Synchronization

Properties of correct critical section implementation

- Mutual exclusion (only one process in CS)
- Progress (if no process in CS, one of the waiting process should be granted)
- Bounded wait (after a process requests to enter CS, there should be a limited number of times other processes can enter CS before it)
- Independence (process not executing in CS should not block other process)

Incorrect synchronization implementation

- Deadlock
- Livelock (processes change state; make no other progress)
- Starvation (some processes blocked forever)

CS IMPLEMENTATION

Assembly level implementation

- `TestAndSet Register, MemoryLocation`
- Atomic machine instruction
- Employs busy waiting (keep checking condition)

High level language implementation

- Maintain a `Want` array (elem = if a process wants to go CS)
- Maintain a `turn` variable (which process's turn)
- Writing to turn is an atomic operation
- Busy waiting
- Low level programming construct
- Not general synchronization mechanism

High level abstraction implementation

- Semaphore (provides a way to block a number of processes – sleeping processes, and a way to unblock)
- Wait(S) – if S <= 0, blocks; decrement S
- Signal(S) – increments S; wakes up one sleeping process
- Given $S_{initial} >= 0$, $S_{current} = S_{initial}$ + #signal(S) - #wait(S)
- Number of signals() ops executed
- Number of wait() ops completed
- Usage: wait(S); critical section; signal(s)
  - Commonly known as mutex
- Alternative: conditional variable
  - Allow a task to wait for certain event first

SYNCHRONIZATION IMPLEMENTATION

- POSIX semaphore
  - Initialize a semaphore, perform wait() or signal()
- pthread mutex
  - Lock: pthread_mutex_lock()
  - Unlock: pthread_mutex_unlock()
- pthread conditional variable
  - Wait: pthread_cond_wait()
  - Signal: pthread_cond_signal()
  - Broadcast: pthread_cond_broadcast()

Memory Abstraction

Memory usage

- Transient data: params, local vars
- Persistent data: global var, constant var, dynamically alloc
- Both data sections can grow/ shrink during execution

CONTIGUOUS MEMORY MANAGEMENT

- Each process occupies a contiguous memory region
- Physical memory is large enough to contain >=1 with complete memory space

Memory Partitioning

- Fixed-size partitions: physical memory spit into fixed number of partitions of equal sizes
  - Leftover space wasted: internal fragmentation
- Variable-size partitions: partition is created based on actual size of process
  - Large number of holes: external fragmentation

Allocation Algorithms

- First-fit: take the first hole large enough
- Next-fit: take the hole from last allocated block
- Best-fit: take the smallest hole large enough
- Worst-fit: take the largest hole
- To reduce external fragmentation: **merge** – freed partition with adjacent hole, **compaction** – move occupied partitions to create bigger, consolidated holes

Buddy System

- Provides efficient:
  + Partition splitting
  + Locating good match for a free partition
  + Partition de-allocation and coalescing
- Free block is split into half repeatedly to meet request size
- When buddy blocks are both free, merged to form larger

1. Find smallest S, such that $2^S >= N$
2. Access A[S] for a free block
3. If free block exists, remove from free blk list, allocate blk
4. Else, find smallest R from S+1 where A[R] is free
5. For (R-1) to S, repeatedly split free block and go to 2

DISJOINT MEMORY SCHEMES

- Physical address = frame_number x `sizeof`(physical_frame) + offset
- Offset: displacement from beginning of physical frame

Page Table

- **Paging**
  - Split the logical address into fixed size pages
- TLB: cache for the page table entries
- Access right bits: each page table entry has w, r, x bits
- Valid bit: bit to indicate whether page is valid to access
- **Page sharing**
  - Copy-on-write: parent child process share a page until one tries to change a value in it
- **Segmentation scheme**
  - Split the logical address into variable size segments according to their usage
  - Each memory segment has a name and a limit
  - *Logical address <SegID, Offset>*
  - SegID is used to look up <Base, Limit> of segment in segment table
  - All memory references specified as: seg. name + offset
  - + segment is an independent contiguous memory space
  - + segments grow/shrink and be protected independently
  - - requires variable-size contiguous memory regions: can cause external fragmentation
  - Important process are given more lottery tickets
  - **Physical Address = Base Address of Segment + Offset**
  - *Offset < Limit for valid access*
  - *Physical address = Base + Offset*

VIRTUAL MEMORY MANAGEMENT

- Secondary storage capacity >> physical memory capacity
- Some pages are accessed much more often than others

