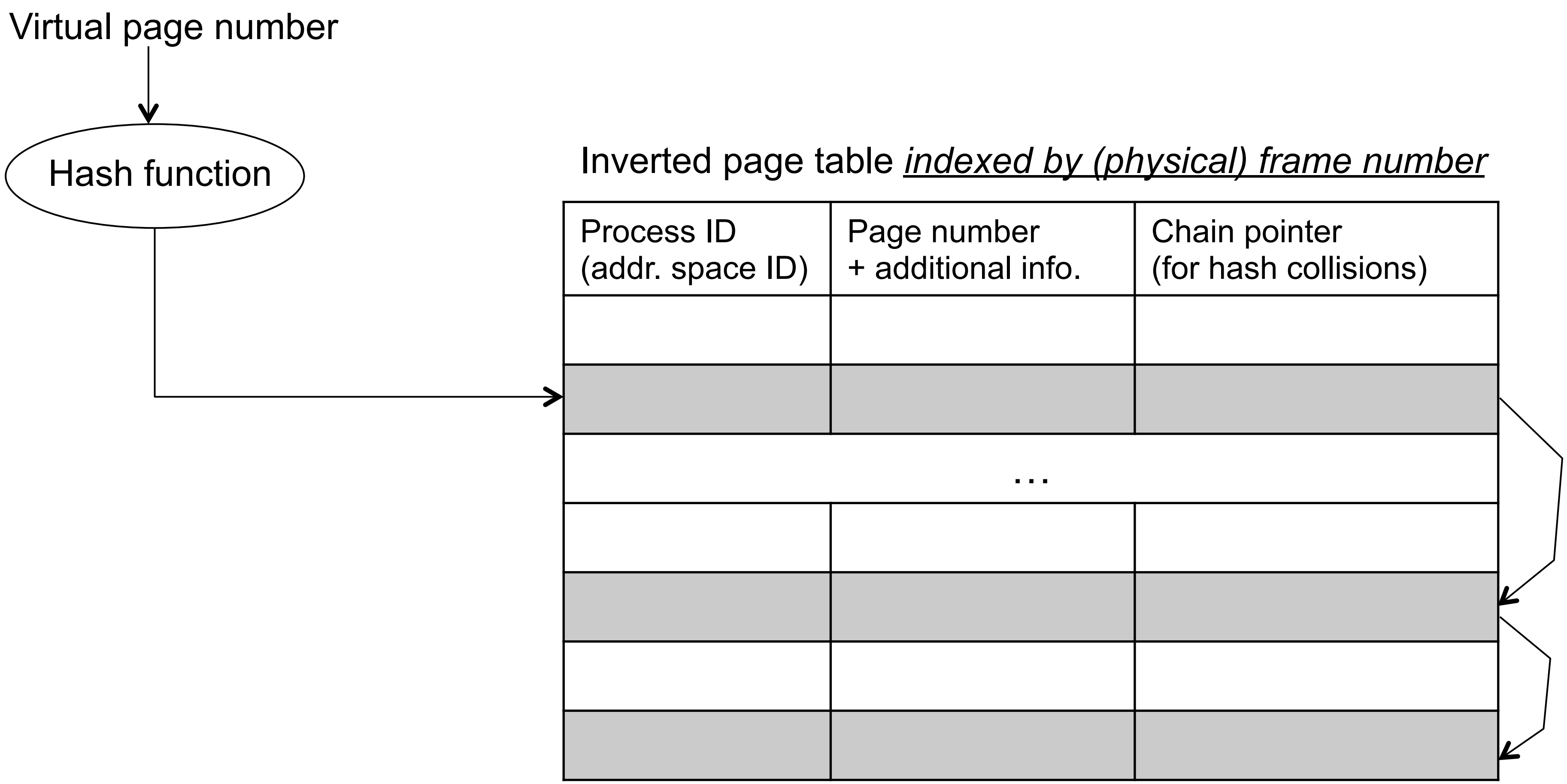
Picture from: Silberschatz et al., *Operating systems concepts* (8th edition)

64-bit address spaces (continued)

