6 Synchronization

Deterministic: Execution of a single sequential process always yields the same result
Race Condition: Concurrent processes race for shared resources resulting in different outcomes
Corret CS Implementation: Mutual Exclusion, Progress (use resource if no one in CS), Bounded Wait, Independence (don't block others if not in CS)
Symptoms of Incorrect Impl: Deadlock, Livelock, Starvation
Low Level Impl: Atomic Test and Set with busy waiting
Semaphore: $S_{initial} \geq 0$, $S_{current} = S_{initial} + \#signal(S) - \#wait(S)$
General semaphore (mutex) – $S \geq 0$
Binary semaphore – $S = 0$ or 1
$N_{CS} = \#Wait(S) - \#Signal(S)$ (completed wait but not signal). $S_{current} + N_{CS} = 1$
Peterson's Algorithm: Use a `turn` variable with busy waiting. Cons: waste CPU, not general enough, low level
Producer Consumer: Produce only when buffer not full, consume only when buffer not empty. Use `notFull`, `notEmpty` semaphores (no busy waiting)
Readers Writers: Writers write alone to D, readers can read together. Use `roomEmpty` before writing/reading, mutex to incr/decr `nReaders`
Dining Philosophers: *Tanenbaum's Solution*: When a philosopher wants to eat `(pickup)`, they become `HUNGRY` and check if neighbors are not eating via `test(i)`. If it's safe, the philosopher starts eating; otherwise, they wait. After eating, `putdown` is called, marking the philosopher as `THINKING` and checking if either neighbor can now eat. *Limited Eater*: only 4 allowed to sit and eat, so deadlock is impossible. Once done eating, signal `seat` semaphore

7 Memory Management

Role of OS: Allocate memory for new process, manage memory for current process, Protect process memory, Provide syscalls to process
Problems of no Memory Abstraction:
1. No protection between processes – can write to other process' memory space
2. External Fragmentation – costly to relocate everything
Logical Address: How the process views its own memory space (might not coincide with physical memory). Use base + limit register. Actual address = base + offset, check against limit
Fixed Partitioning: Fixed block size regardless for all processes. Easy to manage and fast to allocate. Cons: Internal fragmentation (not using all allocated space)
Dynamic Partitioning: Pros: flexible and remove internal fragmentation. Cons: external fragmentation (holes between processes), need to merge. More information must be stored
Algorithms: First Fit, Best fit, Worst Fit
Implementations: Bitmap, Linked list
Compaction (Merging): Merged linked list nodes. Bitmap – no need to merge
Implementations: Bitmap, Linked list
Buddy System: Keep an array of linked list for each block size
Allocation: 1. Find smallest S that fits the process. 2. If free block exists at S, allocate. 3. Else, search for smallest R from S+1 to K such that R has a free block. 4. Split block at R into two blocks R-1, repeat until split blocks for level S, then go back to step 1
Deallocation: 1. Find block and it's buddy. 2. If buddy is free, remove block and buddy, merge into block of size S+1. Repeat until buddy doesn't exist or is not free. 3. Mark current block as free

8 Disjoint Memory Schemes

Paging: Physical memory split into *physical frames*. Logical memory split into *logical pages* of the same size 2^n. Logical page is loaded into any available frame.
Page Table: Maps page to frame. Each process *has its own page table*. Active process' pages are stored in RAM. Blocked process may have their pages swapped out to disk.
PCB (stored in kernel space in RAM) stores pointers to page tables' starting address
Address Translation: Physical Address = Frame * 2^n + Offset. m bit logical address, n LSB for offset, $m - n$ MSB for frame no.
Pros of Paging: No ext. fragmentation (no uneven holes), no int. fragmentation (except last page)
Cons of Paging: Require two memory accesses for each memory reference
Translation Lookaside Buffer (TLB): Hardware context; cache a few page table entries. *TLB Hit*: Frame number used to generate physical address. *TLB Miss*: Access main memory to access the full page table + Retrieve frame number + Calculate physical address + Update TLB
During context switch, TLB is flushed. So when a process resumes, initially many misses
Memory Protection: Each page table entry has *Access-Right* bits (R/W/E) and a *Valid* bit (prevent process from accessing out-of-range pages)
Page sharing: Several processes can point to the same frame e.g. shared library, implement copy-on-write (Parent-Child)
Segmentation: Split logical memory into segments (Text/Data/Heap/Stack). Each segment has a ID and Limit. Memory reference is (Segment Id, Offset). ID = segment's index in segment table
Each segment is mapped to a segment table entry (Base, Limit). Offset < limit to check for valid access. Segment table is kept in RAM.
Pros of Segmentation: Each segment is an independent contiguous space, can grow/shrink/be shared/protected independently
Cons of Segmentation: Requires variable size contiguous memory, may cause external fragmentation
Segmentation with Paging: Each segment is further split into pages. Each segment has its own page table (kept in RAM).
Memory reference is (SegmentId, Page No, Offset). SegmentID is used to index segment table. Segment table entry is (Page Limit, Page Table Base Addr). Valid access if Page No < Page Limit. Use base addr + page no to find entry in page table. Page table entry is (Frame No). Frame no is added with offset to find physical address
Pros of S+P: No external fragmentation