Extended Paging Scheme

- Two page types
  - Memory resident (pages in physical memory)
  - Non-memory resident (pages in secondary storage)
- CPU can only access memory resident pages (page fault)

Accessing Page X

1. Check page table
   - Is page X a memory resident?
   - Yes: accessed in physical memory
2. Page fault: OS takes control
3. Locate page X in secondary storage
4. Load page X into a physical memory (use replacement algo if there is no more space in physical memory)
5. Update page table
6. Go to step 1 to re-execute the same instruction

- If memory access leads to page fault a lot –> thrashing
- Locality principles
  - Temporal: mem. address used now likely used again
  - Spatial: mem. addresses close to address likely used soon
- Demand paging
  - Process start with no memory resident page
  - + Fast startup time for new process
  - - Appear sluggish at the start due to page faults

Page Table Structure

- Direct paging
  - Keep all entries in a single page table
- 2-level paging: page the page table
  - Process may not use entire virtual memory space
  - Original page table has 2^P entries
  - With 2^M smaller page tables, M bits needed for page tables
  - Smaller page tables have $2^{(P-M)}$ entries each
- Inverted page table
  - Keep a single mapping of physical frame to <pid, page#>
  - Page table is a per-process information: with M processes in memory, there are M independent page tables
  - Only N physical memory frames can be occupied
  - Out of M pages tables, only N entries are valid
  - Huge waste if N << overhead of M page tables

PAGE REPLACEMENT ALGORITHMS

- No free physical memory frame during a page fault
- Clean page: not modified (no need to write back)
- Dirty page: modified (need to write back)
- $T_{access} = (1 - p) * T_{mem} + p * T_{page\_fault}$
- p = probability of page fault
- T_{mem} / T_{page_fault} = access time for mem resident/ page fault

Optimal Page Replacement (OPT)

- Replace the page that will not be needed again for the longest period of time
- Guarantees minimum number of page faults
- Need future knowledge of memory references

FIFO Page Replacement

- Memory pages are evicted based on their loading time
- Maintain a queue of resident page numbers
- Belady's Anomaly (more frames → more page faults)

Least Recently Used (LRU)

- Make use of temporal locality, replace the page that has not been used in the longest time
- Implemented with counter
  - "Time" counter is incremented with every memory ref.
  - Need to search through all pages
  - Time of use is forever increasing
- Implemented with a stack
  - Replace the page at the bottom of the stack
  - Not a pure stack: entries can be removed from anywhere

Second-Chance Page Replacement (CLOCK)

- Modified FIFO to give a second chance to pages
- PTE maintains a reference bit which is decremented to 0
- Algorithm continues from the latest victim page referenced
- Maintain a circular list of all pages, and a pointer to next potential victim page
- Degenerate into FIFO algorithm if all pages has ref bit = 1

FRAME ALLOCATION

- Equal alloc.: each process gets N/M frames
- Proportional alloc.: processes get $size_p/size_{total}$*N frames
- Local replacement: victim page selected among pages of the process that causes page fault
  - + Number of frames for process is constant
  - - If insufficient frames allocated → hinder progress
- Global replacement: victim page selected among all physical frames (process P can take frame from process Q)
  - + Allow self-adjustment between processes
  - - Badly behaved processes can affect others
- Insufficient physical frame → thrashing
- Working set model: models memory usage of processes
  - Transient region: working set changing in size
  - Stable region: working set about the same for a long time

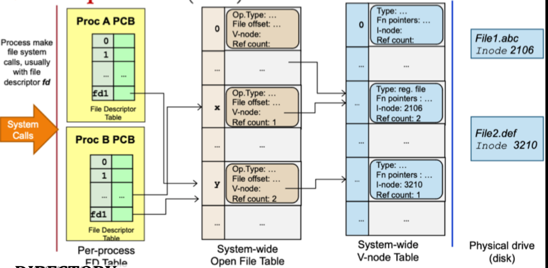
File Management

FILE SYSTEM CRITERIA

- Self-contained: information stored on media enough to describe the entire organization
- Persistent: beyond the lifetime of OS and processes
- Efficient: good management of space, minimum overhead

FILE SYSTEM ABSTRACTION (abstraction for hard disk)