- **Inverted page table**
  - Examples: 64-bit UltraSPARC and PowerPC architectures
  - In the previous designs that we have studied, each process (address space) has an associated page table
  - In contrast, **an inverted page table design uses only a single page table for the whole system**
  - **One entry for each physical frame**
  - Each entry contains:
    - Corresponding virtual page number (+ details such as protection information)
    - Information about the process that owns the page
- **Issues with inverted page tables**
  - A lookup is costly (may require whole table scan) => Use a hash table (mapping a virtual page number to an index in the inverted page table)
  - Longer worst-case access time than hierarchical page tables
  - Sharing physical memory between address spaces is more difficult to implement

64-bit address spaces (continued)

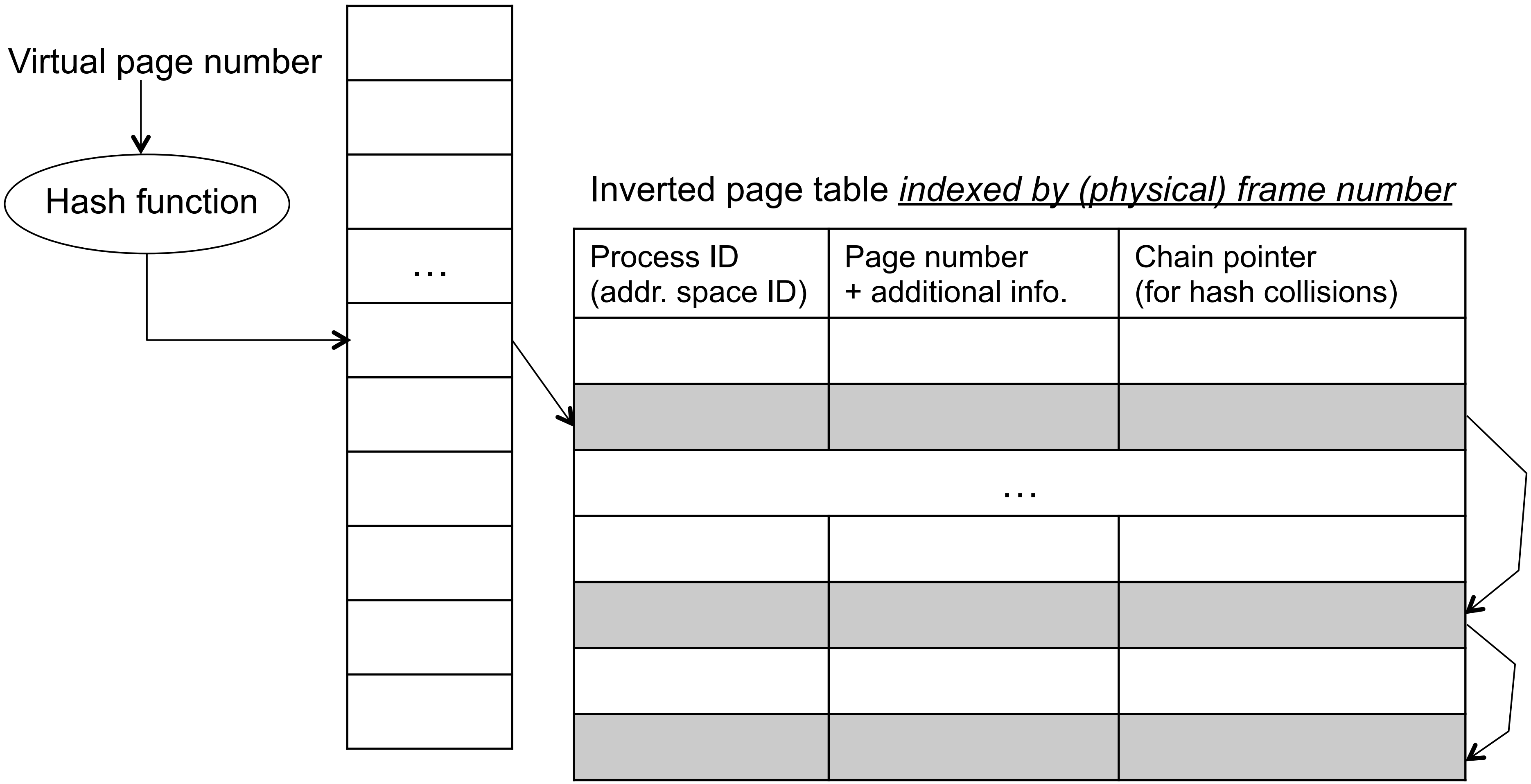
Hashed inverted page table



For more details, see:
Bruce Jacob and Trevor Mudge. Virtual memory: issues of implementation. IEEE Computer, June 1998.

