MODULE 16 and 17 : Memory Management, Virtual Memory and File Systems

What is Memory Management?

From an operating systems point of view, memory management happens at multiple levels:

- Hardware-assisted – Memory management unit (MMU) assisted address space segmentation and paging
- Software-assisted – Operating Systems management of memory allocations and freeing of virtual memory (kernel and user space) that has underlying MMU-mappings to physical addresses
- Software-assisted (Swap) – Operating Systems management of swapping from disk and corresponding freeing up of physical pages in RAM

Address Spaces

Let us look at hardware-assisted memory management first:

At a basic level, address spaces can be categorised as:

- Kernel/interrupt-level
- Process/user-level

Further to this, there is address spaces separation across processes

In addition, there are Hypervisor-assisted MMU management and IO-assisted MMU configurations - we won't look at these right now

Address Spaces – Terminology Explained

- Logical address = compiler generated address
- Virtual address = address as per segmentation unit (sometimes referred to as linear address)
- Physical address = actual hardware address listed in the Page Table Entry (PTE) used to access the physical memory address via the memory controller
- The mapping in the MMU (combination of segmentation + paging) = Logical → Virtual (linear) → Physical (actual)

Address Spaces Side Effects Summary

- On PowerPC (RISC), the segment register based separation
- Segment register content specific to process, and hence part of process context
- Lightweight process – on systems with no swapping, everything is fairly light weight on RISC these days – why?
  - ✓ Physical caches (virtually indexed, physically tagged) – no cache flushes. Tagged TLB's – no flushing of TLBs
- Logical caches, physical caches

Address Spaces Side Effects Summary (contd.)

- On Intel x86, there is the overhead though of flushing TLB’s on context switches as TLB’s are not tagged
- Fork is a bit more time intensive than thread creation – why?
- All the why’s will be answered in the following slides!

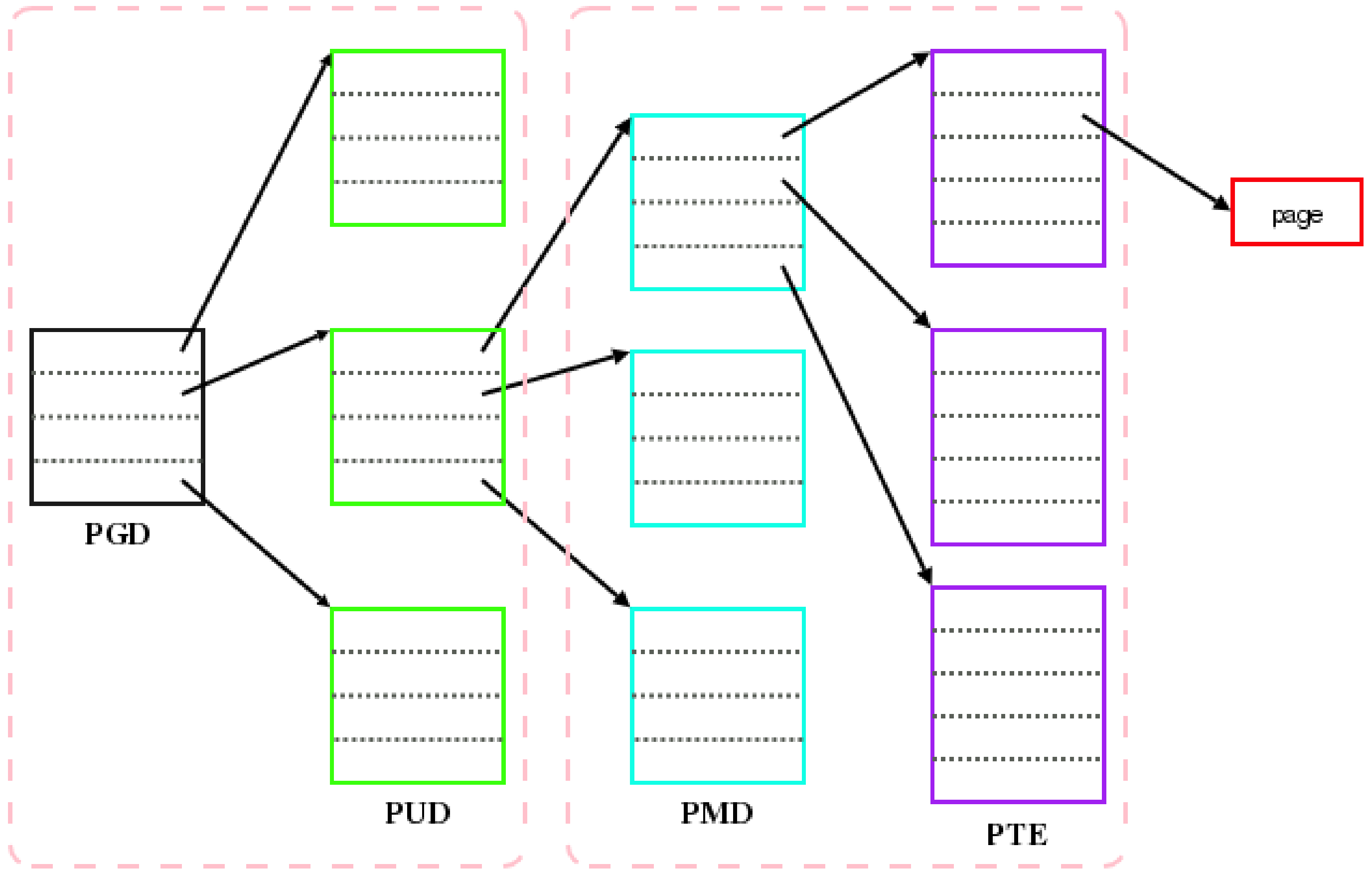
Kernel Memory Management Hardware – The Processor Interface

MMU internals – PPC and Intel x86 case studies

Linux interworking with Intel MMU

- Multi-level (tree-based) page tables
- Hardware TLB miss-handling
- Linux uses a tree-based, hierarchical page table – this fits in with a lot of processor architectures including x86, and saves space for sparsely filled page tables (as compared to a single-level linear array)
- Names of the multiple levels in software are a bit confusing and don’t correspond to the x86 hardware naming convention
- PGD = highest-level directory, PUD = page upper directory (only present for x86-64, collapses for 32-bit and PAE), PMD = page middle-level directory (collapses for 32-bit, but present for PAE), PTE = page table entry
- Collapse means pud_offset(pgd, address) would for instance return pgd because that’s what the ifdef for PAE will compile to

Linux interworking with Intel MMU (contd.) – full 4-level page table (x86_64)



Copy-on-write

- Copy-on-write is a neat little trick implemented on Unix-based systems like Linux to save on space and time when a process does a fork.
- A fork is intended to clone a process (including the main thread)
- A fork is in a lot of places followed by an exec of a new binary however, and so fork-exec is a standard OS execution pattern to start the execution of a new application
- As this is the case, OS designers decide to use a processor MMU architecture setting to avoid potentially unnecessary copying of memory, in this case, the data sections, heap etc. of the original process

Copy-on-write(contd.)