- File type
  - Regular files: contains user information
  - Directories: system files for FS structure
  - Special files: character/ block oriented
  - Distinguished with file extension/ embedded information (magic number at the beginning of file in Unix)
- Ops on file metadata: rename, change/ read attributes
- File protection
  - Permission (r/w/x) bits for owner, group, universe
- File Data: Structure
  - Array of bytes: each byte has a unique offset from start
  - Fixed length records: array of records, can jump to any
  - Variable length records: flexible but harder to locate
- File Data: access methods
  - Sequential : data read in order, cannot be skipped
  - Random : data can be read in any order; read(offset) to access; seek(offset) to move
  - Direct: used for fixed-length record, rand. access allowed
- File related Unix System Calls:
  - open(), read(), write(), lseek(), close()
- File information kept for an opened file
  - File pointer, file descriptor, disk location, open/ ref. count
  - Uses 3 tables: per-process open-file table, system-wide open-file table, system-wide v-node table



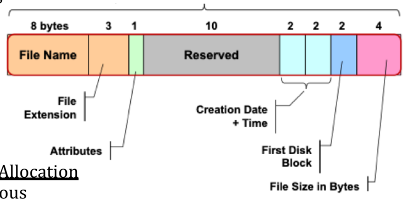
DIRECTORY

- Single-level: all files in same directory
- Tree-structure: directories can be recursively embedded in other directories
  - Absolute pathname
  - Relative pathname
- Direct Acyclic Graph: only one copy of actual content
  - Unix: hard link (`ln`) [not allowed for directories]
    - + Low overhead, only pointers are added in directory
    - - Deletion problems
  - Unix: soft link (`ln -s`) special link file independent of file (can dangle when file is deleted or renamed)
    - + Simple deletion, - Larger overhead
- General Graph: cyclic directories can be linked
  - - Need to prevent infinite looping

FILE SYSTEM IMPLEMENTATION

- Disk organization: master boot record at sector 0

Directory entry in FAT 16:



File Block Allocation

- Contiguous
  - Allocate consecutive disk blocks to a file
  - Good for sequential and random access
  - + Simple to keep track, fast access
  - - External fragmentation, file size needs to be specified in advance
- Linked list
  - Of disk blocks which stores next block number, file data
  - + Solve fragmentation problem
  - - Random access in a file is very slow
- FAT Allocation
  - Entry contains either FREE, next block #, EOF, BAD
  - + Faster random access
  - - Keeps track of all disk blocks in a partition (huge)
  - Size of FAT16 = 2^{16} * 16 bits of space = 2^{17} B
  - Runtime overhead: entire size of FAT is in memory

Indexed Block Allocation

- Maintain blocks for each file
- + Lesser memory overhead
- - Limited max. size (max # of blocks = # of index blk entry)
- Linked scheme
  - Keep a linked list of index nodes (ex. traversal cost)
- Multilevel index
  - Similar idea as multilevel paging
- Combined
  - Combination of direct indexing and multilevel index

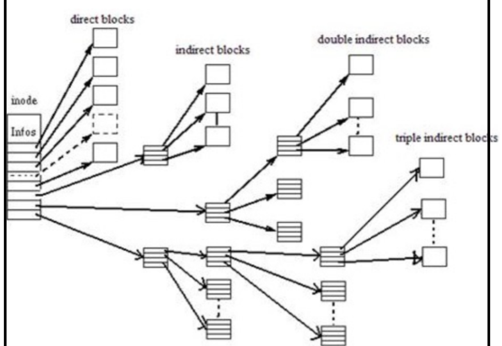
Free Space Management

- Maintain free space information
- Allocate: remove free disk block from free space list
- Free: Add free disk block to free space list
- Bitmap
  - + Provides a good set of manipulation
  - - Need to keep in memory for efficiency reason
- Linked list
  - Each disk block contains number of free disk block numbers or a pointer to the next free space disk block
  - + Easy to locate free block
  - - High overhead

Implementing Directory

- Keeps track of files in a directory
- Map the file name to the file information
- Linear list
  - Each entry represents a file
  - Requires linear search to locate a file
- Hash table
  - Each directory contains a hash table of size N
  - + Fast lookup
  - - Hash table has limited size
  - - Depends on good hash function
- File information consists of:
  - File name and other metadata
  - Disk blocks information
- Two common approaches to store
  - Everything in a directory entry
  - Only file name and point to other data struct for info

I-Node Structure



For a 4 bytes block address and disk block size of 1 KiB: With 12 direct blocks, 1 single, 1 double and 1 triple indirect blocks,

Direct blocks: 12 * 1KiB = 12 KiB

Single indirect:
Number of entries = 1 KiB / 4 B = 256
Total possible file size = 256 * 1 KiB = 256 KiB

