Deadlock

**Approaches to handle deadlock**:

1. Ignore the problem
2. Use a protocol to prevent or avoid deadlocks
3. Allow deadlocks, detect them, and recover

**Necessary conditions and solutions**:

1. Mutual exclusion (at least one resource can't be shared)
2. Hold & wait (thread must be holding at least one resource) 🡪 *prevent holding*
3. No preemption (resources not preempted) 🡪 *allow preemption*
4. Circular wait (set of threads are circularly dependent) 🡪 *banker's algorithm*

**Mutex** (acquire() & release() ): This is how test & set works. Test a lock 🡪 execute

**Semaphore**

|  |  |
| --- | --- |
| Producer | Consumer |
| wait(&empty) | wait(&full) |
| acquire() | acquire() |
| … | … |
| release() | release() |
| sig(&full) | sig(&empty) |

Paging

**High bits** = page #

**Low bits** = page offset

**# entries** = 2high - low

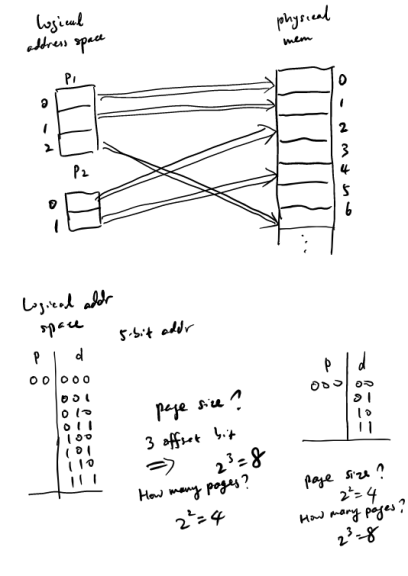
**Logical(Virtual) address space** = # of pages \* offset

**Physical address space** = # of frames \* offset

**Page size** = 2n Bytes

**Offset** = n (taken from page size)

**# of virtual pages** = Virtual address space/offset



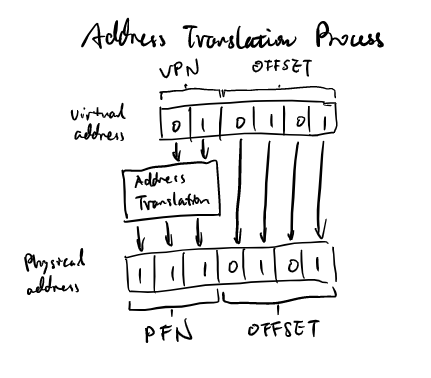
**TLB**: A special high-speed memory cache used to increase page lookup speed. Hash

**Demand Paging**: Only load pages when we need them. Fault when a page must be loaded

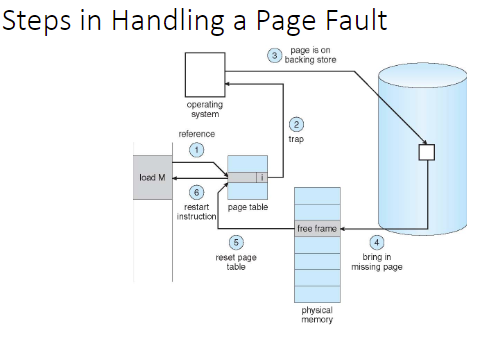
**Hashed Paging**: Used for handling addresses larger than 32-bit

**Inverted Paging**: Only store real addresses.

**Address Translation**:



**Page Fault**:



Memory Management

**Virtual Memory**: Memory that doesn't exist but can be used to store useless code. Allows more programs to run concurrently and less I/O needed to swap between programs

**Static Allocation**: OS assigns a process an address when loaded. Once the process is assigned, the OS cannot move it. Process can be moved before it's assigned.

**Dynamic Allocation**: OS can move the address of a process after is assigned (during runtime). Accomplished by using a relocation register whose info is appended to the virtual address of a process.

*Pros*: OS can move processes. Allows processes to grow. Simple and fast

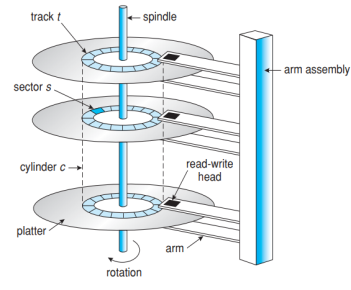
*Cons*: Slows down hardware due to the add. No memory sharing. Still limited to physical memory size. Complicates memory management and multiprogramming.