- What is done on the fork is therefore: A copy of the original process's page tables are made for the newly forked process to use.
- A different CR3 (refer to PTE slides 20, 21) is used so that a copy is done of the PTE entries of the original process into the page tables of the new process
- And, the new mappings (same contents as the original process PTEs) in the forked process mark the physical pages as read-only so that when a write happens to any of the physical pages from this process's code, a trap happens and inside the trap handler, a new physical page is allocated and its PTE is updated accordingly

Dirty Pages

- A PTE can be marked as dirty.
- This facility is used when file writes happen to files that have contents loaded from disk into the page cache in RAM. When writes happen, the PTE’s that correspond to pages in the page cache are marked dirty.
- These dirty pages are periodically, asynchronously (with respect to the file writes to the page cache) flushed to their original locations on disk.
- Only once this is done, can those page cache pages become potential spots to shrink the page cache, in low memory scenarios

Memory Allocation – Bird's Eye View

- Note: A lot of this content has a Linux bias, although the user-libraries and naming has a Linux flavour to it, similar ideas hold true for just about every OS.

- What you get:

  Malloc : Contiguous logical within user space, non-contiguous physical (usage: normal user space allocation)

  Kmalloc : Contiguous logical and physical (usage: kernel memory allocation, DMA areas)

  Vmalloc : Contiguous logical within kernel space, non-contiguous physical (kernel module space allocation)

- How you get it – how the user space library manages virtual memory, how kernel manages physical memory

- The additional complexity of logical→ physical address space mappings

Kernel memory allocation, page table updates – kernel interactions

Change of entry – fresh kernel space allocation

- Find free physical space – search lists of free blocks whose sizes are of order 2^(n-1) page size blocks, where n = 0 .. MAX_ORDER-1 (get_free_pages)
- Buddy, coalescing – if you can't find free blocks from within the appropriate order list (best fit) , go to the next best fit
- Split the next best fit and mark the split halves as buddies to reduce internal fragmentation
- Coalesce when freeing a buddy to reduce external fragmentation

Kernel memory allocation, page table updates – kernel interactions (contd.)

- Manipulating the page table – mark PTE as present (address mapping already set up by early kernel code, page table accessed using self-mapping trees)
- PPC – cache of the first-level of page table tree is the hash table
- No change to entry
  - **Keep a group of pre-allocated, content-initialized pages, use the slab allocator to point new data structures to pre-existing slab caches**
    - ✓ Performance factor of not having to allocate and initialize
    - ✓ Used generally for relatively smaller (less than KMALLOC_MAX_CACHE_SIZE) allocations to reduce internal fragmentation
- Change of entry – Swap – covered later

The Linux Virtual Address Space - Configuration

- User virtual address space – 0x0000000 (if 32-bit, 0x08048000, if 64-bit 0x0000000000400000) – 0xBFFFFFFF (3 GB)

- Low user virtual address space is used to map in libc libraries etc.

- Kernel virtual address space – 0xC0000000 (PAGE_OFFSET) – 0xFFFFFFFF (1 GB)

- Kernel virtual address space maps to 0x00100000 (1 MB mark from physical page 0 onward) – the first 1 MB is used for things like POST results

- The first 8 MB starting at virtual address 0xC0010000 (physical address 0x00100000) is where the kernel image is located

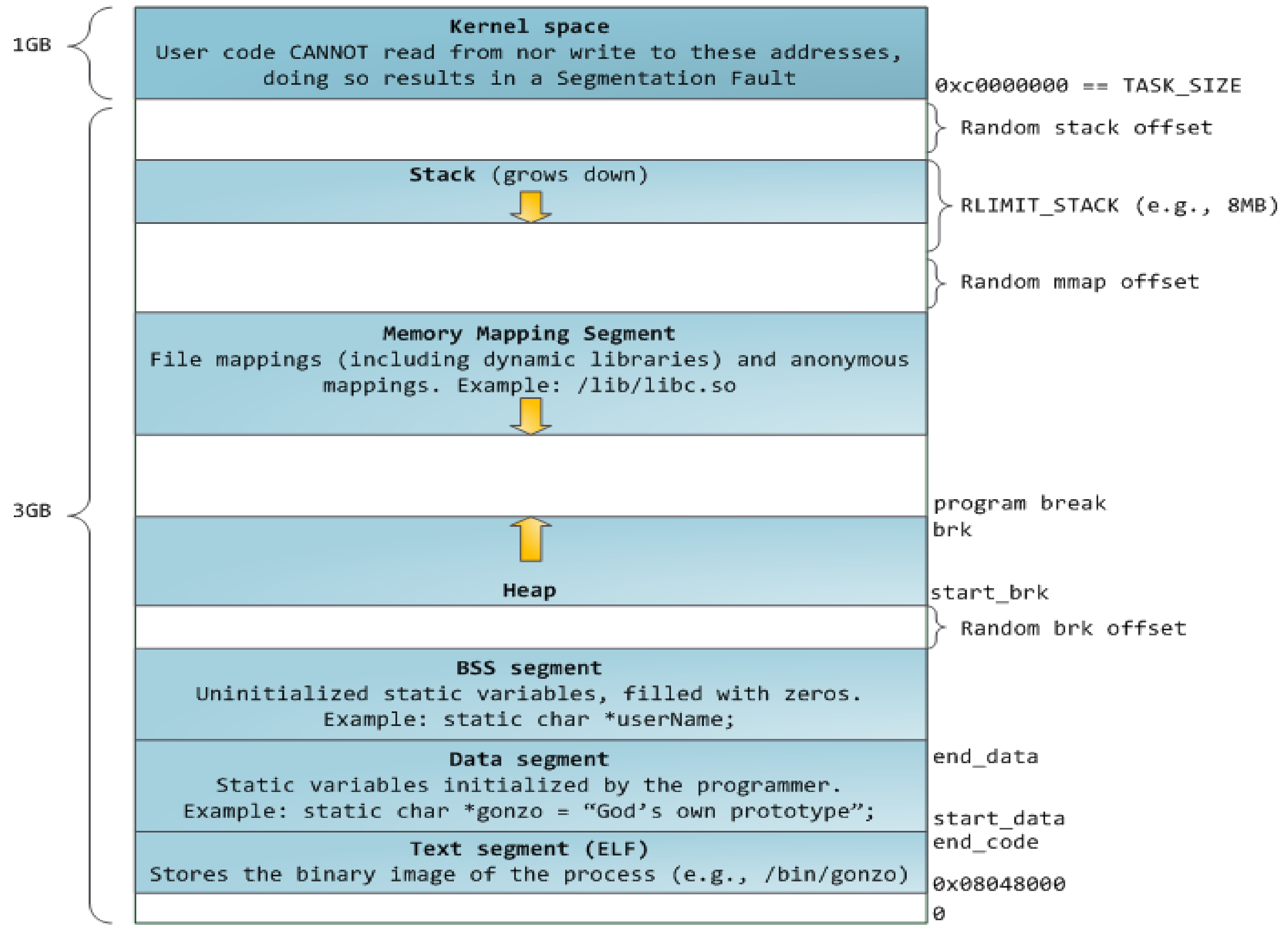
The Linux Virtual Address Space – Configuration (contd.)