Double indirect: 256^2 * 1KiB = 64 MIB
Triple indirect: 256^3 * 1KiB = 16 GiB

File Descriptors and Processes

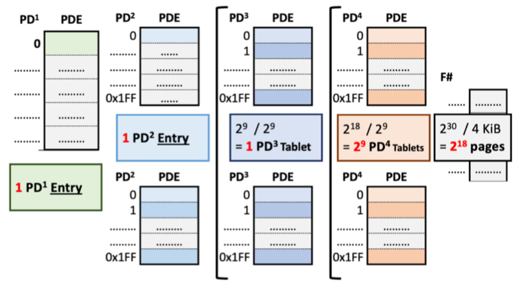
- Opening the same file twice leads to 2 entries in FD table
- Fork duplicates FD and parent and child has same offset

Calculation Questions

Virtual address: 48 bits long
Physical frame size: 4KiB
Each table/ directory entry size: 8 bytes

Maximum memory space is 2^{48} bytes
Number of pages = 2^{48} / 2^{12} = 2^{36} pages

If we are keeping the "page directory" at each level to a single page, the branching factor is 2^{12} / 8 = 2^9 → each directory can point to 512 next level directories



Virtual address: 16 bits long
Page Size: 32 bytes = 2^5 bytes
PTE Size: 8 bytes = 2^3 bytes

Page offset = 5 bits
Bits left for multi-level paging: 16 – 5 = 11 bits
Number of rows in PT not in root level = 2^5 /2^3 = 2^2
→ Bits for each level of paging = 2 bits
If there are 3 levels, then bottom 2 levels = 2 bits each
Bits left for root level page table = 11 – 2(2) = 7 bits
→ Number of entries in root level = 2^7 bytes
Size of root level in terms of page = 2^7 * PTE Size / Page Size
= 2^7 * 2^3 / 2^5 = 2^5

Virtual address space: 2^{64} bits = 2^{61} B
Physical memory (RAM): 32 KiB = 2^{15} B
Page size: 1 KiB = 2^{10} B
PTE size: 2 B

Number of virtual pages = 2^{61} / 2^{10} = 2^{51}
Number of physical pages = 2^{15} / 2^{10} = 2^5
Size of standard page table = 2^{51} * 2 = 2^{52} B
Size of inverted page table = 2^5 * 2= 2^6 B

---

32 bits in 1-byte addressable system: 2^{32} bytes of mem space
32 bits in 8-byte addressable system: 2^{32} * 2^3 = 2^{35}

For a paged memory reference, there will be 2 memory accesses
1 for retrieving the frame number in page table in memory and another for the actual data in memory.

To map a file to memory, it needs to be smaller than the virtual address space on the machine.

When there is no more space for the heap to expand, find a new block of suitable size and copy all of heap over.

Quick access to common numbers

4KiB = 2^{12} bytes; 8 bits = 1 bytes
2048 = 2^{11}; 4096 = 2^{12}; 512 = 2^9

- Reader Writer

```
// Writer
while (TRUE) {
    wait(roomEmpty);
    // modify data
    signal(roomEmpty)
}

// Reader
while (TRUE) {
    wait(mutex);
    nReader++;
    if (nReader == 1) {
        wait(roomEmpty);
    }
    signal(mutex)

     // read data

    wait(mutex);
    nReader--;
    if (nReader == 0) {
        signal(roomEmpty);
    }
    signal(mutex)
}
```

Physical frame size = 4KiB
4 Level Page Table each with 9 bits (2^9 entries)
`Memory usage: 1GiB = 2^30`

Logical Address to PA Translation

$2^{\text{Number of page offset bits}}$ = Frame/Page Size
Logical Address: XXXX PPPP (PPPP is page offset)
-> Translate XXXX to frame number YYYY
Physical Address: YYYY PPPP

The only difference between dynamic allocation and pure segmentation is that the segmentation divides the process into its 4 respective memory spaces (text, data, heap, stack) whereas in a dynamic allocation, the entire process itself is contiguous.
Therefore, for both types of allocation, there is only external fragmentation

Paging divides memory into fixed-size blocks called pages, while Segmentation divides memory based on data type or function into variable-sized segments

Pure paging:
- Has internal fragmentation (due to page size)

Paging with segmentation:
- Has less internal fragmentation than pure paging
- Has more external fragmentation than pure paging

Pure segmentation:
- Can cause external fragmentation
- No internal fragmentation

Segmentation with paging:
- Has internal fragmentation (due to page size)