**Address Binding:**

* *Compile time*: for absolute code that can always inhabit the same address
* *Load time*: for code where address isn't known at compile time
* *Execution time*: For code whose address may change during execution

**Logical(Virtual) Address**: A memory address generated by the CPU instead of the MMU

**Effective Access Time**: (1-pfault)\*memoryaccess + pfault\*(faultoverhead + memoryaccess)



**Seek Time**: the time necessary to move the disk arm to the desired cylinder

**Rotational latency**: the time for the desired sector to rotate to the disk head

**Data Transfer**: the rate at which data flows between the drive and the computer

**Access time**: LI/O = Lseek + Lrotate + Ltransfer

**Ltransfer** = 1/Transfer rate \* size of data transferred

**FCFS**: First Come First Serve

*Pros*: intrinsically fair

*Cons*: not the fastest service

**SSTF**: Shortest Seek Time First selects the request with the minimum seek time from the current head position.

*Pro*: low rotational latency

*Con*: can cause starvation

**SCAN**: The disk arm starts at one end of the disk, and moves to the other end, servicing requests until it reaches then end in which it reverses.

*Con*: only good if requests are uniformly dense, but if most requests are at one end of disk then lots of delay

**C-SCAN**: same as scan but when it reaches one end of disk it goes back to the other end instead of reversing. Treats cylinders like a circular lists.

**LOOK & C-LOOK**: a version of SCAN and C-SCAN that only goes to the last request instead of all the way to the end of the cylinder.

**Which is best?**: SCAN & C-SCAN best for disks under constant heavy load. SSTF & LOOK are good choices for default algorithms.

|  |  |
| --- | --- |
| Unit | Value |
| 1KB | 210 Bytes |
| 8KB | 213 Bytes |
| 1MB | 220 Bytes |
| 8MB | 223 Bytes |
| 1GB | 230 Bytes |
| 8GB | 233 Bytes |

Types:

**Text File**: a sequence of characters organized into lines (and possibly pages)

**Source file**: a sequence of functions, each is further organized as declarations followed by executable statements

**Executable file**: a series of code sections that the loader can bring into memory and execute

Attributes:

**Name**: a symbolic file name is the only information kept in human readable form. Three types: inode (not readable), absolute path, runtime state (descriptor)

**Identifier**: this unique tag usually a number. identifies the file within the file system; not human readable.

**Type**: this information is needed for the systems that support different types of tiles

**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 and possibly the maximum allowed size are included

**Protection**: access control information about who can read/write/execute

**Timestamps and user info**: useful for protection, security and usage monitoring

**Create**: Allocate disk space. Create a file descriptor for the file including name, location on disk, and all file attributes. Dd the file descriptor to the directory that contains the file. An optional file attribute is file type, like word file or executable.

**Delete**: file the directory containing the file, free the disk blocks used by the file. Remove the file descriptor from the directory. Remove behavior dependent on hard links

**Open**: check if the file already open by another process, if not then find the file and copy the descriptor into a system wide open file table. Check the protection of the file against the requested mode and abort if not. Increment open count. Create an entry in the process file table pointing to the entry in the system wide file table. Initialize file pointer to start of file.

**Close**: remove entry for the file in the process’s file table. Decrement the open file count and if open count is 0, then remove the entry in the system wide file table.

**Read**(fileID, from, size, buffAddress): OS reads *size* bytes from file position *from* into *buffAddress*

**Read**(fileID, size, buffAddress) – sequential access: OS reads *size* bytes from current file position, fp, into *bufAddress* and increments current file position by size.

**Write**: similar to reads but copies from the buffer to file

**Seek**: just updates fp

**Memory mapping a file**: map a part of the portion virtual address space to a file. Read/Write to that portion of memory. File access are greatly simplified as no read write calls are necessary.

From programmer perspective:

**Sequential**: Data process in order, 1 byte or record at a time like compiler reading source file.

**Direct**: Address a block based on a key value like a database search.

From OS perspective:

**Sequential**: keep a point to the next byte in the file. Update the pointer on each read/write

**Direct**: address any block in the file directly given its offset within the file.

**Operations and access path**

*Search for a file*: locate an entry for a file

*Create a file*: add a directory listing (mkdir)

*Delete a file*: remove directory listing (rm -rf)

*List a directory*: list all files (ls command in unix)

*Rename a file*

**Traverse the file system**

**Single Leveled**: All files are contained in a single directory. Very east to support and maintain, but all files must have a unique name and it gets very difficult to keep files organized.

**Two Level**: A master level directory points to user file directory. Thus different users can use the same file name and there is less confusion. In order to search for a file you must specify the location of the file using a search path

**Tree structured**: Like two level but as many directories as you want. Absolute path names which begin at the root lead to the file. While relative path names start at current directory to the file.

**Acyclic-Graph Directories**: Like tree structured but now files can be shared between directories as well.

**Linear List**: Simplest method of implementing a directory. Just have a list of file names with pointers to data blocks. The problem is that finding a file requires a linear search. This is very time consuming as it will have to be done very frequently.

**Hash Table**: here a data linear list stores the directory entries, but a hash table is implemented. This decreases search time but its downside is its fixed size. This can be fixed by using a chained overflow hash table. Where each hash entry is a linked list. Collisions are resolved by adding a new value to the linked list. This sacrifices search time but is still much faster than a linear list.

