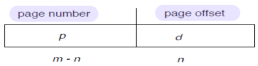
# Chapter 9 Main Memory

- Cache sits between main memory and CPU registers
- Base and limit registers define logical address space usable by a process
- Compiled code addresses bind to relocatable addresses
  - Can happen at three different stages
    - Compile time: If memory location known a priori, absolute code can be generated
    - Load time: Must generate relocatable code if memory location not known at compile time
    - Execution time: Binding delayed until run time if the process can be moved during its execution
- Memory-Management Unit (MMU) device that maps virtual to physical address
- Simple scheme uses a relocation register which just adds a base value to address
- Swapping allows total physical memory space of processes to exceed physical memory
  - Def: process swapped out temporarily to backing store then brought back in for continued execution
- Backing store: fast disk large enough to accommodate copies of all memory images
- Roll out, roll in: swapping variant for priority-based scheduling.
  - Lower priority process swapped out so that higher priority process can be loaded
- Solutions to Dynamic Storage-Allocation Problem:
  - First-fit: allocate the first hole that is big enough
  - Best-fit: allocate the smallest hole that is big enough (must search entire list) → smallest leftover hole
  - Worst-fit: allocate the largest hole (search entire list) → largest leftover hole
- External Fragmentation: total memory space exists to satisfy request, but is not contiguous
  - Reduced by compaction: relocate free memory to be together in one block
    - Only possible if relocation is dynamic
- Internal Fragmentation: allocated memory may be slightly larger than requested memory
- Physical memory divided into fixed-sized frames: size is power of 2, between 512 bytes and 16 MB
- Logical memory divided into same sized blocks: pages
- Page table used to translate logical to physical addresses
  - Page number (p): used as an index into a page table
  - Page offset (d): combined with base address to define the physical memory address
- Free-frame list is maintained to keep track of which frames can be allocated

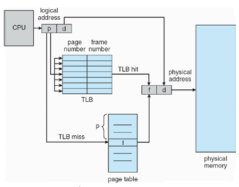


For given logical address space $2^m$ and page size $2^n$

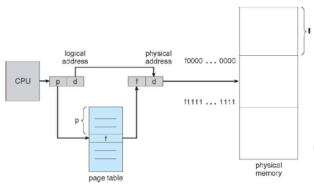
physical = frames
logical = pages
CPU generates logical address and MMU translates it to physical address aka generates the physical add.

lress translation in IA-32.

- Transition Look-aside Buffer (TLB) is a CPU cache that memory management hardware uses to improve virtual address translation speed
  - Typically small – 64 to 1024 entries
  - On TLB miss, value loaded to TLB for faster access next time
  - TLB is associative – searched in parallel



*Paging with TLB*



*Paging without TLB*

- Effective Access Time: EAT = (1 + ε) α + (2 + ε)(1 – α)
  - ε = time unit, α = hit ratio
- Valid and invalid bits can be used to protect memory
  - "Valid" if the associated page is in the process' logical address space, so it is a legal page
- Can have multilevel page tables (paged page tables)
- Hashed Page Tables: virtual page number hashed into page table
  - Page table has chain of elements hashing to the same location
  - Each element has (1) virtual page number, (2) value of mapped page frame, (3) a pointer to the next element
  - Search through the chain for virtual page number
- Segment table – maps two-dimensional physical addresses
  - Entries protected with valid bits and r/w/x privileges

**OSC 9.15:** Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:

**a.** External fragmentation

**Answer:** Contiguous allocation can result in external fragmentation, while paging essentially eliminates it.

**b.** Internal fragmentation

**Answer:** In contrast, contiguous allocation has no internal fragmentation, while it is possible for paging to have unused space in a page up to the page size - 1.

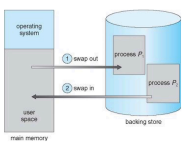
**c.** Ability to share code across processes

**Answer:** It is possible to map code to the same logical addresses across processes such that the pages referenced are the same. Then there needs only be 1 set of frames used to hold the code in memory. However, contiguous allocation requires all addresses reference by a process to be in its own process-specific memory region. Thus, it is not possible to share code in the same way. A copy of the code would be in the memory allocated to each process.

## Swapping

- Physical memory space is finite
- Suppose we allocate physical memory to meet the needs of a process
  - What happens when the total physical memory space requested by processes exceeds physical memory?
- A process can be *swapped* temporarily out of memory
  - Moved to a *backing store*– fast disk large enough to accommodate copies of all memory images for all users
  - Later brought back into memory for continued execution
- Swapping is used along with process scheduling
  - *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms

### Schematic View of Swapping




### Swapping Methods

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Also consider pending I/O since it might be associated with particular addresses
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if greater then memory allocation threshold
  - Disabled again once memory demand reduced

8.3 Why are page sizes always powers of 2?
Answer:
Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

### Swapping Constraints

- Other constraints as well on swapping
  - Can not swap out if process has pending I/O
    - otherwise I/O would occur to wrong process
  - Could transfer I/O to kernel space, then to I/O device
    - known as *double buffering*, adds overhead
- Standard swapping not really used in modern operating systems
  - But modified version common
    - swap only when free memory extremely low

multiple page entries for a frame = can share code data
Slack → C C → boot app
Chrome →

### Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - A page number consisting of 22 bits
  - A page offset consisting of 10 bits
- Since the page table is paged, the page number is divided into:
  - 12-bit page number
  - 10-bit page offset
- Thus, a logical address is as follows:

| page number | | page offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 12 | 10 | 10 |

where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the inner page table

- Known as *forward-mapped page table*

### Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
  - 100MB process swapping to hard disk with transfer rate of 50MB/sec
    - swap out time of 2000 ms
    - plus swap in of same sized process
    - total context switch swapping component time of 4000ms
- Can reduce if reduce size of memory swapped
  - Know how much memory really being used
  - System calls to inform OS of memory use via *request_memory()* and *release_memory()*

### IA-32 Logical to Physical Address Translation


| page number | | page offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 10 | 10 | 12 |

### 64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB ($2^{12}$)
  - Then page table has $2^{52}$ entries
  - If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  - Address would look like

| outer page | inner page | page offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 42 | 10 | 12 |

  - Outer page table has $2^{42}$ entries or $2^{44}$ bytes

### 2nd Outer Page Table

- One solution is to add a 2nd outer page table
- In this example, the 2nd outer page table is still 234 bytes in size
  - Possibly 4 memory access to get to one physical memory location

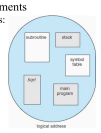
| outer page | inner page | offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 42 | 10 | 12 |

| 2nd outer page | outer page | inner page | offset |
|---|---|---|---|
| $p_1$ | $p_2$ | $p_3$ | $d$ |
| 32 | 10 | 10 | 12 |

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

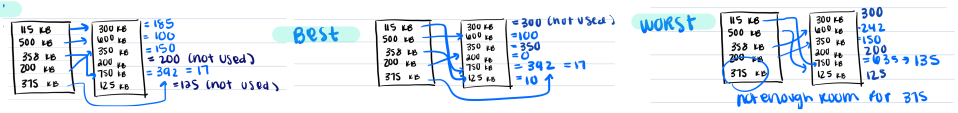
### Segmentation

- Memory-management scheme that supports a user view of memory
  - Allows different parts to be allocated separately
- A program is a collection of segments
  - A segment is a logical unit such as:
    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays




- **First fit.** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fi .** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit.** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

**OSC 9.6:** Given six memory partitions of 300KB, 600KB, 350KB, 200KB, 750KB, and 125KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115KB, 500KB, 358KB, 200KB, and 375KB (in order)? Rank the algorithms in terms of how efficiently they use memory.