- The first 16 MB (including the above 8 MB) is allocated to ZONE_DMA
- Therefore, the first virtual address that is used for kernel purposes is 0xC0010000 + 16 MB = 0xC1000000 – which is where mem_map is located
- The lower 896 MB of this area after ZONE_DMA is used for permanent kernel mapping
- The higher 128 Mb of this area after ZONE_DMA is dedicated to temporary mappings and vmalloc

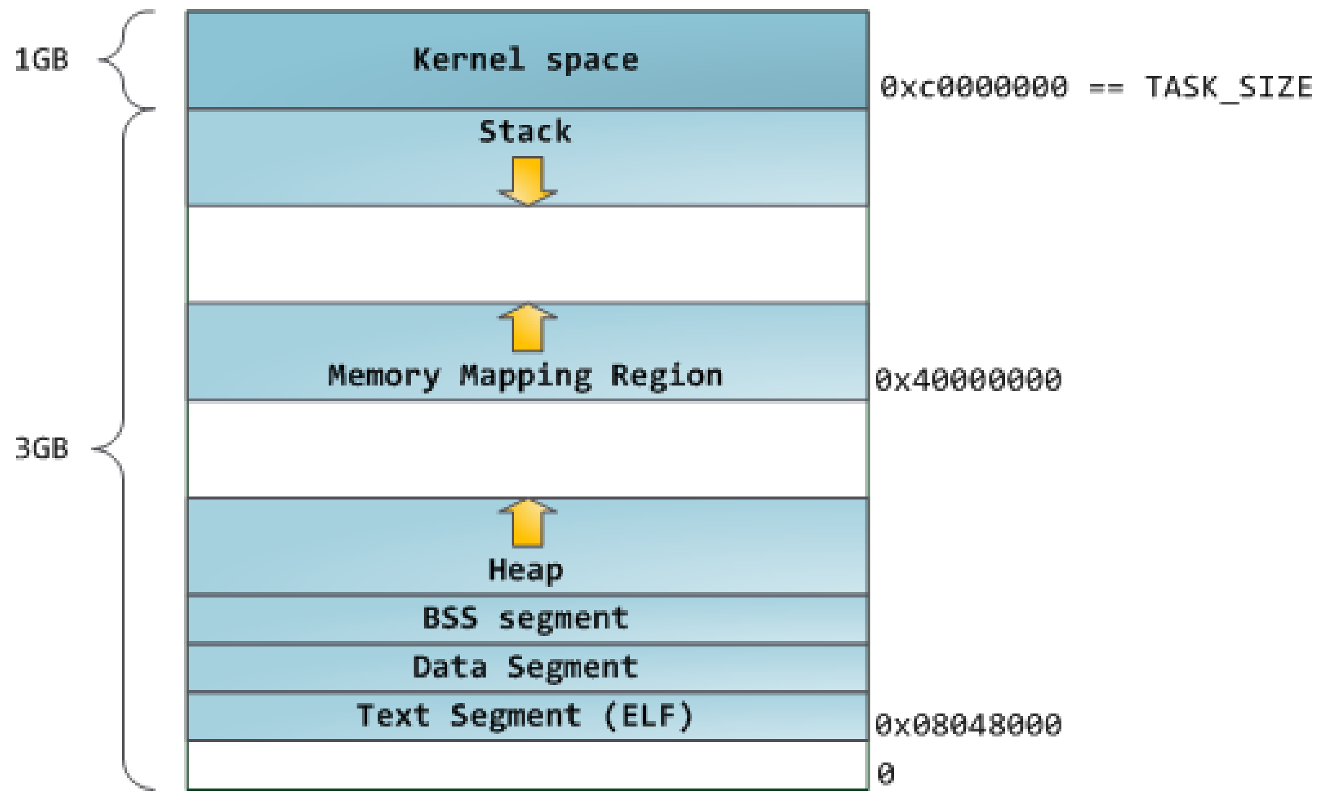
The Linux Virtual Address Space – Configuration (contd.)