9 Virtual Memory Management

Motivation: Logical address space may be bigger than physical address space. Use secondary storage. Virtual memory space = Main memory + Secondary memory
Page types: Memory Resident (in RAM), non-memory resident (in secondary storage). Use *is memory resident* bit in each PTE to track. CPU can only access memory resident pages. Page fault when accessing non-resident page
Page Access Steps: 1. Check page table if page X is resident. If yes, done. 2. Else, Page fault, trap to OS. 3. Locate X in secondary storage. 4. Load X into RAM. 5. Update Page Table and go to step 1
Thrashing: Secondary storage access time >> RAM access time. *Thrashing* is when memory access frequently leads to page fault
Temporal locality: A page just used is likely to be used again in the near future
Spatial Locality: A page contains contiguous memory locations that are likely to be accessed again
Pros of Virtual Memory: 1. No longer limited by physical memory. 2. More efficient use of physical memory (pages not in use can be stored on secondary storage). 3. More processes can reside in memory
Demand Paging: Process starts with no memory resident pages (All pages in disk). Only load a page into RAM during page fault. Motivation: some processes have many pages, initialization is costly
Pros: Fast start up for new processes, small memory footprint (more processes can be memory resident)
Cons: Process may appear sluggish at the start due to multiple page faults. Page fault may have cascading effect i.e. thrashing
Page Table Structure: Modern computers provide huge logical memory spaces, leading to huge page tables. If huge page tables are kept in RAM, leads to high overhead and fragmented page table
Direct Paging: Keep all entries in a single page table. 2^p page table entries, p bits to specify one page.
2-Level Paging: Split page table into outer and inner page table. m MSB on outer page table to identify 2^m smaller page tables. Each smaller page table uses $p - m$ LSB to address 2^{p-m} PTE. Outer page table is called *page directory*. Inner PTE stores Frame Number, add offset to get physical addr
Inverted Page Table: M processes in memory = M page tables. Only N frames can be occupied. $N <<$ overhead of M page tables. Instead, map each frame to (PID, Page No.) Pros: huge space savings. Cons: slow translation (to check if page X is in memory, need to search whole table)
Page Replacement Policy: No free frame during page fault → evict existing page from frame. If page is clean (not modified) no need to write back. If page is dirty → need to write back.
Memory Access Time: $T_{access} = (1-p) * T_{mem} + p * T_{page\ fault}$. p: probability of page fault, T_{mem}: access time for memory resident page, $T_{page\ fault}$: access time for page fault
Optimum (OPT): Replace page that will not be used again for the longest time. Not realizable (requires future knowledge), but good for comparison. Closer to OPT = better
FIFO: evict oldest page (added earliest). Use a queue, evict head, update queue during page fault. No hardware support needed. *Belady's Anomaly*: #frame increases but #fault increases, as FIFO does not exploit temporal locality
LRU: Replace page that has not been used in the longest time. Exploits temporal locality, does not suffer from Belady's Anomaly. Need substantial hardware support. Using *Time Counter*: Increment t when page is used. Cons: Time of use is forever increasing, may overflow. Need to search through all pages to find minimum. Using *Stack*: If X is referenced, remove X from stack, push to top. To replace, remove bottom of stack. Cons: not a real stack, hard to implement in hardware.
Second Chance (CLOCK): Each PTE maintains *accessed* bit. Oldest FIFO page is selected. If *accessed* is 0, replace page. Else, give second chance, set *accessed* to 0, move to next page in queue. When all *accessed* bits are 1, degenerates to FIFO.
Frame Allocation Policy: N frames, M processes. *Equal Allocation*: each process gets N/M frames. *Proportional Allocation*: each process gets $size_p / size_{total} * N$ frames
Local Replacement: Process can only replace its own pages. Each process is allocated a fixed no. of frames. Pros: performance is stable. Cons: performance may be hindered by limited frames
Global Replacement: One process can replace page of any other process. Pros: self-adjustment (process can take more pages from other processes). Cons: performance may differ, badly behaved processes can steal many pages
Thrashing: Insufficient physical frames → high overhead to load non-memory resident pages. Global replacement can cause *cascading thrashing. Local replacement* can lead to thrashing within one process
Working Set Model: The set of pages (i.e. *locality*) used by a process is relatively constant in a period of time. Working Set Window Δ = an interval of time. $W(t, \Delta)$ = active pages in the interval at time t. Allocate enough frames for pages in W to avoid page faults. *Accuracy* of working set model is directly affected by the choice of Δ. Too big: may

contain pages from different locality. Too small: may miss pages from current locality.