**OSC 9.1:** What are the differences between logical and physical addresses.

**Answer:** A logical address is an address that references the (logical) address space of a process. That is, it is the address that is used by a process to access its memory space. Practically, it is the address generated by instructions during execution. A physical address is the address that is used to reference physical memory. It is generated during address translation, which maps logical addresses to physical addresses, in order to then access physical memory.

### Page Table TLB

- Some TLBs store *address-space identifiers (ASIDs)* in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be wired down for permanent fast access

### Fragmentation

- *External fragmentation*
  - Total memory space exists to satisfy a request, but it is not contiguous
- *Internal fragmentation*
  - Allocated memory may be slightly larger than requested memory
  - This size difference is memory internal to a partition, but not being used

### Fragmentation Improvement

- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem
    - keep job in memory while it is involved in I/O
    - do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

### Implementation of Page Table

- Page table is kept in main memory
- *Page-table base register (PTBR)* points to the page table
- *Page-table length register (PTLR)* indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *translation look-aside buffers (TLBs)* implemented with associative memory

**OSC 9.9:** Consider a logical address space of 256 pages of 4,096 bytes page size, mapped onto a physical memory of 64 frames.

**a.** How many bits are there in the logical address?
**Answer:** Total size of logical address space = $2^8 * 2^{12} = 2^{20}$. Thus, the logical address has 20 bits. You need 8 bits for the page number and 12 bits for the index into the page.

**b.** How many bits are there in the physical address?
**Answer:** Total size of physical address space = $2^6 * 2^{12} = 2^{18}$. Thus, the logical address has 18 bits. You need 6 bits for the frame number and again 12 bits for the index into the frame.

- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.
- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per-process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.
- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.
- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.
- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.
- The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports page-address extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.
- Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address.
- One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.

### Paging

- Physical address space of a process can be noncontiguous
  - Process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called *frames*
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called *pages*
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a page table to translate logical to physical addresses
- Backing store likewise split into pages
- Still have internal fragmentation

# Chapter 10 - Virtual Memory

Phenomenon in which increasing the number of page frames results in an increase in the # of page faults for certain memory access patterns. common w/ FIFO.

| Rank | Algorithm | Suffer from Belady's anomaly |
|---|---|---|
| 1 | Optimal | no |
| 2 | LRU | no |
| 3 | Second-chance | yes |
| 4 | FIFO | yes |

**9.1** Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

**Answer:**

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

**9.5** Discuss the hardware support required to support demand paging.

**Answer:**

For every memory-access operation, the page table needs to be consulted to check whether the corresponding page is resident or not and whether the program has read or write privileges for accessing the page. These checks have to be performed in hardware. A TLB could serve as a cache and improve the performance of the lookup operation.

**OSC 10.21:** Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

**OSC 10.37:** What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

**Answer:** Thrashing is occurs in a virtual memory system when there is an increase in the number of page faults in the system caused by the inability to keep pages that processes need in memory. Thrashing can be attributed to memory allocation difficulties brought on by increasing the degree of multiprogramming. Essentially, the higher the degree of multiprogramming, the smaller the amount of memory that can be allocated to any process. Consequently, the OS will spend a relatively larger amount of time time servicing page faults versus performing useful work. Thrashing can be detected when the page fault rates increases and the CPU utilization falls significantly. The only realy way to eliminate the problem is to reduce the degree of multiprogramming to an acceptable level.

## Summary

- Virtual memory abstracts physical memory into an extremely large uniform array of storage.
- The benefits of virtual memory include the following: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- Demand paging is a technique whereby pages are loaded only when they are demanded during program execution. Pages that are never demanded are thus never loaded into memory.
- A page fault occurs when a page that is currently not in memory is accessed. The page must be brought from the backing store into an available page frame in memory.
- Copy-on-write allows a child process to share the same address space as its parent. If either the child or the parent process writes (modifies) a page, a copy of the page is made.
- When available memory runs low, a page-replacement algorithm selects an existing page in memory to replace with a new page. Page-replacement algorithms include FIFO, optimal, and LRU. Pure LRU algorithms are impractical to implement, and most systems instead use LRU-approximation algorithms.
- Global page-replacement algorithms select a page from any process in the system for replacement, while local page-replacement algorithms select a page from the faulting process.
- Thrashing occurs when a system spends more time paging than executing.
- A locality represents a set of pages that are actively used together. As a process executes, it moves from locality to locality. A working set is based on locality and is defined as the set of pages currently in use by a process.
- Memory compression is a memory-management technique that compresses a number of pages into a single page. Compressed memory is an alternative to paging and is used on mobile systems that do not support paging.
- Kernel memory is allocated differently than user-mode processes; it is allocated in contiguous chunks of varying sizes. Two common techniques for allocating kernel memory are (1) the buddy system and (2) slab allocation.
- TLB reach refers to the amount of memory accessible from the TLB and is equal to the number of entries in the TLB multiplied by the page size. One technique for increasing TLB reach is to increase the size of pages.
- Linux, Windows, and Solaris manage virtual memory similarly, using demand paging and copy-on-write, among other features. Each system also uses a variation of LRU approximation known as the clock algorithm.

EXAMPLE:

L.A.S. = 4 GB → $2^2 \cdot 2^{30} = 2^{32}$ → LA: 20 | 12 = 32

P.A.S. = 64 GB → $2^6 \cdot 2^{20} = 2^{26}$ → PA: 14 | 12 = 26 (frames)

page size = 4 KB = 4064 = $2^{12}$ = offset

# pages = $2^{20}$

# frames = $2^{14}$

# entries into page table = $2^{20}$

size of page table = $2^{20} \times 14$ bits

**9.10** You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.

**Answer:**

No. An optimal algorithm will not suffer from Belady's anomaly because—by definition—an optimal algorithm replaces the page that will not be used for the longest time. Belady's anomaly occurs when a page-replacement algorithm evicts a page that will be needed in the immediate future. An optimal algorithm would not have selected such a page.

**9.9** Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware, or explain why it is not possible to do so. If it is possible, calculate what the cost would be.

**Answer:**

You can use the valid/invalid bit supported in hardware to simulate the reference bit. Initially set the bit to invalid. On first reference a trap to the operating system is generated. The operating system will set a software bit to 1 and reset the valid/invalid bit to valid.

**9.11** Segmentation is similar to paging but uses variable-sized "pages." Define two segment-replacement algorithms based on FIFO and LRU page-replacement schemes. Remember that since segments are not the same size, the segment that is chosen to be replaced may not be big enough to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated, and those for systems where they can.