- HighMem - temporarily mapped into kernel space in the upper 128 MB of the 1 GB kernel virtual memory) - use of HighMem when available kernel physical memory is 1 GB – the available virtual memory is restricted to less than 1 GB by vmalloc , mem_map and PTE requirements, hence you need some virtual space to temporarily map part of this 1 GB to parts of this HighMem-designated 128 MB. Mention PAE ?? (Without PAE no highmem??)
- Mem_map - array of struct page corresponding to every page – why we shall soon see

Process Virtual Address Space (flexible layout)


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Process Virtual Address Space (classic layout)



Advantages of the Flexible Layout

Primarily the flexible layout enables:

- Mmap grows upward from 0x40000000 in the classic layout, and down from RLIMIT_STACK + some random offset in the flexible layout
- The new layout is in essence 'self-tuning' the mmap() and malloc() limits: both malloc() and mmap() can grow until all the address space is full. With the old layout, malloc() space was limited to 900MB, mmap() space to ~2GB, as the heap had a hard upper limit of 0x40000000
- Hence, both malloc(), mmap()/shmat() users to utilize the full address space: 3GB on stock x86, 4GB on x86 4:4 or x86-64 running x86 apps.
- The new layout also allows potentially very large continuous mmap()s
- Address space randomization using a random offset is also included in the flexible layout

Sample output from cat /proc/<pid>/maps

```
cat /proc/self/maps
address                  perms   offset      dev     inode      pathname
08048000-08053000 r-xp 00000000 08:07 1048597    /bin/cat
08053000-08054000 r--p 0000a000 08:07 1048597    /bin/cat
08054000-08055000 rw-p 0000b000 08:07 1048597    /bin/cat
08651000-08672000 rw-p 00000000 00:00 0          [heap]
b734b000-b754b000 r--p 00000000 08:07 795175     /usr/lib/locale/locale-archive
b754b000-b754c000 rw-p 00000000 00:00 0
b754c000-b76f0000 r-xp 00000000 08:07 548347     /lib/i386-linux-gnu/libc-2.15.so
b76f0000-b76f2000 r--p 001a4000 08:07 548347     /lib/i386-linux-gnu/libc-2.15.so
b76f2000-b76f3000 rw-p 001a6000 08:07 548347     /lib/i386-linux-gnu/libc-2.15.so
```

Sample output from cat /proc/<pid>/maps

```
cat /proc/self/maps
b76f3000-b76f6000 rw-p 00000000 00:00 0
b770a000-b770b000 r--p 00858000 08:07 795175     /usr/lib/locale/locale-archive
b770b000-b770d000 rw-p 00000000 00:00 0
b770d000-b770e000 r-xp 00000000 00:00 0          [vdso]
b770e000-b7710000 r--p 00000000 00:00 0          [vvar]
b7710000-b7730000 r-xp 00000000 08:07 547269     /lib/i386-linux-gnu/ld-2.15.so
b7730000-b7731000 r--p 0001f000 08:07 547269     /lib/i386-linux-gnu/ld-2.15.so
b7731000-b7732000 rw-p 00020000 08:07 547269     /lib/i386-linux-gnu/ld-2.15.so
bf838000-bf859000 rw-p 00000000 00:00 0          [stack]
```

User-space memory allocation – Malloc

What happens when malloc is called from a process' context the very first time?

- You need to know where free space is within boundaries imposed by system (heap area start, and end of mmap as per flexible layout)
  - ✓ For this you need to first look at memory hierarchy: Page-level and block level
  - ✓ What is the minimum block size? – 16 bytes (8 bytes + header-footer overhead) (implementation dependent)
  - ✓ Malloc (4), malloc (8) both return 16 bytes, malloc (12), malloc (16) both return 24 bytes – multiples of 8

User-space memory allocation – Malloc (contd.)

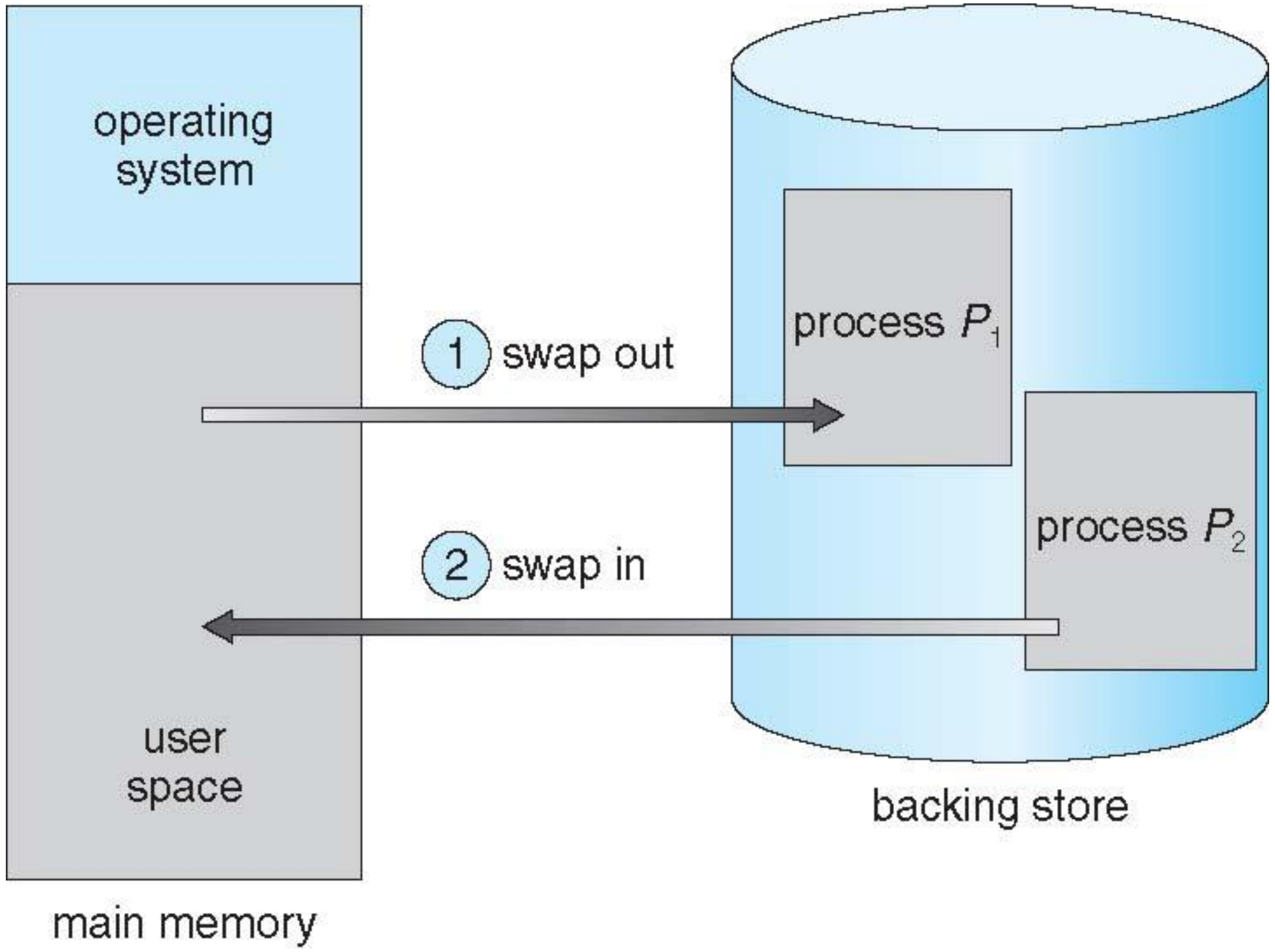
- Does malloc need to return a virtually contiguous space? Too much space overhead otherwise (next pointer etc.) and temporal locality might be lost as well, but most importantly, the compiler relies on the fact that the virtual space is contiguous.