10 File Management

Motivation: Physical memory is volatile → use external storage to persist data. Direct access to storage media is not portable → use file system as an *abstraction* on top of physical media. File system provides *high level resource management*, *protection* and *sharing* between processes and users
Criteria: *Self-contained* (plug and play), *Persistent* beyond the lifetime of OS and processes, *Efficient* (good management of free and used space, minimum overhead for bookkeeping)
File: A logical unit of information. Contains *data* and *metadata*.
Metadata: Name, id, Type (exe, txt, directory etc.), Size (in bytes/words/blocks), Protection (access permissions), Time/date/owner, Table of content
File name: Different FS have different naming rules. Common rules: length of file name, case sensitivity, allowed special symbols, file extension
File Types: Regular files (user information), Directories (FS structure), Special files. Two major types: ASCII (text), Binary executable e.g. pdf, mp3
Distinguishing File Types: Use extension e.g. docx, or embedded information e.g. magic numbers in Unix
File Protection: Read/Write/Execute (load into memory and run), Append, Delete, List (read metadata). Most common approach: restrict based on user identity.
Access Control List: list of users and their allowed access types. Pros: customizable. Cons: too much overhead
User Groups: Owner/Group/Universe (`ls -l` to see permission bits, `getfacl <file>` to get file access control lists)
Operations on File Metadata: Rename, Change attributes (access permission, date, ownership), Read attributes (e.g. file creation time)
Structure of File Data: 1. *Array of bytes*: Each byte has unique offset from file start. 2. *Fixed Length records*: Array of records, can grow/shrink. Offset of n^{th} record = size of record * $(n-1)$. 3. Variable length records: flexible but harder to locate a record
Access Methods: *Sequential* (read in order), can't be rewound. *Random access*: read in any order, `read(offset)`, `seek(offset)`. *Direct access*: used for fixed-length records. Allows random access to any record directly. Useful when there is large amount of records.
File Operations: Create, Open, Read, Write, Reposition, Truncate (remove data from specified position to end of file)
System Calls: OS provides file operations as system calls. Information kept for opened file: 1. current location in file (file pointer), 2. location of file on disk, 3. open count (how many times opened)
System-wide open-file table: One entry per unique file
Per-process open-file table: Each entry points to system-wide table
Process File Sharing: Two processes using different file descriptors: I/O can happen at independent offsets. Using same file descriptor: I/O changes offset for the other process
Directory: Logical grouping of files (user view), Keep track of files (system usage). Directory structures: Single level, Tree-structure, Directed Acyclic Graph (DAG), General graph
Tree Structure: Directories are recursively embedded. Uses relative and absolute paths
DAG: Can have hard link `ln`/symlink `ln -s` to files. Hard link: a separate pointer, for files only. Faces problem with deletion. Symlink: can be to directory or file. Creates separate file with path of original file. More info overhead but easier deletion
General graph: Not desirable. Hard to traverse, need to prevent infinite loop. In Unix, symlink to directory can be created to create general graph

11 File System Implementations

Disk Structure: 1D array of logical blocks (smallest accessible unit, usually 512bytes to 4KB). Logical block is mapped to disk sector.
Disk Organization: Master Boot Record (MBR) at sector 0, followed by one or more partitions. Each partition can contain an independent file system. A file system generally contains: Boot sector, partition details (no. of blocks etc.), directory structure, file information, actual file data
File Implementation: A file is a collection of logical blocks. When file size != multiple of logical block, last block has internal fragmentation. A good file impl must keep track of logical blocks, allow efficient access, utilize disk space effectively
Contiguous Block Allocation: Allocate consecutive disk blocks to file. Pros: simple to keep track, fast access (only need to seek to first block). Cons: external fragmentation, file size needs to be known in advance.
Linked List Allocation: Store pointer to next disk block. Pros: solves fragmentation. Cons: random access is slow.
File Allocation Table (FAT): in memory at all times. Stores next block pointed to by each disk block. Pros: fast random access. Cons: FAT can be huge
Indexed Allocation: Each file has an indexed block, which contains an array of disk block addresses. Pros: less memory overhead, only index block of opened files need to be stored in memory. Fast direct access. Cons: Limited max file size, no. of index block entries = max no. of blocks, index block overhead
Free Space Management: *Bitmap* – free disk block represented by 1, occupied 0. Pros: bitwise manipulation. Cons: need to keep in memory. *Linked list* – Each disk block keeps a pointer to next free disk block. Pros: easy to locate free block, only pointer to first block is kept in memory (others can be cached). Cons: high overhead
Directory Structure:
Linear List: Each node in the list represents a file. Store file name and other metadata, and file information (or pointer to it). Linear scan to search for a file in a directory is inefficient for deep trees but searches can be cached.
Hash Table: of size n. Filename is hashed and the entry is checked against the table. Collisions resolved by chaining. Pros: fast lookup. Cons: Limited hashtable size, need good hash function
File Information: Filename, Metadata, Disk block information. Either store everything in directory entry, or store pointers **File Creation**: 1. Use full pathname to locate parent dir, search for file in dir. If found, error. (Searched can be cached) 2. Find free disk space, 3. Add entry to parent dir with relevant file information
File Opening: 1. Search system-wide table for existing entry E. If found return it. 2. Use full pathname to locate F. If not found, error. Else load F into the system-wide table into entry E. 3. Create an entry in P's table to point to E, return the pointer