**Answer:**

a. FIFO. Find the first segment large enough to accommodate the incoming segment. If relocation is not possible and no one segment is large enough, select a combination of segments whose memories are contiguous, which are "closest to the first of the list" and which can accommodate the new segment. If relocation is possible, rearrange the memory so that the first $N$ segments large enough for the incoming segment are contiguous in memory. Add any leftover space to the free-space list in both cases.

b. LRU. Select the segment that has not been used for the longest period of time and that is large enough, adding any leftover space to the free space list. If no one segment is large enough, select a combination of the "oldest" segments that are contiguous in memory (if relocation is not available) and that are large enough. If relocation is available, rearrange the oldest $N$ segments to be contiguous in memory and replace those with the new segment.

**9.7** Consider the two-dimensional array A:

```
int A[][] = new int[100][100];
```

where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement and assuming that page frame 1 contains the process and the other two are initially empty?

a.
```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```

b.
```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

**Answer:**

a. 5,000

b. 50

**9.12** Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of CPU and the paging disk. The results are one of the following alternatives. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

a. CPU utilization 13 percent; disk utilization 97 percent

b. CPU utilization 87 percent; disk utilization 3 percent

c. CPU utilization 13 percent; disk utilization 3 percent

**Answer:**

a. Thrashing is occurring.

b. CPU utilization is sufficiently high to leave things alone, and increase degree of multiprogramming.

c. Increase the degree of multiprogramming.

**9.13** We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can the page tables be set up to simulate base and limit registers? How can they be, or why can they not be?

**Answer:**

The page table can be set up to simulate base and limit registers provided that the memory is allocated in fixed-size segments. In this way, the base of a segment can be entered into the page table and the valid/invalid bit used to indicate that portion of the segment as resident in the memory. There will be some problem with internal fragmentation.

**Answer:** The effective access time formula we are interested in is:

$$r * .3 * 8 + r * .7 * 20 + (1 - r) * .0001 = .0002$$

where $r$ is the page fault rate. Solving for $r$ we get:

$$2.4r + 1.4r - r * .0001 = 3.7999r = .0001$$

$$r = .0001/3.7999 = .0000263$$

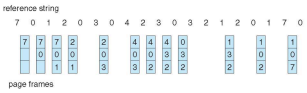
To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal, and determine the location of the page in secondary storage.
5. Issue a read from the storage to a free frame:
   a. Wait in a queue until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU core to some other process.
7. Receive an interrupt from the storage I/O subsystem (I/O completed).
8. Save the registers and process state for the other process (if step 6 is executed).
9. Determine that the interrupt was from the secondary storage device.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU core to be allocated to this process again.
12. Restore the registers, process state, and new page table, and then resume the interrupted instruction.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 10.5):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

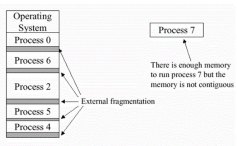
- Virtual memory: separation of user logical memory and physical memory
  - Only part of program needs to be in memory for execution → logical address space > physical address space
  - Allows address spaces to be shared by multiple processes → less swapping
  - Allows pages to be shared during fork(), speeding process creation
- Page fault results from the first time there is a reference to a specific page → traps the OS
  - Must decide to abort if the reference is invalid, or if the desired page is just not in memory yet
    - If the latter: get empty frame, swap page into frame, reset tables to indicate page now in memory, set validation bit, restart instruction that caused the page fault
  - If an instruction accesses multiple pages near each other → less "pain" because of locality of reference
- Demand Paging only brings a page into memory when it is needed → less I/O and memory needed
  - Lazy swapper – never swaps a page into memory unless page will be needed
  - Could result in a lot of page-faults
  - Performance: EAT = [(1-p)*memory access + p*(page fault overhead + swap page out + swap page in + restart overhead)]; where Page Fault Rate 0 ≤ p ≤ 1
    - if p = 0, no page faults; if p = 1, every reference is a fault
  - Can optimize demand paging by loading entire process image to swap space at process load time
- Pure Demand Paging: process starts with no pages in memory
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- Modify (dirty) bit can be used to reduce overhead of page transfers → only modified pages written to disk
- When a page is replaced, write to disk if it has been marked dirty and swap in desired page
- Pages can be replaced using different algorithms: FIFO, LRU (below)
  - Stack can be used to record the most recent page references (LRU is a "stack" algorithm)



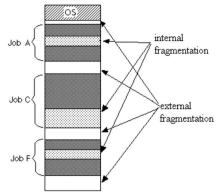

  - Second chance algorithm uses a reference bit
    - If 1, decrement and leave in memory
    - If 0, replace next page

- Fixed page allocation: Proportional allocation – Allocate according to size of process
  - $s_i$ = size of process $P_i$, $S = \Sigma s_i$, m = total number of frames, $a_i$ = allocation for $P_i$
  - $a_i = (s_i/S)*m$
- Global replacement: process selects a replacement frame from set of all frames
  - One process can take frame from another
  - Process execution time can vary greatly
  - Greater throughput
- Local replacement: each process selects from only its own set of allocated frames
  - More consistent performance
  - Possible under-utilization of memory
- Page-fault rate is very high if a process does not have "enough" pages
  - Thrashing: a process is busy swapping pages in and out → minimal work is actually being performed
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- I/O Interlock: Pages must sometimes be locked into memory

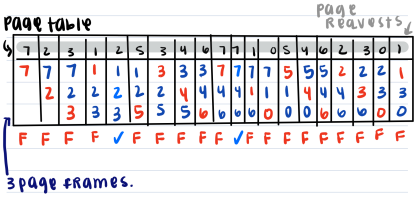
**External fragmentation**
Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous so it can not be used.



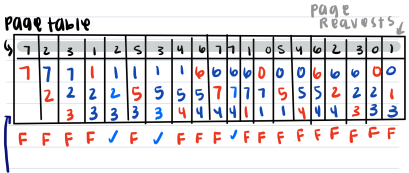

**Internal fragmentation**
Memory block assigned to process is bigger. Some portion of memory is left unused as it can not be used by another process.



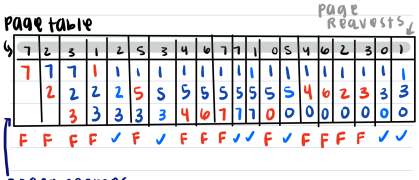

LRU = Least Recently Used
When a page frame is full, then replace the least recently used reavest with the current page reavest.




FIFO = First in, First out
When a page frame is full, then replace the first reavest with the current page reavest.




Optimal Replacement.
When a page frame is full, then replace page we don't use for the longest time w/ current page reavest.