64-bit address spaces (continued)

Hashed inverted page table with hash anchor table



For more details, see:
Bruce Jacob and Trevor Mudge. Virtual memory: issues of implementation. IEEE Computer, June 1998.

Superpages

- How should the OS make use of “large” mappings?
  - E.g., x86 has 2/4 MB pages that might be useful
  - Some other processors have even more choices (e.g., Alpha: 8 kB, 64 kB, 512 kB, 4 MB)

- Sometimes, more pages in L2 cache than TLB entries
  - Try to avoid costly TLB misses going to main memory
- Or have two-level TLBs
  - Try to maximize hit rate in faster L1 TLB

- The OS can transparently support “superpages” [Navarro]
  - “Reserve” appropriate physical pages if possible
  - Promote contiguous pages to superpages
  - Does complicate evicting (esp. dirty pages) – demote

- The OS can also export an interface to applications, allowing them to explicitly request large pages

Outline

- Systems calls related to virtual memory
- Copy-on-Write
- Paging in day-to-day use
- Hardware/OS paging extensions
- **Exposing page faults to applications**

Exposing page faults to applications (1/2)

- Any invalid memory access requested by the application triggers a hardware trap
  - Any access to an invalid page (no mapping defined)
  - Write access to a read-only page
  - Attempt to execute code stored in a page defined as “non-executable”

Exposing page faults to applications (2/2)

- The code of the trap handler for invalid memory accesses is registered by the OS kernel
- (On a Unix system) the kernel handler sends a **`SIGSEGV`** signal to the process
- By default, the **`SIGSEGV`** handler of the application simply terminates the process (+ generates optional *"core dump"* file with debugging information)
  - When the invalid memory access is due to a bug in the application, there is usually no other choice
  - But this mechanism can be also used by application programmers to implement advanced memory management at the application level (see next slides for details)

Virtual-memory tricks at user level (1/2)

- **General idea**: allow application to detect and trigger execution of specific procedure when the application attempts to access some memory addresses

- Useful for many different purposes, such as:
  - **Application level strategies for paging to disk**
    - Example: Big object-oriented application (e.g., database)
      - Manages main memory as a cache for much larger on-disk state
      - Can make more informed page-replacement decisions than general purpose OS kernel
      - Bring in objects on-demand (and must keep track of dirty objects)
  - **Concurrent services** (running concurrently with respect to the "regular" application code)
    - Examples: concurrent garbage collector, concurrent checkpointing
    - Need to keep track of the pages that are concurrently modified by the application

Virtual-memory tricks at user level (2/2)

- **General approach (implementation):**
  - Application registers specific handler for **`SIGSEGV`** signal
  - Application uses specific syscall (**`mprotect`**) to restrict the accessibility of the user-level page(s) that must be monitored
    - Most common example: set R/W page to read-only
  - Next access to the page triggers invocation of the **`SIGSEGV`** handler provided by application
  - **`SIGSEGV`** handler goes through the following steps:
    - Identify faulting address
    - Perform some (application-specific and address specific) action
    - Remove accessibility restriction on the faulting page (using **`mprotect`**)
  - Completion of **`SIGSEGV`** triggers <u>re-execution</u> of instruction that faulted (successful this time)
  - (For unwanted faults, **`SIGSEGV`** handler still triggers termination of process)

- For more details (on motivation, use cases and implementation), see [Appel and Li]

References

- Bruce Jacob and Trevor Mudge. Virtual memory: issues of implementation. IEEE Computer, June 1998.
- AMD and Intel technical documentations (cf. references from previous lectures)
- Juan Navarro et al. Practical, transparent operating system support for superpages. Proceedings of the 5th Symposium on Operating System Design and implementation (OSDI), 2002.
- Andrew Appel and Kai Li, Virtual memory primitives for user programs. Proceedings of the ASPLOS conference, 1991.