**Contiguous Allocation**: OS maintains an ordered list of free disk blocks. OS allocates a contiguous chunk of free blocks when it creates a file. Need to store only the start location and the size in the file descriptor.

*Pro*: Simple.

*Con*: Difficult to increase file sizes. Fragmentation. Compaction

**Linked Allocation**: Keep a list of all the free sectors/blocks, In the file descriptor, keep a pointer to the first sector/block. In each sector keep a pointer to the next sector.

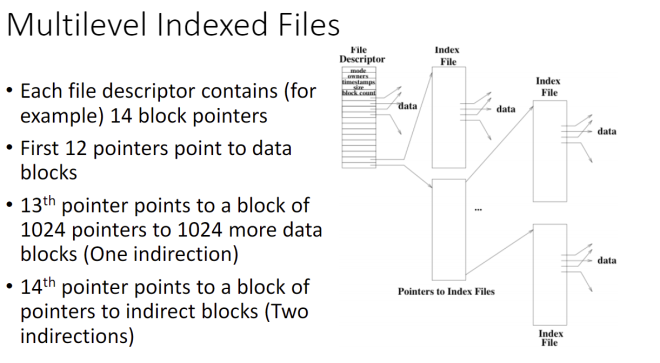
*Pro*: no fragmentation. Easy to increase file size. Good for sequential access

*Cons*: no direct access. High number of seeks.

**Index Allocation**: OS keeps an array of block points for each file. The user or OS must declare the maximum length of the file when it is created. OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only on demand. OS fills in the pointers as it allocates blocks.

*Pros*: not much wasted space for files. Both sequential and random accesses are easy.

*Cons*: Wasted space in file descriptors. Sets a maximum file size. Lots of seeks as data isn’t continuous.



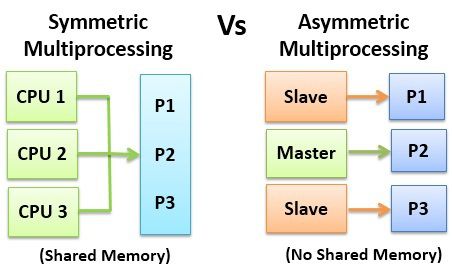
**Number of ptrs/block** = block size/disk address

**Total** = (#direct + #indirect(#ptrs)1 + #double-ind(#ptrs)2 + …) \* block size

Operating System:

**Definition**: Resource allocator, manager of resources. Make the computer user friendly and take care of regular maintenance.

**Kernel**: Always running. Central module of an OS

****

**Asymmetric Multiprocessing**: Each processor is assigned a specific task.

**Symmetric Multiprocessing**: Each processor performs all tasks

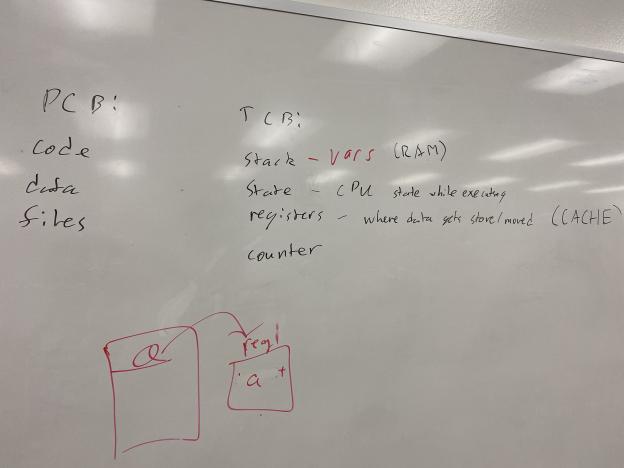
**Dual-mode**:

*Kernel Mode*: Executing code has complete and unrestricted access to the underlying hardware. Execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the OS.

*User Mode*: Executing code has no ability to directly access hardware or reference memory. Must delegate to system APIs to access hardware or memory. Crashes always recoverable. Mode most code executes in.

Processes:

**PCB**:



OS data structures to keep track of all processes

• The PCB tracks the execution state and location of each process

• The OS allocates a new PCB on the creation of each process and places it on a state queue

• The OS deallocates the PCB when the process terminates

*Contains*:

• Process state – running, waiting, etc.

• Program counter – location of instruction to next execute

• CPU registers – contents of all process-centric registers

• CPU scheduling information – priorities, scheduling queue pointers, state

• Memory-management information – memory allocated to the process. heap

• Accounting information – CPU used, clock time elapsed since start, time limits

• I/O status information – I/O devices allocated to process, list of open files

Short term scheduler runs infrequently (ms), long term runs frequently (min)

**TCB**:

Register Values, Stack pointer, Program counter, Scheduling State