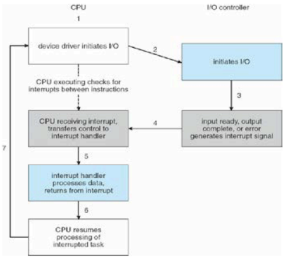


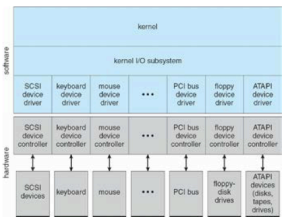

200 NS Nanosec. = 0.0002 Milisec. MS

200 Nanosec. = 0.2 Microsec. us

# Chapter 12 I/O Systems

- Device drivers encapsulate device details – present uniform device access interface to I/O subsystem
- Port: connection point for device
- Bus: daisy chain or shared direct access
- Controller (host adapter): electronics that operate port, bus, device – sometimes integrated
  - Contains processor, microcode, private memory, bus controller
- Memory-mapped I/O: device data and command registers mapped to processor address space
  - Especially for large address spaces (graphics)
- Polling for each byte of data – busy-wait for I/O from device
  - Reasonable for fast devices, inefficient for slow ones
  - Can happen in 3 instruction cycles
- CPU interrupt-request line is triggered by I/O devices – interrupt handler receives interrupts
  - Handler is maskable to ignore or delay some interrupts
  - Interrupt vector dispatches interrupt to correct handler – based on priority; some nonmaskable
  - Interrupt chaining occurs if there is more than one device at the same interrupt number
  - Interrupt mechanism is also used for exceptions
- Direct memory access is used to avoid programmed I/O for large data movement
  - Requires DMA controller
  - Bypasses CPU to transfer data directly between I/O device and memory
- Device driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions: character stream/block, sequential/random access, synchronous/asynchronous, sharable/dedicated, speed, rw/ro/wo
- Block devices include disk drives: Raw I/O, Direct I/OU
  - Commands include read, write, seek
- Character devices include keyboards, mice, serial ports
  - Commands include get(), put()
- Network devices also have their own interface; UNIX and Windows NT/9x/2000 include socket interface
  - Approaches include pipes, FIFOs, streams, queues, mailboxes
- Programmable interval timer: used for timings, periodic interrupts
- Blocking I/O: process suspended until I/O completed – easy to use and understand, not always best method
- Nonblocking I/O: I/O call returns as much as available – implemented via multi-threading, returns quickly
- Asynchronous: process runs while I/O executes – difficult to use, process signaled upon I/O completion
- Spooling: hold output for a device – if device can only serve one request at a time (ex: printer)
- Device Reservation: provides exclusive access to a device – must be careful of deadlock
- Kernel keeps state info for I/O components, including open file tables, network connections, character device states
  - Complex data structures track buffers, memory allocation, "dirty" blocks
- STREAM: full-duplex communication channel between user-level process and device in UNIX
  - Each module contains read queue and write queue
  - Message passing used to communicate between queues – Flow control option to indicate available or busy
  - Asynchronous internally, synchronous where user process communicates with stream head
- I/O is a major factor in system performance – demand on CPU, context switching, data copying, network traffic




**OSC 12.5:** Why might a system use interrupt-driven I/O to manage a single serial port and polling I/O to manage a front-end processor, such as a terminal concentrator?

**Answer:** The behavior of a single serial port is likely to be more variable and asynchronous than a concentrator that is aggregating input from multiple devices. In this case, it is more efficient to service the serial port via interrupts. Polling is perhaps better used for the concentrator, since there is a higher likelihood that its servicing needs will be more regular and continous. Also, often these have limited buffering that requires more constant checking.

**OSC 12.5:** How does DMA increase system concurrency? How does it complicate hardware design?

**Answer:** DMA (direct memory access) increases (the potential for) system concurrency because it frees the CPU from the burden and overhead of "programmed I/O" operations. The allows the CPU to be used for other things while the I/O is taking place. Devices that want to use DMA have to have "DMA controller" hardware that coordinates data transfers between the device and the memory sub-system. Some DMA controllers are designed to work with physical addresses, while others can work with virtual addresses. Depending on the bandwidth and device complexity, DMA controller hardware can be more or less complicated.

**OSC 12.13:** Describe two circumstances under which blocking I/O should be used. Describe two circumstances under which nonblocking I/O should be used. Why not just implement nonblocking I/O and have processes busy-wait until their devices are ready?

**Answer:** Generally, blocking I/O is appropriate when the process will be waiting only for one specific event. Examples include a disk, tape, or keyboard read by an application program. Non-blocking I/O is useful when I/O may come from more than one source and the order of the I/O arrival is not predetermined. Examples include network daemons listening to more than one network socket, window managers that accept mouse movement as well as keyboard input, and I/O-management programs, such as a copy command that copies data between I/O devices. In the last case, the program could optimize its performance by buffering the input and output and using non-blocking I/O to keep both devices fully occupied.

Non-blocking I/O is more complicated for programmers, because of the asynchronous rendezvous that is needed when an I/O occurs. Also, busy waiting is less efficient than interrupt-driven I/O so the overall system performance would decrease.

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware design, the concepts that we have just described are sufficient to enable us to understand many I/O features of operating systems. Let's review the main concepts:

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller
- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel. The I/O subsystem supervises these procedures:

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot `seek()`)
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization
- Power management of I/O devices

- The basic hardware elements involved in I/O are buses, device controllers, and the devices themselves.
- The work of moving data between devices and main memory is performed by the CPU as programmed I/O or is offloaded to a DMA controller.
- The kernel module that controls a device is a device driver. The system-call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character-stream devices, memory-mapped files, network sockets, and programmed interval timers. The system calls usually block the processes that issue them, but nonblocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.
- The kernel's I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, caching, spooling, device reservation, error handling. Another service, name translation, makes the connections between hardware devices and the symbolic file names used by applications. It involves several levels of mapping that translate from character-string names, to specific device drivers and device addresses, and then to physical addresses of I/O ports or bus controllers. This mapping may occur within the file-system name space, as it does in UNIX, or in a separate device name space, as it does in MS-DOS.
- STREAMS is an implementation and methodology that provides a framework for a modular and incremental approach to writing device drivers and network protocols. Through STREAMS, drivers can be stacked, with data passing through them sequentially and bidirectionally for processing.

I/O system calls are costly in terms of CPU consumption because of the many layers of software between a physical device and an application. These layers imply overhead from several sources: context switching to cross the kernel's protection boundary, signal and interrupt handling to service the I/O devices, and the load on the CPU and memory system to copy data between kernel buffers and application space.

- The **data-in register** is read by the host to get input.
- The **data-out register** is written by the host to send output.
- The **status register** contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The **control register** can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex com-

### *Nonblocking and Asynchronous I/O*

- *Blocking* – process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- *Nonblocking* – I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - *select()* to find if data ready, *read()* or *write()* to transfer
- *Asynchronous* – process runs while I/O executes
  - More tricky to use
  - I/O subsystem signals process when I/O completed

### *Kernel I/O Subsystem*

- *Caching* – faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- *Spooling* – hold output for a device
  - If device can serve only one request at a time
  - Print spooling is a good example
- *Device reservation* – provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

### *Polling*

- For each byte of I/O
  - Read busy bit from status register until 0
  - Host sets read or write bit
    - If write copies data into data-out register
  - Host sets command-ready bit
  - Controller sets busy bit, executes transfer
  - Controller clears busy bit, error bit, command-ready bit when transfer done
- Step 1 is *busy-wait* cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - but if miss a cycle data overwritten / lost

### *Interrupts (1)*

- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
  - How to avoid the overhead of polling?
- CPU *interrupt-request line* triggered by I/O device
  - Checked by processor after each instruction
- *Interrupt handler* receives interrupts
  - *Maskable* to ignore or delay some interrupts
- *Interrupt vector* to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some *nonmaskable*
- Interrupt chaining if more than one device at same interrupt number