12 File System Case Studies

Windows FAT
Layout: MBR (Sector 0) | Partn 1 | Partn 2 ...
Partition Information: Volume Boot Record | FAT | FAT duplicate | Root Directory | Data Blocks FAT entry contains: FREE (0x000), Block number (0x002 – 0xFEF), EOF, BAD (0xFF7)
Directory Structure: Directory is a special type of file. Root directory is stored in its own location, other directories are stored in data blocks. Each file/subdirectory in the folder is represented as a directory entry
Directory entry fields: File Name (8) | Extension (3) | Attribute (1) | Reserved (10) | Creation Date/Time (2/2) | First Disk Block (2) | File Size in Bytes (4). *First byte* of file name may have special meaning. Creation time accuracy of second is ±2 sec, Year is limited to 1980-2107. FAT variants (12/16/32) use 12/16/32 bits for first block index
Cluster: Contiguous disk blocks. Larger cluster → larger usable partition, but larger internal fragmentation. On FAT32, only 28 bits are usable. Actual usable partition size is smaller due to special values (EOF, FREE etc.) occupying some blocks
Long File Name Support: Using virtual FAT (VFAT). Support file names up to 255 characters. Uses multiple directory entries. 8+3 version kept for backwards compatibility
Mounting a partition: Making a file system accessible. E.g. `mount /dev/sdb1 /mnt/media`. Can be done at runtime on new devices e.g. USB
Formatting a partition: Writes the file system structures (prepare a partition to hold a specific file system). Will overwrite previous file structure.
Linux Extended-2 (ext-2)
Layout: MBR | Block Group 0 | Block Group 1 ...
Block Size: 1/2/4/8 KB (configurable)
Block Group Layout: Superblock | Group Descriptors | Block bitmap | Inode bitmap | Inode table | Data blocks. For ext-2 revision 0, every block group has a superblock and a group descriptor table (for redundancy), but from revision 1 onwards, only block group 0 and 1 has
Superblock: Describes the **whole** file system. Includes total inode count, inodes per group, total disk blocks, etc.
Group Descriptors Table: Each entry in the table describes one block group: location of bitmaps/inode table, information on free space etc.
Block bitmap/Inode bitmap: 0 = free space, 1 = occupied
Inode table: An array of inodes of that block group. Each inode can be accessed by an index
Inode Structure: Mode (file type i.e. file/directory/link, owner, access permission) (2 bytes), user id (2), size of data (4), access/creation/modifed/deleted time (4/4/4/4), group id (2), number of hard links (2), Behaviour flags (4), `i_block`
I-block: 12 direct block pointers, 1 indirect BP, 1 double indirect BP, 1 triple indirect BP. Total 15x4=60 bytes. For 1KB blocks, indirect BP points to a block with 256 direct block pointers. The first data block for a file is at the first direct pointer
Directory: Treated as a special type of file. Identified by the `mode` of the inode. The data block of a directory is a linked list storing the entries of the directory table.
Directory Entry: Each entry in the table has an inode index, file name, length of file name, length of the entry, and padding (to align to 4 bytes). Entry length (`rec_len`) tells you how much to jump to the next entry. Note: `ls` prints the inode numbers, not the actual inodes
FAT vs ext-2: All block groups are 'mounted' and accessible in ext-2, whereas only one partition is initially mounted in FAT. The goal of block groups is to keep files and their inodes close together on disk to reduce seek times.
File Deletion: Block bitmap, inode bitmap, inode table, group descriptor needs to be updated
Hard Link: Hard link increases the reference count for an inode. The directory entry directly refers to the inode that is being linked. Ref count is decremented for every deletion (if reach 0, can delete the inode)
Symbolic Link (symlink): A new inode and file is created for the symlink. The content of file is the path to the file being linked. It does not increase the reference count for the original file. Symlinks can be invalidated if the pathname changes or the file is deleted.