- The physical non-contiguousness is the basis of paging virtual memory, and results in zero external fragmentation at the page level. Plus, it facilitates swap which the kernel space doesn’t need to.

Malloc (contd.)

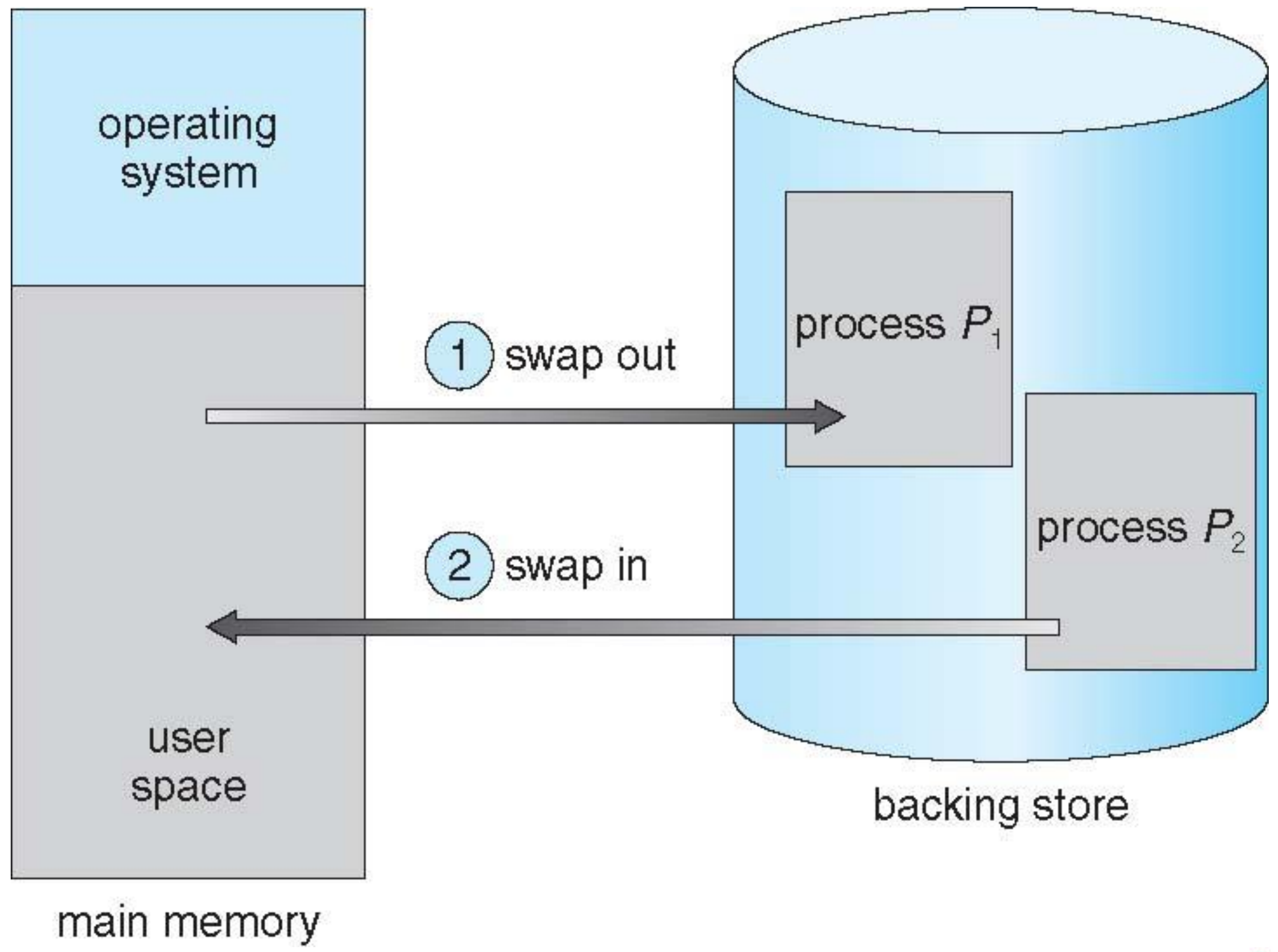
- Small size (>= 64, < 512 bytes) – best-fit with coalescing.
- Large size (>= 512, < mmap threshold (128 KB) - variable size bins with trees /optimal-for-search data structures whose nodes are sorted by size – bigger range, lesser number of bins, more searching.
- Very large (> mmap threshold 128 KB) – relies on mmap mapping facilities and free now actually returns memory to the operating system by calling munmap. The tradeoff is that more minor faults occur on subsequent big malloc's as pages are freed.
- For cache line alignment, cache-line aligned alloc routines, posix_memalign.
- Can malloc-ed memory sit astride a page boundary? No, because when you run out of user space memory, you mmap/sbrk and go into the next page completely, no piece-by-piece thing.

Swapping


operating system
user space
main memory
1 swap out
2 swap in
process P_1
process P_2
backing store

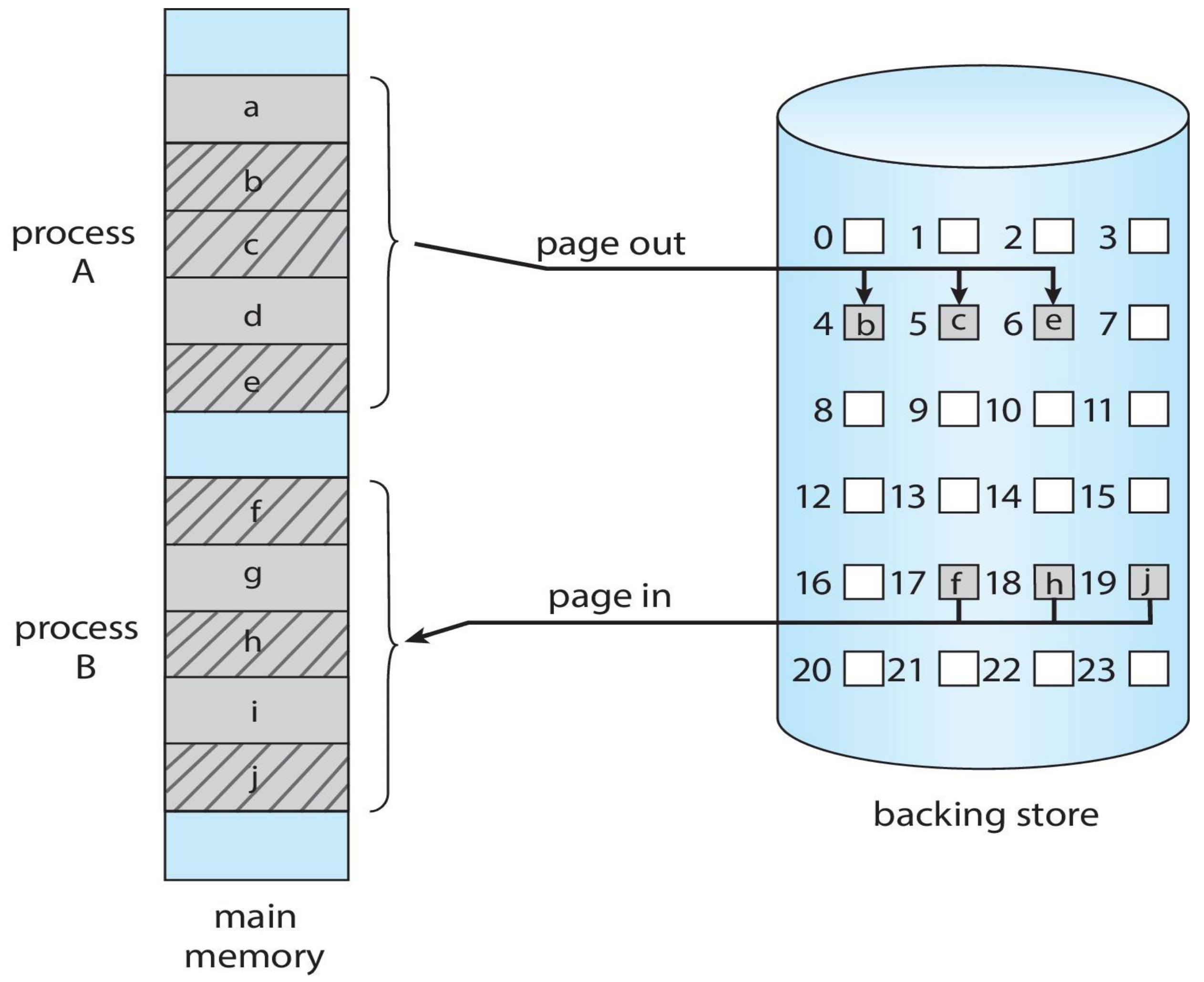
Schematic View of Swapping

What swapping used to mean originally

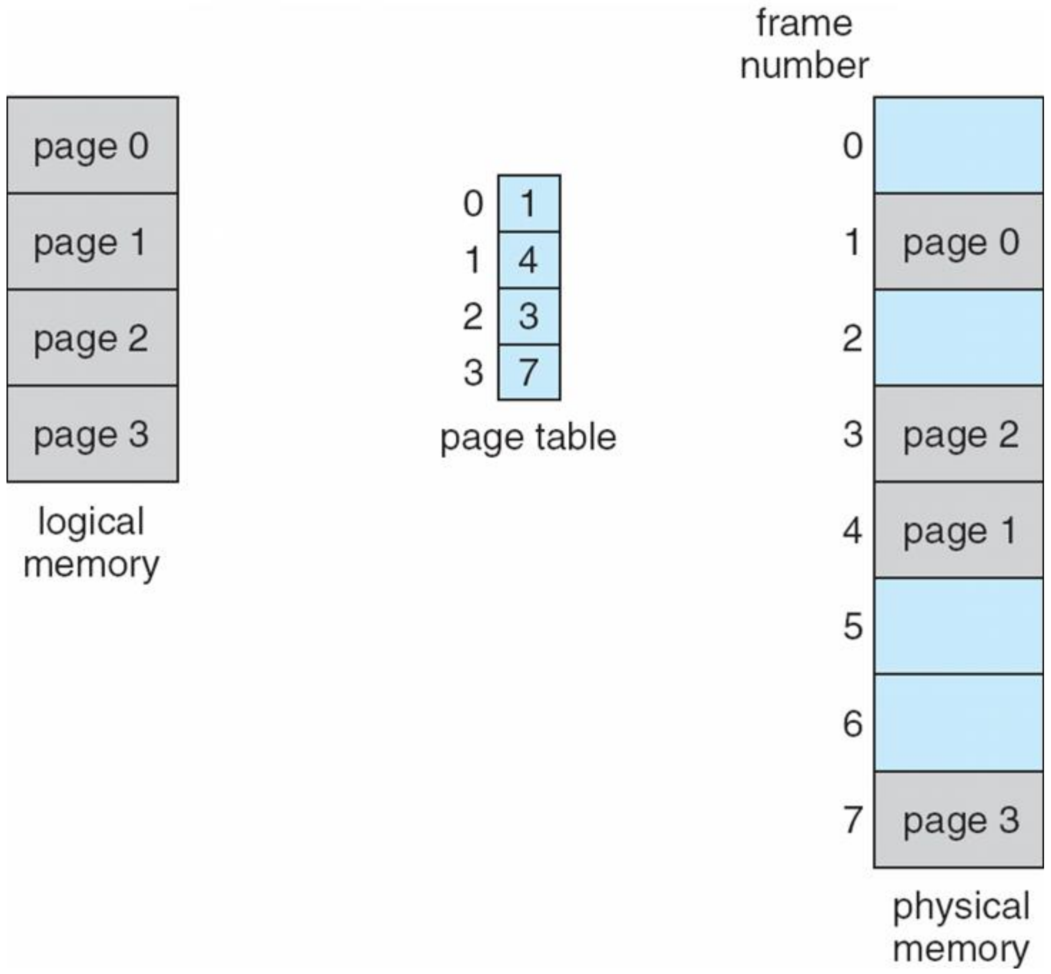


Swapping with Paging

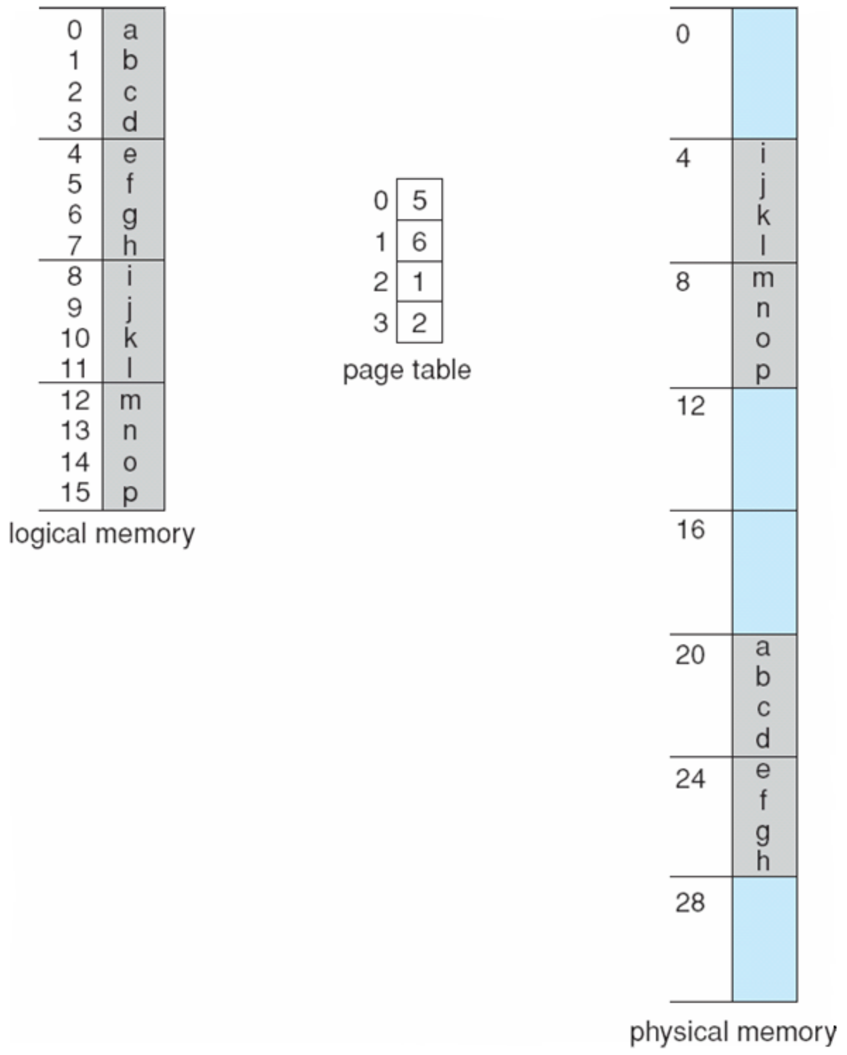
What swapping means in modern operating systems



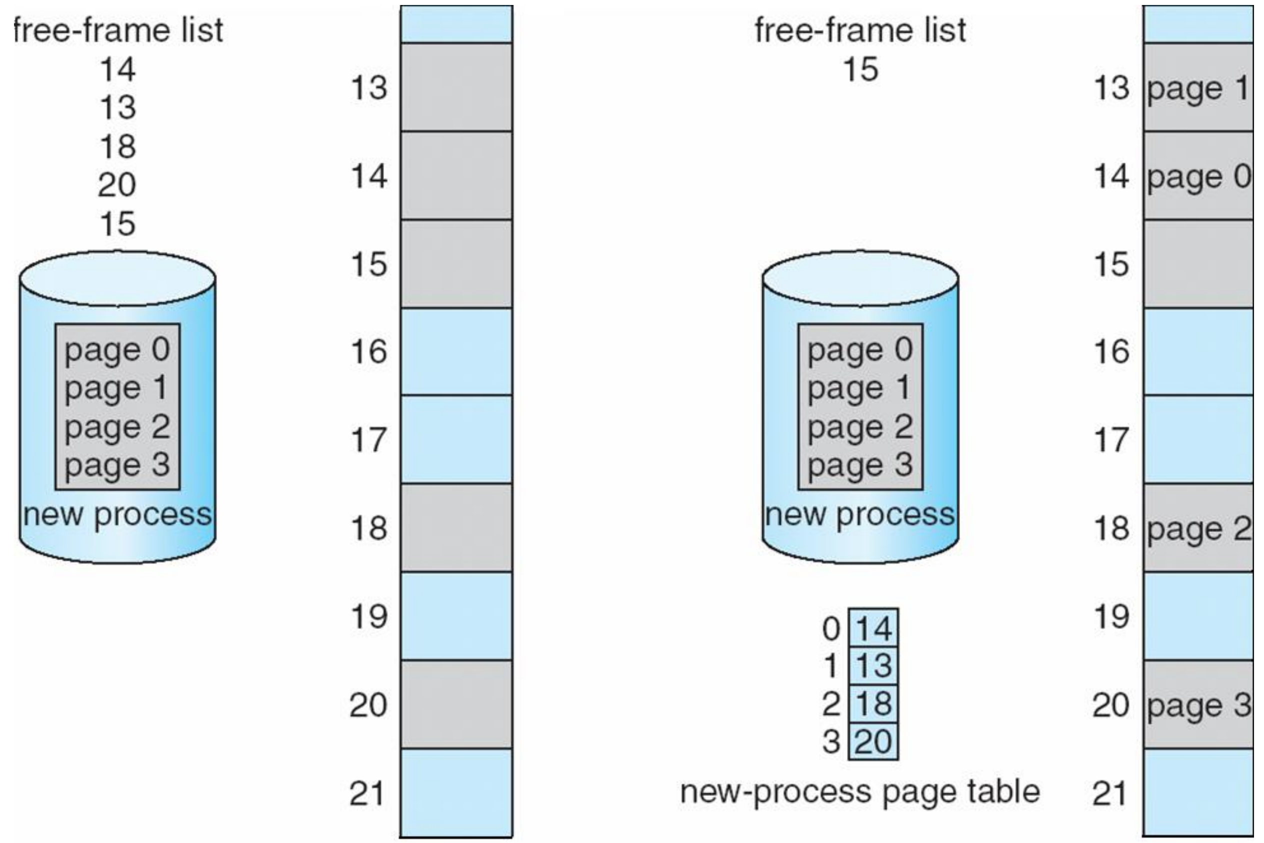
Paging model of logical and physical memory



Paging example (with values loaded in memory locations)



New process physical page allocation when there are free pages



(a) (b)

Page faults – Basic Premise

- The previous slides serve as a pictorial view of a simple swapping scheme where free frames are marked and can be used as a target a new process physical page allocation

- That's simple enough

- Now, let's think of a scenario, where there are not enough free pages to be used by a new process

Fault handling/demand paging of user-space pages

Minor fault:

- So-far unmapped page, PTE all zeroes, find virtual address in memory management kernel data struct

  - ✓ If found, then allocate space for PTE, get physical page from free list, set PTE to point the relevant virtual address to the physical page.
    - ❖ If belonging to page cache, but freshly allocated, then page in file page from disk

  - ✓ If not found, cause segmentation fault/trap and cause the process to exit

Fault handling/demand paging of user-space pages

Major fault:

- PTE found, non-zero entry, but present bit reset. Find swap index from PTE, swap page in after finding target physical slot, and swap out of target
- Swap-out involves cycling through reverse mapping to figure out set of logical pages mapped to the target
- Swap-in in involves interacting with the bio subsystem and device driver to get the page contents
- Target physical slot – LRU, accessed bit in PTE, periodically reset by software, set on access by hardware to implement LRU
- Note: Only anonymous pages get swapped out – the file page cache is flushed and shrunk, so will have zeroes in its PTE

File Systems software design – modularity

- On Linux, there is clear division of labour in File Systems implementation:
  - ✓ System call wrappers
  - ✓ Virtual File System (VFS): Page cache, flushing to disk, and associated memory support (Agnostic to actual file system)
  - ✓ Mapping layer: File system-specific mapping routines
  - ✓ Generic Block layer
  - ✓ IO schedulers
  - ✓ Block device driver: Actual file (on hardware) access routines

File Systems software design – System call wrappers

- System call wrappers
  - ✓ Information available to SC wrapper:
    - Filename, file offset, file modes
  - ✓ How does it view a file?
    - Filename with path, file offset to read from/write to
  - ✓ What does it do?
    - Provides generic wrappers to the actual Virtual File System(VFS)-implemented system call

File Systems software design – Virtual File System (VFS)

- VFS
  - ✓ Select information available to VFS:
    - Filename, offset mapped to struct file – struct file has lower-level file/directory information, per-process File Descriptor, Superblock, Inode
    - Per-process File Descriptor is an index into an array of pointers that point to a per-process file descriptor table that in turn points to a system-wide open file table. This has references to fields like the Inode field described below
    - Superblock stores information concerning a file system (that has been mounted) – this in-memory data structure corresponds to a file system control block on disk
    - Inode stores general information about a specific file – this in-memory data structure corresponds to a file control block on disk. Each inode object is associated with an inode number which is a unique identifier of the file within the file system.

File Systems software design – Virtual File System (VFS)

✓ How does it view a file and what/how does it use the page cache?

- Filename, offset is mapped to struct file – struct file has lower-level information via a per-process File Descriptor. A call to open() returns a file descriptor id that is specific to the process. This id is referenced by subsequent application calls to read/write for the same file.
- Internal to the kernel, inside the open system call handler, this id is associated with a kernel file specific data structure (struct file) that along with the file offset has associations with physical page-related structures that help with reads/writes to/from the page cache. Dirtying the page cache triggers the flush out of the physical page to disk.

File Systems software design – Mapping layer

- Mapping layer
  - ✓ Information available to mapping layer:
    - VFS file block number(offset within each file) to Logical Block Address (LBA) mapping (done at initial page cache allocation time)
    - File system specific disk inode access routines that help set up this mapping
  - ✓ How does it view a file?
    - Since this layer is a software glue between VFS and file-system specific information on the disk, it is aware of both logical block offset information (VFS) and the LBA addresses
    - It has access to actual on-disk file system control blocks and file control blocks and hence is the first layer (from a bottom-up point of view) that actually views files and filesystems as a whole complete with control and data information

File Systems software design – Mapping layer

- ✓ What does it do?
  - Accesses disk inode data structures that contain the LBA addresses (relative to disk/partition) corresponding to each file block offset (relative to each file = file offset) and sets up the VFS file block offset to LBA mapping
  - Helps populate the VFS in-memory superblocks and inodes with information from actual file-system specific on-disk file system control blocks and file control blocks
- ✓ How does it do it?
  - Function calls set up at file system initialization time

File Systems software design – Mapping layer – ext2, ext3 filesystems overview

- The Ext2 file system has the following noteworthy features:
  - ✓ Configuration option of setting up optimal block size (1024 to 4096) – whichever leads to less fragmentation based on expected average file lengths.
  - ✓ Configuration option of limiting how many inodes can be allowed for a partition of a given size depending on the expected number of files to be stored on it. This maximizes the effectively usable disk space.
  - ✓ Partitioning disk blocks into groups. Each group will include data blocks and inodes stored on adjacent tracks so that files stored in a single block group can be accessed with a lower average disk seek time.
  - ✓ Pre-allocation of disk data blocks, leading to more physical contiguousness and hence lesser file fragmentation.
  - ✓ Fast symbolic link information storage in the inode, so as to avoid reading of actual data blocks on every file information request

File Systems software design – Mapping layer – Ext2, ext3 filesystems overview (contd.)

- The Ext3 file system has in the main the following additional feature (on top of the Ext2 features listed on previous slide)
  - ✓ Journaling:
    - The goal of a journaling file system is to avoid time-consuming consistency checks on the whole file system by looking instead at a special disk area that contains the most recent disk write operations, i.e. a journal

File Systems software design – IO scheduler

- IO scheduler
  - ✓ Information available to IO scheduler
    - Blocks to read from/write to
    - Queued requests in the order that they came in from the higher File System layer (VFS and mapping layers)
  - ✓ How does it view a file?
    - A set of logical block addressable (LBA) blocks

File Systems software design – IO scheduler

- ✓ What does it do?
  - Converts the FIFO queued read/write requests to a queue based on its internal algorithm
  - Most favoured algorithm is one of the Elevator-based ones

- ✓ How does it do it?
  - A kernel thread that processes the FIFO inputs and converts it into favoured output queue that is then passed to driver thread as grouped data buffers with LBA association

Thank You!