### *Interrupts (2)*

- Interrupt mechanism also used for *exceptions*
  - Terminate process, crash system due to hardware error
  - Page fault executes when memory access error
- System call executes via *trap* to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing
  - Frequent interrupts that must be processed fast

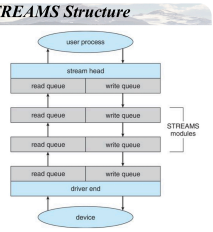
### *Direct Memory Access (DMA)*

- Used to avoid *programmed I/O* (one byte at a time)
  - Primarily for large data movement
- Requires DMA controller
  - Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - *cycle stealing* from CPU but still much more efficient
  - When done, interrupts to signal completion
- If aware of virtual addresses, it can be even more efficient
  - Direct Virtual Memory Access (DVMA)

### *STREAMS*

- *STREAM* – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
- A STREAM consists of:
  - STREAM head interfaces with the user process
  - driver end interfaces with the device
  - zero or more STREAM modules between them
- Each module contains a *read queue* and a *write queue*
- Message passing is used to communicate between queues
  - Flow control option to indicate available or busy
- Asynchronous internally, synchronous where user process communicates with stream head

### *The STREAMS Structure*




### *Performance*

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful

### *Improving Performance*

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads

Interprocess comm (IPC): a mechanism for communication between 2 or more processes running on 1 or more computer systems. It provides for interprocess interaction and data exchange. The 2 IPC models are shared memory and message passing. Shared memory segments and pipes are types of shared memory IPC.

Events in a context-switch between kernel-level threads: if the context-switch is to a thread in a different process, it is necessary to do a full process context switch. In this case, the running thread's CPU state is saved in the thread-specific region of its process control block AND the process-related information that needs to be saved is handled. The thread is put into the appropriate thread state, depending on what caused the context switch. The process information of the thread to be run is loaded, the thread CPU state is loaded, and the new thread starts executing. Second, if the threads are from the same process, it is only necessary to switch the thread CPU states.This is a much faster operation.

Threading models: many (user) to many (kernel), many to one, one to one

### Process and Threads

Possible states of a process and reasons for state transitions: new (upon creation), ready (waiting to be scheduled by the OS onto the CPU), running, possibly waiting (by interrupt, preemption, etc) followed by ready, terminate.

Process address space: process address space(or logical address space of a process) is the range of addresses, starting at logical address 0 to $2^n$-1 (where n is the # bits) that any process can reference. It contains the process text, global data, heap and stack, each is arranged in particular places in the PAS.

Interrupts: interrupts are a hardware-based mechanism to stop the processor from executing the current process and start executing the operating system. The OS runs an interrupt handler to do what needs to be done for the specific interrupt. In general, interrupts serves a role of giving control back to the OS.

Traps: traps are generated by software instead of the hardware.

### Synchronization (Problem)

Concurrency and purpose: the interleaving of processes in time to give the appearance of simultaneous execution. It differs from parallelism, which offers genuine simultaneous execution. Concurrent programming offers increased efficiency of an application.

Atomic instructions: a set of instructions is atomic if they cannot be interrupted

Race condition: when two or more threads can access shared data and they try to change it at the same time.

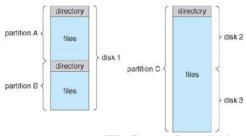
# Chapter 13 File System Interface

- File – Uniform logical view of information storage (no matter the medium)
  - Mapped onto physical devices (usually nonvolatile)
  - Smallest allotment of nameable storage
  - Types: Data (numeric, character, binary), Program, Free form, Structured
  - Structure decided by OS and/or program/programmer
- Attributes:
  - Name: Only info in human-readable form
  - Identifier: Unique tag, identifies file within the file system
  - Type, Size
  - Location: pointer to file location
  - Time, date, user identification

### File Operations

- Sequential Access
  - *reset*
  - *read* next (advance file pointer automatically)
  - *write* next (advance file pointer automatically)
- Direct Access
  - *read n*
  - *write n*
  - *position* to *n*
  - *read* next
  - *write* next
- *n* = relative block number

- File is an abstract data type
- Operations: create, write, read, reposition within file, delete, truncate
- Global table maintained containing process-independent open file information: open-file table
  - Per-process open file table contains pertinent info, plus pointer to entry in global open file table
- Open file locking: mediates access to a file (shared or exclusive)
  - Mandatory – access denied depending on locks held and requested
  - Advisory – process can find status of locks and decide what to do
- File type can indicate internal file structure
- Access Methods: Sequential access, direct access
  - Sequential Access: tape model of a file
  - Direct Access: random access, relative access
- Disk can be subdivided into partitions; disks or partitions can be RAID protected against failure.
  - Can be used raw without a file-system or formatted with a file system
  - Partitions also knows as minidisks, slices
- Volume contains file system: also tracks file system's info in device directory or volume table of contents
- File system can be general or special-purpose. Some special purpose FS:
  - tmpfs – temporary file system in volatile memory
  - objfs – virtual file system that gives debuggers access to kernel symbols
  - ctfs – virtual file system that maintains info to manage which processes start when system boots
  - lofs – loop back file system allows one file system to be accessed in place of another
  - procfs – virtual file system that presents information on all processes as a file system
- Directory is similar to symbol table – translating file names into their directory entries
  - Should be efficient, convenient to users, logical grouping
  - Tree structured is most popular – allows for grouping
  - Commands for manipulating: remove – rm<file-name> ; make new sub directory - mkdir<dir-name>
- Current directory: default location for activities – can also specify a path to perform activities in
- Acyclic-graph directories adds ability to directly share directories between users
  - Acyclic can be guaranteed by: only allowing shared files, not shared sub directories; garbage collection; mechanism to check whether new links are OK
- File system must be mounted before it can be accessed – kernel data structure keeps track of mount points
- In a file sharing system User IDs and Group IDs help identify a user's permissions
- Client-server allows multiple clients to mount remote file systems from servers – NFS (UNIX), CIFS (Windows)
- Consistency semantics specify how multiple users are to access a shared file simultaneously – similar to synchronization algorithms from Ch.7
  - One way of protection is Controlled Access: when file created, determine r/w/x access for users/groups




*File-System Organization*

### 13.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these seven basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **Creating a fil** . Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 14. Second, an entry for the new file must be made in a directory.
- **Opening a fil** . Rather than have all file operations specify a file name, causing the operating system to evaluate the name, check access permissions, and so on, all operations except create and delete require a file open() first. If successful, the open call returns a file handle that is used as an argument in the other calls.
- **Writing a fil** . To write a file, we make a system call specifying both the open file handle and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place if it is sequential. The write pointer must be updated whenever a write occurs.
- **Reading a fil** . To read from a file, we use a system call that specifies the file handle and where (in memory) the next block of the file should be put. Again, the system needs to keep a read pointer to the location in the file where the next read is to take place, if sequential. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-positio pointer. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file.** The current-file-position pointer of the open file is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file seek.
- **Deleting a fil** . To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry. Note that some systems allow hard links—multiple names (directory entries) for the same file. In this case the actual file contents is not deleted until the last link is deleted.
- **Truncating a fil** . The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length. The file can then be reset to length zero, and its file space can be released.

### 13.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, and another user might edit that file by specifying its name. The file's owner might write the file to a USB drive, send it as an e-mail attachment, or copy it across a network, and it could still be called `example.c` on the destination system. Unless there is a sharing and synchonization method, that second copy is now independent of the first and can be changed separately.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name.** The symbolic file name is the only information kept in human-readable form.
- **Identifie** . This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type.** This information is needed for systems that support different types of files.
- **Location.** This information is a pointer to a device and to the location of the file on that device.
- **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- **Timestamps and user identificatio** . This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

### 13.4.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an **access-control list** (ACL) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list.

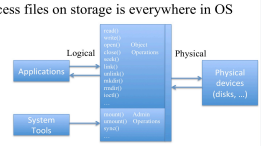
To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner.** The user who created the file is the owner.
- **Group.** A set of users who are sharing the file and need similar access is a group, or work group.
- **Other.** All other users in the system.

| | |
|---|---|
| 2 | $2^1$ |
| 4 | $2^2$ |
| 8 | $2^3$ |
| 16 | $2^4$ |
| 32 | $2^5$ |
| 64 | $2^6$ |
| 128 | $2^7$ |
| 256 | $2^8$ |
| 512 | $2^9$ |
| 1K/1024 | $2^{10}$ |
| 4K/4096 | $2^{12}$ |
| 1M | $2^{20}$ |
| 1G | $2^{30}$ |
| 1T | $2^{40}$ |

### File System Interface

- Most visible part of the OS
  - Need to access files on storage is everywhere in OS
- Consists of
  - Files
  - Directories
  - Partitions




### Access Control/Authorization

- An access control system determines what *rights* a particular *entity* has for a set of *objects* (allows what you can make active entities to do things; passive things that things can be done to)
- It answers the question
  - Do you have the right to read /etc/passwd?
  - Does Alice have the right to view the CIS website?
  - Do students have the right to share project data?
  - Does Prof. Malony have the right to change your grades?
- An *Access Control Policy* answers these questions

### Protection vs Security

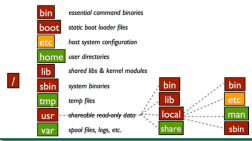
- Protection
  - Security goals met under trusted processes
  - Protects against an error by a non-malicious entity
- Security
  - Security goals met under potentially malicious processes
  - Protects against any malicious entity
  - Hence, for J:
    - non-malicious process shouldn't leak the private key by writing it to $O_3$
    - a potentially malicious process may contain a Trojan horse that can write the private key to $O_3$

### Links

- Two different names (aliasing)
- Link
  - Makes a file appear in more than 1 directory
- Is a convenience in several situations
- 2 types of links:
  - Soft links
    - create a file which contains the name of the other file
    - use this path name to get to the actual file when it is accessed
    - problem: Extra overhead of parsing and following components till the file is reached.
  - Hard links
    - create a directory entry and make a reference that file.
    - others may reference the same file

### File System Hierarchy Standard

- UNIX *filesystem hierarchy standard* (FHS)
  - Convention to enable software and users to find the location of installed files and directories




**OSC 13.2:** Why do some systems keep track of the type of a file, while others leave it to the user, and others simple do not implement multiple file types? Which system is "better" in you opinion?

**Answer:** Keeping track of file types can be useful for more efficiently using and storing a file. It would allow type-specific file interfaces to be applied to file operations, without having to determine how the file is structured and its data formatted in some other way. It also allows the OS to manage files according to their type and application programs to quickly check their validity. While different types of files will have different structure and data formats, there are more common types of files than others. A system might keep track of only standard file types, leaving it up to the user for other types it does not recognize. For these cases, typically file "header" data is used to describe the type, structure, and format.

A decision to not implement multiple file types could be motivated by, for instance, storage efficiences if all file are treated the same. Or it might be because special files would have to have file headers, so why not just have that be the default.

A combination of common file types and user-defined types would be better, in my opinion.

**OSC 13.5:** What is the purpose of the `open()` and `close()` operations?

**Answer:** These calls essentially set up and terminate file activity is a way that lets the OS record and track more efficiency which files the process is working with and gives the process a more succinct "file handle" for other file operations, rather than always requiring the file pathname to be searched for in the file systems.

**OSC 13.15:** Give an example of an application that could benefit from operating system support for random access to indexed files.

**Answer:** Indexed allocation uses an index directory to find blocks for purposes of rapid positioning. It is better to use this strategy for larger files that are supporting non-sequential access. For instance, working on a large spreadsheet with multiple rows and columns stored in a single file would benefit from faster positioning to a particular cell in the table.

## 13.6 Summary

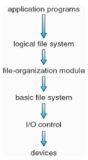
- A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (of fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.
- A major task for the operating system is to map the logical file concept onto physical storage devices such as hard disk or NVM device. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.
- Within a file system, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user's files. The directory lists the files by name and includes the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.
- The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize files. Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.
- Remote file systems present challenges in reliability, performance, and security. Distributed information systems maintain user, host, and access information so that clients and servers can share state information to manage use and access.
- Since files are the main information-storage mechanism in most computer systems, file protection is needed on multiuser systems. Access to files can be controlled separately for each type of access—read, write, execute, append, delete, list directory, and so on. File protection can be provided by access lists, passwords, or other techniques.

### Open Files

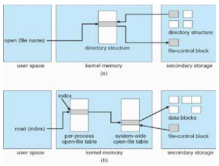
- Several data are needed to manage open files:
  - *Open file table*: tracks open files
  - *File pointer*: pointer to last read/write location, per process that has the file open
  - *File open count*: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - *Disk location of the file*: cache of data access information
  - *Access rights*: per-process access mode information

# Chapter 14 - File System Implementation

## Ch.12 – File System Implementation



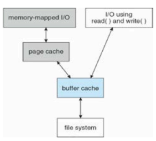

- File system resides on secondary storage – disks; file system is organized into layers →
- File control block: storage structure consisting of information about a file (exist per-file)
- Device driver: controls the physical device; manage I/O devices
- File organization module: understands files, logical addresses, and physical blocks
  - Translates logical block number to physical block number
  - Manages free space, disk allocation
- Logical file system: manages metadata information – maintains file control blocks
- Boot control block: contains info needed by system to boot OS from volume
- Volume control block: contains volume details; ex: total # blocks, # free blocks, block size, free block pointers
- Root partition: contains OS; mounted at boot time
- For all partitions, system is consistency checked at mount time
  - Check metadata for correctness – only allow mount to occur if so
- Virtual file systems provide object-oriented way of implementing file systems
- Directories can be implemented as Linear Lists or Hash Tables
  - Linear list of file names with pointer to data blocks – simple but slow
  - Hash table – linear list with hash data structure – decreased search time
    - Good if entries are fixed size
    - Collisions can occur in hash tables when two file names hash to same location




*(a) open()* *(b) read()*

- Contiguous allocation: each file occupies set of contiguous blocks
  - Simple, best performance in most cases; problem – finding space for file, external fragmentation
  - Extent based file systems are modified contiguous allocation schemes – extent is allocated for file allocation
- Linked Allocation: each file is a linked list of blocks – no external fragmentation
  - Locating a block can take many I/Os and disk seeks
- Indexed Allocation: each file has its own index block(s) of pointers to its data blocks
  - Need index table; can be random access; dynamic access without external fragmentation but has overhead
- Best methods: linked good for sequential, not random; contiguous good for sequential and random
- File system maintains free-space list to track available blocks/clusters
- Bit vector or bit map (n blocks): block number calculation → (#bits/word)*(# 0-value words)+(offset for 1st bit)
  - Example:

    block size = 4KB = 212 bytes
    disk size = 240 bytes (1 terabyte)
    $n$ = 240/212 = 228 bits (or 256 MB)
    if clusters of 4 blocks -> 64MB of memory




- Space maps (used in ZFS) divide device space into metaslab units and manages metaslabs
  - Each metaslab has associated space map
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS – no buffering
  - Asynchronous writes are more common, buffer-able, faster
- Free-behind and read-ahead techniques to optimize sequential access
- Page cache caches pages rather than disk blocks using virtual memory techniques and addresses
  - Memory mapped I/O uses page cache while routine I/O through file system uses buffer (disk) cache
- Unified buffer cache: uses same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching

**OSC 14.3:** Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

**Answer:** Contiguous allocation requires that each file occupy a set of contiguous blocks on disk. It facilitates file access and can be higher performing, but can cause problems with finding space for new files. Short files that are long-lived and do not fluctuate in size over time are good candidates for contiguous allocation.

Linked allocation can address contiguous allocation problems by maintaining a linked list of disk blocks that make up a file. It is easy to extend the file by adding more blocks to the list. It is effective for files of medium size that might grow and shrink, but is really for sequential access only because positioning within a file requires traversing the links to find the righ tblock.

Indexed allocation uses an index directory to find blocks for purposes of rapid positioning. It is better to use this strategy for larger files that are supporting non-sequential access.

**OSC 14.4:** One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area of a specified size. If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

**Answer:** The solution is certainly an improvement over the standard contiguous allocation since it solves the problem mentioned, allowing the file to grow in size. It has similarities to the linked allocation, but does not necessarily require a particular blocking factor to be used.

**OSC 14.11:** Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4KB of disk space as a single 4KB block or as eight 512B blocks. How could we take advantage of this flexibility to improve performance? What modifications would have to be made to the free-space management scheme in order to support this feature?

**Answer:** There are 2 general performance benefits that come from variable granularity in storage allocation. First, there is the benefit of smaller granularity in storage efficiency, due to less potential internal fragmentation. Second, larger granularity can benefit larger files, especially for sequential access, because larger blocks of data can be moved on/off the disk at a time, without having to reposition to another block. Essentially, the free-space management scheme will have to be modified cognizant of the block size. It could have a fixed partitioning of the disk for blocks for different sizes. Then allocation could be done for each granularity using only the blocks of that size. The free-space management then would be the same as before, except for each level of granularity.

Another strategy is to allocate everything at a small granularity, but allow contiguous smaller granularity blocks to be dynamically combined into a larger granularity block. This would make the free-space management more complicated as it would keep track of all of the free small granularity blocks, but also store information to indicate if a block's neighbor was also free. Here, the allocator might use some optimization to represent free blocks of different granularities and to decide which blocks to allocate to retain larger blocks in the free list.

### *Virtual File Systems (1)*

- File systems have general and storage method-dependent parts
  - *Virtual file system* is specific to the OS
    - file system-generic operations
    - works with inodes (FCB), files, directories, superblocks (partitions)
    - the stuff that we have discussed
  - Physical file system is specific to how secondary storage will be used to manage data
    - converts the objects above into blocks

### *File System Implementation (1)*

- We have system calls at the API level, but how implement their functions?
  - On-disk and in-memory structures
- *Boot control block* contains info needed by syst boot OS from that volume
  - Needed if volume contains OS
    - usually first block of volume
- *Volume control block* (superblock, master file ta contains volume details (aka *partition control b*
  - Total # of blocks, # of free blocks, block size, free b pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

### *Virtual File Systems (2)*

- *Virtual File Systems (VFS)* on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - implements *vnodes* which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines

### *Disks as Secondary Store*

- What makes them convenient for storing file
  - Rewrite in place
    - read, modify in memory, write back to original locat
  - Access any block (sequentially or random)
    - got to move the head to get there
  - I/O transfers performed in blocks of sectors
    - usually 512 bytes
- Solid-state drives (SSDs) becoming more pr

### *File System Layers (1)*

- *Device drivers* manage I/O devices at the *I/O control layer*
  - High-level commands like:
    "read drive1, cylinder 72, track 2, sector 10, into memory location 1060"
  - Translate into low-level hardware specific commands to hardware controller
- Basic file system given command like "retrieve block 123" translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- *File organization module* understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation

### *File System Layers (2)*

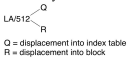
- *Logical file system* manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks
    - *inodes* in UNIX
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer

### *File System Layers (3)*

- Many different file systems
  - Sometimes many within an operating system
  - Each with its own format
    - CD-ROM is ISO 9660
    - Unix has UFS (Unix File System), FFS (Fast File System)
    - Windows has FAT, FAT32, NTFS, floppy, CD, DVD Blu-ray
    - Linux has more than 40 types, with extended file system ext2 and ext3 leading, plus distributed file systems, …
  - New ones still arriving
    - ZFS, GoogleFS, Oracle ASM, FUSE, Lustre, …

### *Indexed Allocation*

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of max size of 256K bytes and block size of 512 bytes
  - We need only 1 block for index table



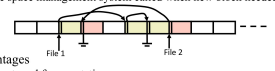

Q = displacement into index table
R = displacement into block

### *Contiguous Allocation*

- Contiguous allocation is where each file occupies set of contiguous blocks
- Advantages:
  - Simple – need to remember only starting location to access any block, plus length (number of blocks)
  - Good performance when reading successive blocks on disk
    - best performance in most cases
- Disadvantages:
  - File size has to be known a priori
    - need to find space for the file of that size
  - External fragmentation
  - Need for compaction off-line (downtime) or on-line

### *Linked Allocation – (1)*

- Linked allocation allocates a linked list of blocks to a file
  - Keep a pointer to the first block of a file
  - Each block contains pointer to next block
  - File ends at nil pointer
  - Free space management system called when new block needed




- Advantages
  - No external fragmentation
  - No compaction
- Disadvantages
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks

## 14.9 Summary

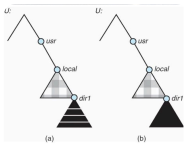
- Most file systems reside on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk, but the use of NVM devices is increasing. non-volatile memory
- Storage devices are segmented into partitions to control media use and to allow multiple, possibly varying, file systems on a single device. These file systems are mounted onto a logical file system architecture to make them available for use.
- File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices and communicating with them. Upper levels deal with symbolic file names and logical properties of files.
- The various files within a file system can be allocated space on the storage device in three ways: through contiguous, linked, or indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block. These algorithms can be optimized in many ways. Contiguous space can be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.
- Free-space allocation methods also influence the efficiency of disk-space use, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.
- Directory-management routines must consider efficiency, performance, and reliability. A hash table is a commonly used method, as it is fast and efficient. Unfortunately, damage to the table or a system crash can result in inconsistency between the directory information and the disk's contents.
- A consistency checker can be used to repair damaged file-system structures. Operating-system backup tools allow data to be copied to magnetic tape or other storage devices, enabling the user to recover from data loss or even entire device loss due to hardware failure, operating system bug, or user error.
- Due to the fundamental role that file systems play in system operation, their performance and reliability are crucial. Techniques such as log structures and caching help improve performance, while log structures and RAID improve reliability. The WAFL file system is an example of optimization of performance to match a specific I/O load.

# Chapter 15 File System Internals

- **tmpfs**—a "temporary" file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **objfs**—a "virtual" file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- **ctfs**—a virtual file system that maintains "contract" information to manage which processes start when the system boots and must continue to run during operation
- **lofs**—a "loop back" file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

The file systems of computers, then, can be extensive. Even within a file system, it is useful to segregate files into groups and manage and act on those groups. This organization involves the use of directories (see Section 14.3).

Mounts:




- Volumes containing file systems can be mounted into the computer's file-system space.
- Depending on the operating system, the file-system space is seamless (mounted file systems integrated into the directory structure) or distinct (each mounted file system having its own designation).
- At least one file system must be bootable for the system to be able to start—that is, it must contain an operating system. The boot loader is run first; it is a simple program that is able to find the kernel in the file system, load it, and start its execution. Systems can contain multiple bootable partitions, letting the administrator choose which to run at boot time.
- Most systems are multi-user and thus must provide a method for file sharing and file protection. Frequently, files and directories include metadata, such as owner, user, and group access permissions.
- Mass storage partitions are used either for raw block I/O or for file systems. Each file system resides in a volume, which can be composed of one partition or multiple partitions working together via a volume manager.
- To simplify implementation of multiple file systems, an operating system can use a layered approach, with a virtual file-system interface making access to possibly dissimilar file systems seamless.
- Remote file systems can be implemented simply by using a program such as `ftp` or the web servers and clients in the World Wide Web, or with more functionality via a client–server model. Mount requests and user IDs must be authenticated to prevent unapproved access.
- Client–server facilities do not natively share information, but a distributed information system such as DNS can be used to allow such sharing, providing a unified user name space, password management, and system identification. For example, Microsoft CIFS uses active directory, which employs a version of the Kerberos network authentication protocol to provide a full set of naming and authentication services among the computers in a network.
- Once file sharing is possible, a consistency semantics model must be chosen and implemented to moderate multiple concurrent access to the same file. Semantics models include UNIX, session, and immutable-shared-files semantics.
- network file system
  NFS is an example of a remote file system, providing clients with seamless access to directories, files, and even entire file systems. A full-featured remote file system includes a communication protocol with remote operations and path-name translation.
- General-purpose operating systems provide many file-system types, from special-purpose through general.

### NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

Just as a file must be opened before it can be used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple file-system-containing volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the mount point—the location within the file structure where the file system is to be attached. Some operating systems require that a file-system type be provided, while others inspect the structures of the device

*Page cache:* cache of virtual memory pages including memory-mapped files

*Buffer cache:* stores blocks from disk, now unified with the page cache (why?)

*Page scanner:* schedules dirty pages to be written to disk

*Dentry cache:* remembers mappings from directory entry to inode (for what purpose?)

*Inode cache:* stores inodes in hash table for fast lookup

### Performance (1)

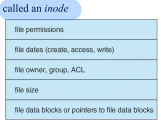
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation
  - Select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

Caches:

An inode contains:

- File attributes (time of creation, permissions, ….)
- 10 direct pointers (logical disk block ids)
- 1 one-level indirect pointer (points to a disk block which in turn contains pointers)
- 1 two-level indirect pointer (points to a disk block of pointers to disk blocks of pointers)
- 1 three-level indirect pointer (points to a disk block of pointers to disk blocks of pointers to pointers of disk blocks)

In Unix, called an *inode*




### Finding Files

- Now we know how to retrieve the blocks of a file once we know:
  - The FAT entry for DOS
  - The inode of the file in UNIX
- But how do we find these in the first place?
  - The directory where this file resides should contain this information

### NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - Searching for a file within a directory
  - Reading a set of directory entries
  - Manipulating links and directories
  - Accessing file attributes
  - Reading and writing files
- NFS servers are stateless; each request has to provide a full set of arguments (NFS V4 is different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- NFS protocol does not provide concurrency-control mechanisms

### Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

### Sun Network File System (NFS) (1)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP) and Ethernet

### Sun Network File System (NFS) (2)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
    - mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
    - files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

### Sun Network File System (NFS) (3)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services

- A race condition occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.
- A critical section is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem is to design a protocol whereby processes can synchronize their activity to cooperatively share data.
- A solution to the critical-section problem must satisfy the following three requirements: (1) mutual exclusion, (2) progress, and (3) bounded waiting. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.
- Software solutions to the critical-section problem, such as Peterson's solution, do not work well on modern computer architectures.
- Hardware support for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.
- A mutex lock provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.
- Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.
- A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
- Solutions to the critical-section problem may suffer from liveness problems, including deadlock.
- The various tools that can be used to solve the critical-section problem as well as to synchronize the activity of processes can be evaluated under varying levels of contention. Some tools work better under certain contention loads than others.
- Classic problems of process synchronization include the bounded-buffer, readers–writers, and dining-philosophers problems. Solutions to these problems can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, monitors, and condition variables.
- Windows uses dispatcher objects as well as events to implement process synchronization tools.
- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.
- The POSIX API provides mutex locks, semaphores, and condition variables. POSIX provides two forms of semaphores: named and unnamed. Several unrelated processes can easily access the same named semaphore by simply referring to its name. Unnamed semaphores cannot be shared as easily, and require placing the semaphore in a region of shared memory.
- Java has a rich library and API for synchronization. Available tools include monitors (which are provided at the language level) as well as reentrant locks, semaphores, and condition variables (which are supported by the API).
- Alternative approaches to solving the critical-section problem include transactional memory, OpenMP, and functional languages. Functional languages are particularly intriguing, as they offer a different programming paradigm from procedural languages. Unlike procedural languages, functional languages do not maintain state and therefore are generally immune from race conditions and critical sections.

### Fsck – File System Consistency Check

- Blocks:
  - for every block keep 2 counters:
    - a) # occurrences in files
    - b) # occurrences in free list.
  - For every inode, increment all the (a)s for the blocks that the file covers.
  - For the free list, increment (b) for all blocks in the free list.
  - Ideally (a) + (b) = 1 for every block.
  - However,
    - If (a) = (b) = 0, missing block, add to free list.
    - If (a) = (b) = 1, remove the block from free list
    - If (b) > 1, remove duplicates from free list.
    - If (a) > 1, make copies of this block, and insert into each of the other files.

### Log Structured File Systems

- *Log structured* (or *journaling*) file systems record each metadata update to the file system as a *transaction*
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

### NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
  - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk