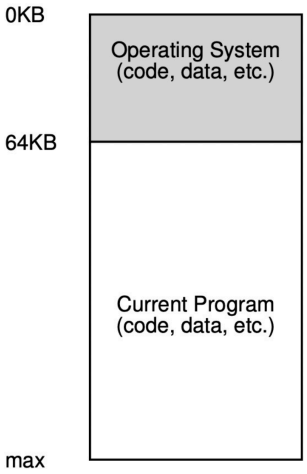
Operating Systems Final Review Notes

(Written by Ruixiang Jiang)

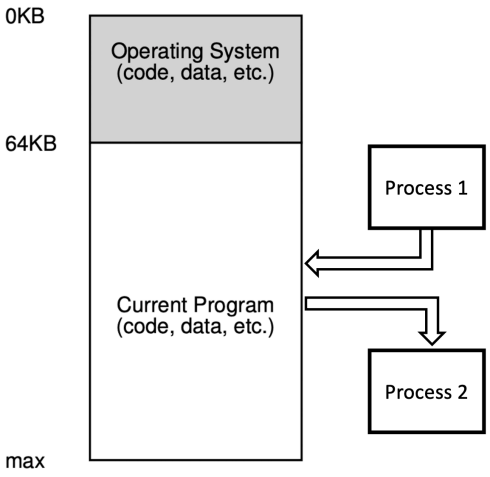
Address Translation

Ways to Use Physical Memory

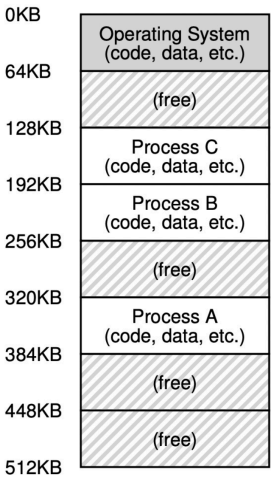
- In the early days, one running program used the rest of the memory



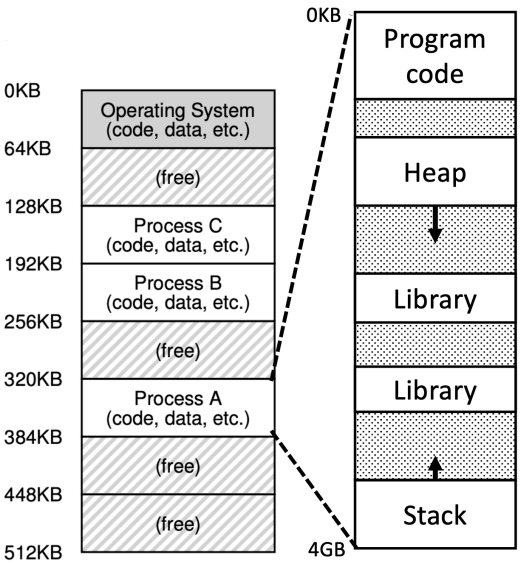
- In 1966, multiple processes were ready to run at a given time, and the OS switched between them



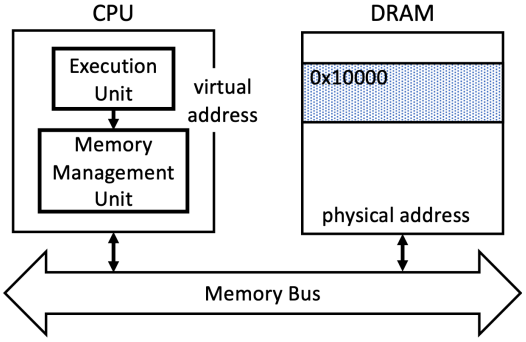
  - benefits:
    - time-sharing of computer resources
    - more effective use of the CPU
  - weakness: moving data in and out of the memory is slow
- Multiprogramming with memory partition



Memory Virtualization

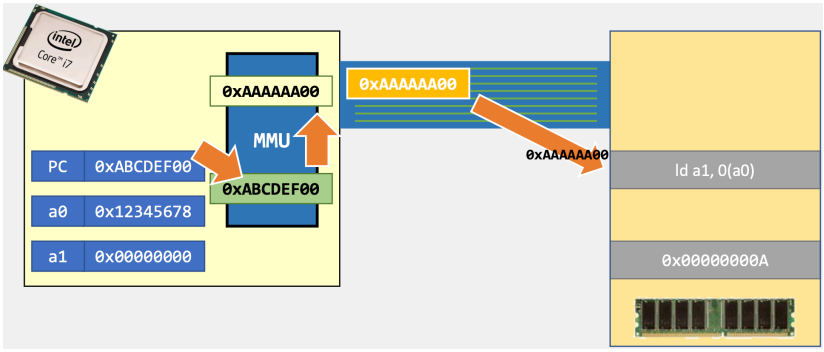


- Virtual address: address in a process's own address space
- Physical address: address of the physical memory
- Virtualized memory should:
  - be transparent: it is invisible to process, which runs as if on a single private memory
  - be efficient: fast and not too space-consuming
  - provide protection: enable memory isolation so that one process can not access others' memory
- CPU (virtual address) and DRAM (physical address) are connected by the memory bus

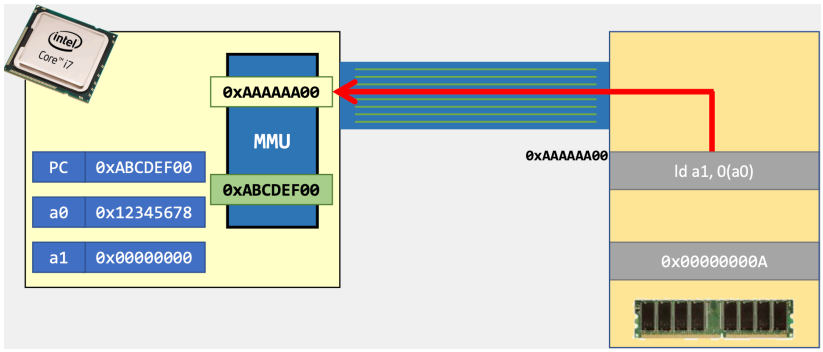


Address Translation

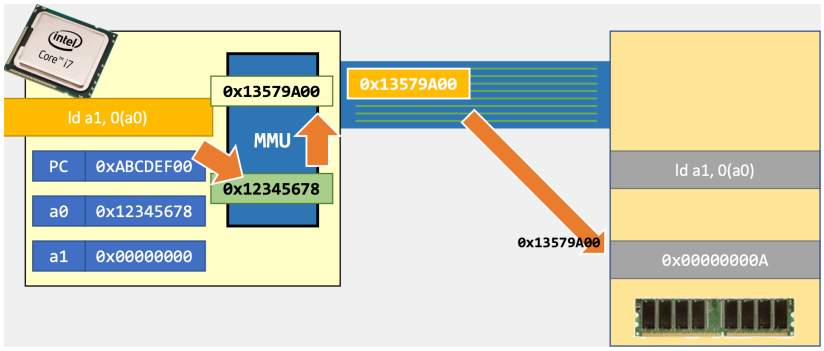
- It needs coordination between MMU and OS software
  - MMU (Memory Management Unit)
    - usually is CPU hardware, but may also be off-chip or pure software
    - translates VirtAddr used by instructions to PhysAddr understood by DRAM
    - CPU interposes every memory access
  - OS
    - sets up hardware for correct translation
    - keeps track of which locations are accessible and which are in use
    - maintains control of how memory is used
- Procedure
  - CPU wants to fetch the instruction at the PC register, which saves the VirtAddr



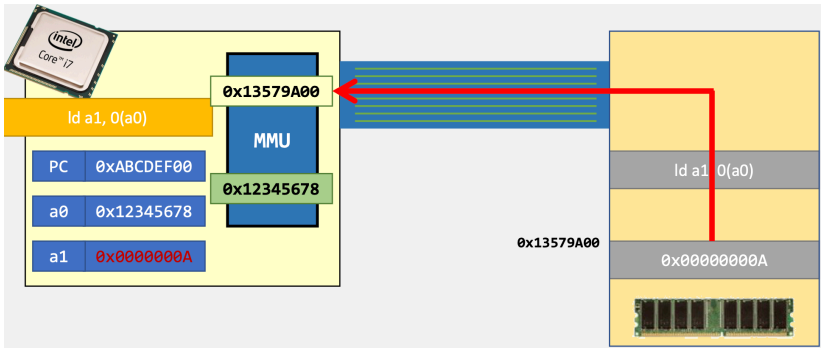
  - CPU gets the instruction fetched from the corresponding physical address



  - CPU executes the instruction and accesses some virtual addresses if need

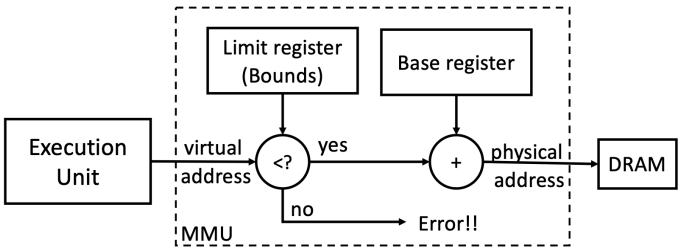


  - CPU stores the data retrieved from memory in the EAX register



Base & Bounds: Dynamic Relocation

- Two hardware registers
  - base register: holds the starting address of a block of memory, indicating the location in physical memory where a process's memory segment begins
  - bounds/limit register: holds the size of the memory segment allocated to a process



- Hardware & OS coordination
  - hardware
    - base and bound registers
    - privileged mode to update base and bounds registers

      processes in user mode can not update the two registers
    - privileged instructions to register exception handlers

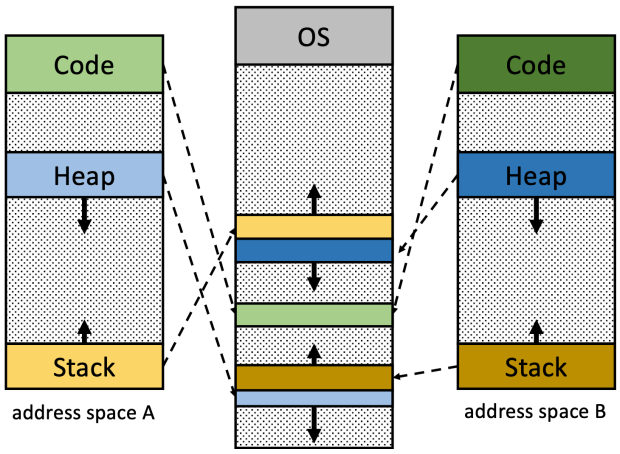
      only the OS but not the processes that can tell the hardware how to handle exceptions
  - OS
    - memory management

      allocates memory for new processes, reclaims memory from terminated processes, and manages memory via the free list
    - base and bounds registers management

      sets the registers properly upon the context switch
    - exception handling
- Limitations
  - waste memory due to internal fragmentations
  - a process can't use a larger address space
  - hard to do inter-process sharing

Segmentation

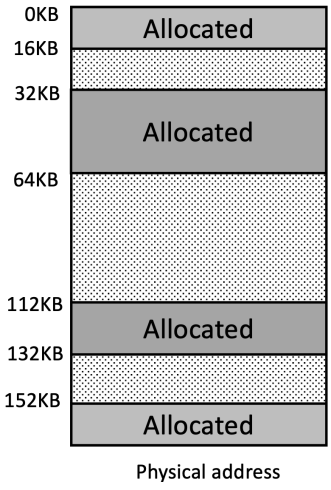
- It uses a pair of base & bounds registers for each segment, which is mapped to a different region of the physical memory



- It solves the following problems:
  - internal fragmentation

    still exists as space between heap and stack may be wasted
  - larger address space
  - inter-process sharing
- It has the following limitations:
  - OS context switch must also save and restore all pairs of segment registers
  - a segment may grow, which may or may not be possible
  - management of free spaces of physical memory with variable-sized segments
  - external fragmentation

    space between segments is hard to use
- OS manages all free physical memory regions



  - best fit
    - search for the smallest free memory chunk that satisfies
    - minimal external fragmentation
    - slow due to an exhaustive search
  - worst fit

        - search for the biggest free memory chunk that satisfies
        - leaves larger holes in physical memory
        - slow due to an exhaustive search, severe fragmentation in practice
    - first fit
        - search for the first free memory chunk that satisfies
        - fast
        - pollutes the beginning of the free list with small chunks

Paging

Put Solutions for Segmentation Together

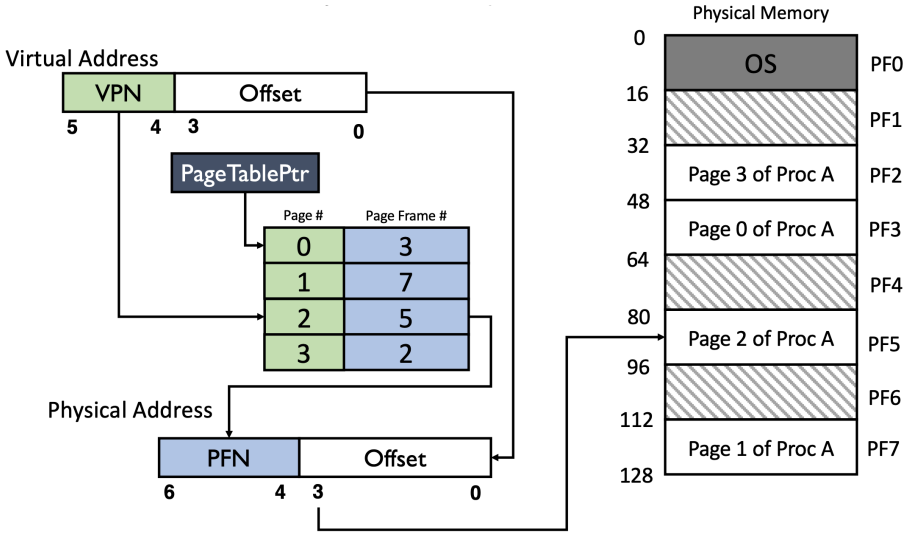
- Divides physical memory into fixed size conceptually, each is called a page frame

  It is not too big (internal fragmentation) and not too small (causes too big page table)
- Divides the virtual address into the same size conceptually, each is called a page

  Each request is of the same fixed size; thus no external fragmentation
- Page is mapped to page frame by one-to-one mapping or many-to-one mapping

  Many-to-one mapping enables memory sharing
- One page table per process resides in the physical memory

Paging Illustrated



- Address translation
    - address space size = $2^{length\ of\ virtual\ address}$
    - page size = $2^{length\ of\ offset}$
    - number of PTEs = $2^{length\ of\ VPN}$
    - page table size = number of PTEs * PTE size
    - page table size = page size if every page table is fitted into a single page
    - virtual address = VPN || Offset

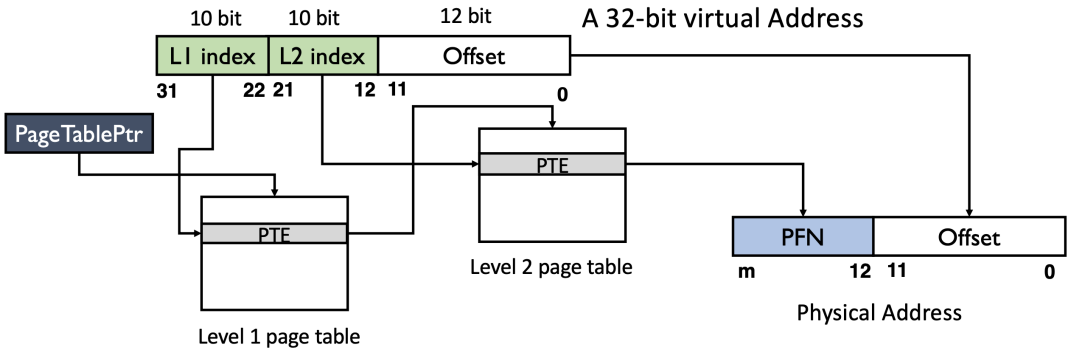
    - physical address = PFN || Offset
- Page table entry = page frame number + status bits
    - valid bit: 0 for pages with no valid mapping
    - protection bit: permission to read / write / execute
    - present bit: whether this page is in physical memory or on the disk
    - dirty bit: whether this page has been modified since it was brought into memory
    - access bit: whether a page has been accessed
- Page tables are stored in memory

  context switch only changes the pointer to page table

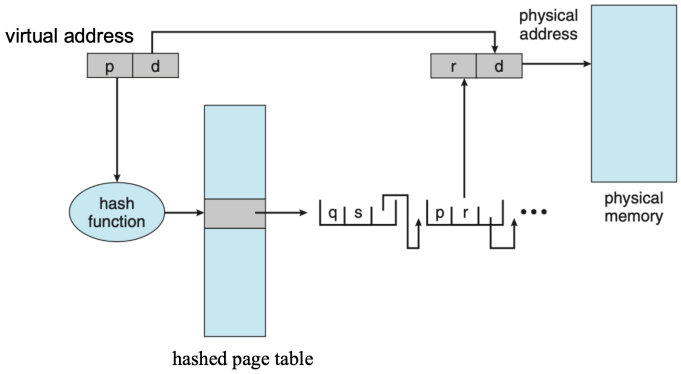
  both accessing the page table to retrieve the PTE and accessing the physical memory frame using the PFN from the PTE are in need of accessing memory

Multi Page Tables Structures

- Two-level page table

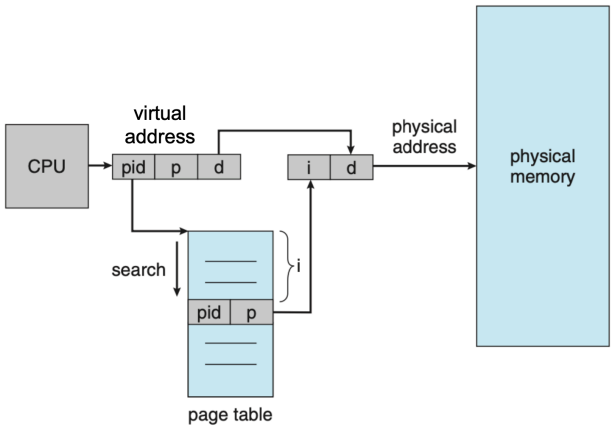


    - some entries in level-1 page table are invalid, which means they don't have level-2 page tables thus saves memory
    - need more steps to find PFN thus slow
- Hashed page table



  p: VPN; r: PFN; nxt: point to the next element

- Inverted page table



- only one page table for the whole system
- each entry is for a process
- need to search the pid linearly thus slow
- hard for sharing memory

Real-world Paging Schemes

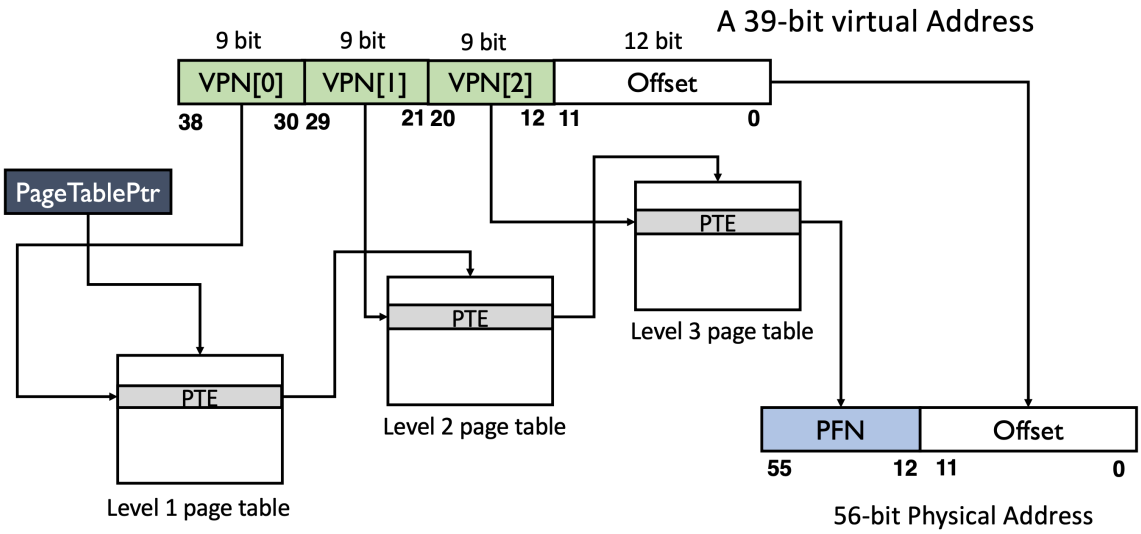
- SV39: three levels of page tables

  virtual address:

63 - 39	38 - 30	29 - 21	20 - 12	11 - 0
Reserved	VPN[0]	VPN[1]	VPN[2]	Offset

  PTE:

63 - 54	53 - 10	9 - 8	7 - 0
Reserved	PPN	RSW	Status bits

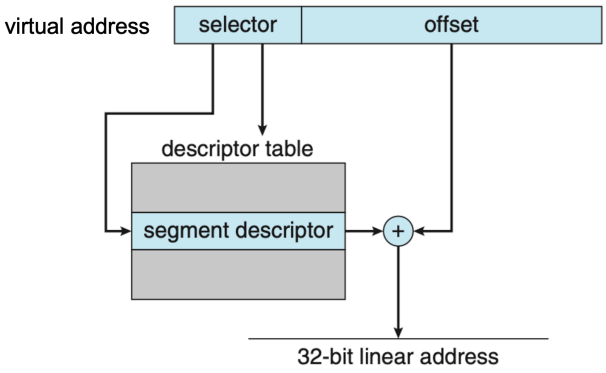


- IA-32

  virtual address -> Segmentation Unit -> linear address -> Paging Unit -> physical address

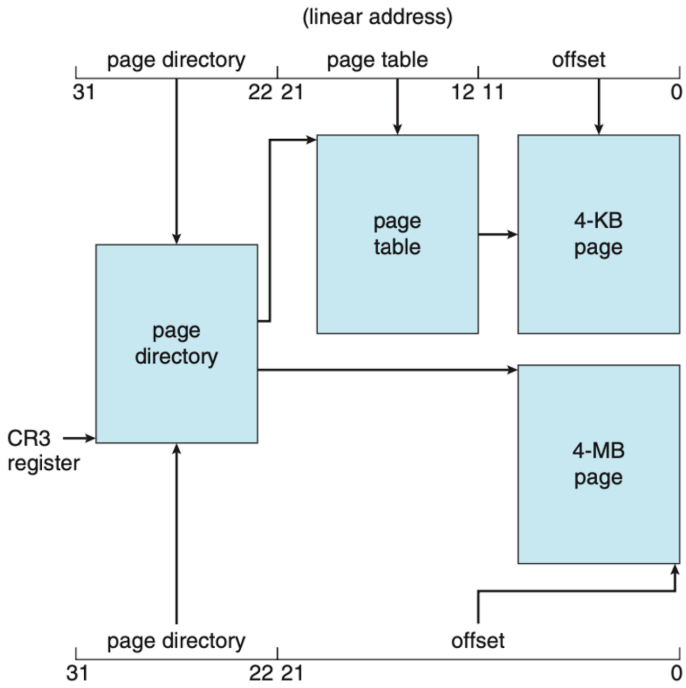
virtual address:

47 - 32	31 - 0
Selector	Offset



linear address:

31 - 22	21 - 12	11 - 0
Page directory	page table	offset



Translation Lookaside Buffer

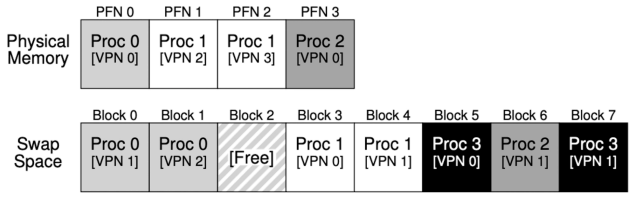
- TLB is a cache for the PTEs
- If TLB hit, just find the PFN in the same entry

  If TLB miss, go to the page table
- Two processes may use the same virtual address
  - flush TLB upon context switch: invalidate all entries by setting valid bit from V to I
  - extend TLB with address space ID: no need to flush TLB

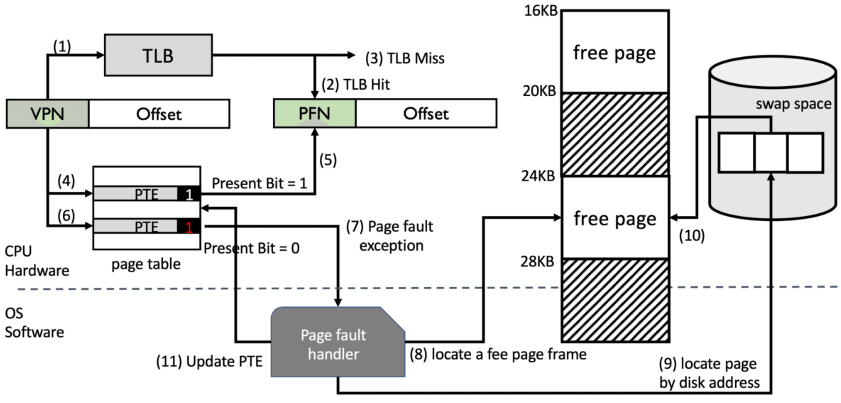
Demand Paging

Demand Paging Mechanisms

- Swap space
  - swap space is a partition or a file stored on the disk, conceptually divided in to page-sized units
  - swap space is used to store pages when the physical memory is used up
  - OS swaps pages between the memory and the swap space on demand
  - some datas moved out of the physical memory will be put into the file system directly
  - in the example below, process 0 has 3 pages; one is in memory and the other two are in the swap space



  - the physical memory can be regarded as a cache, whose block size is the size of a page
- Present bit
  - present = 1: the corresponding page is in the physical memory; access data from the physical memory
  - present = 0: the corresponding page is not in the physical memory; access page from disk
- Page fault
  - present = 0 raises a page fault exception
  - after such an exception, the page fault handler:
    - first, finds a free page frame in the physical memory, or triggers a page replacement
    - second, fetches the page from the disk and store it in the physical memory
  - after handling the page fault, CPU reexecutes the instruction that accesses the virtual memory



Page Replacement Policy

- Effective access time
  - effective access time = hit rate * hit time + (1 - hit rate) * miss penalty time

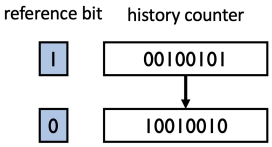
    - slowdown factor = effective access time / hit time
- Cache miss types
    - compulsory / cold-start miss

      pages are not fetched into memory at first, so it causes cold-start miss when a page is accessed for the fist time
    - capacity miss

      a page x is swapped out because the cache is filled up, then accessing the page x causes a capacity miss
    - conflict miss

      a page x is swapped out because it has the same index as another page y as the cache is not full, then accessing the page x causes a conflict miss
- Page replacement policies
    - optimal / MIN: replace the page that will not be used for the longest time
    - FIFO: replace the oldest page
    - random: replace a random page
    - LRU: replace the page that hasn't been used for the longest time (temporal locality)
    - LFU: replace the least frequently used page (spatial locality)
- Belady's anomaly: more page frames may lead to more page faults for FIFO
- LRU approximation
    - clock / second-chance algorithm
        - reference bit
            - one reference bit per page frame which indicates if the page has been accessed
            - all bits are cleared to 0 initially
        - all physical pages are in a circular list
        - OS maintains a pointer points to the current page, and check its reference bit when a replacement occurs:
            - if reference bit = 1, set the bit to 0 and move to the next page, which means a page has two chances
            - if reference bit = 0, choose the page to be replaced
    - enhanced clock algorithm
        - dirty bit
            - one dirty bit per page frame which indicates if the page has been modified recently
            - all bits are cleared to 0 initially
            - CPU sets dirty bit to 1 upon write access
        - the main body is the same as the clock algorithm
        - when a replacement occurs, OS checks (ref bit, dirty bit) and selects a candidate page:
            - (0, 0): neither used nor modified recently, the 1st candidate

        - (0, 1): not used recently but modified, the 2nd candidate
        - (1, 0): used recently but not modified, the 3rd candidate
        - (1, 1): used and modified recently, the 4th candidate
    - additional reference bits algorithm
        - use a 8-bit history register to compare who is the LRU page
        - trigger a timer interrupt every 100ms, and make ref , his = 0 , ref | his >> 1



        - compare the history counter as a unsigned integer, the larger value, the more recently used
    - nth-chance algorithm: expension of second-chance algorithm, usually choose n = 1000

Page Frame Allocation

- Page replacement
    - global replacement: each process selects frame from all page frames
    - local replacement: each process selects frame from its own set of allocated frames
- Allocation algorithms
    - equal allocation: each process gets the same amount of memory
    - proportional allocation: number of page frames is proportional to the size of the process

      s_i = size of process i, m = total number of frames

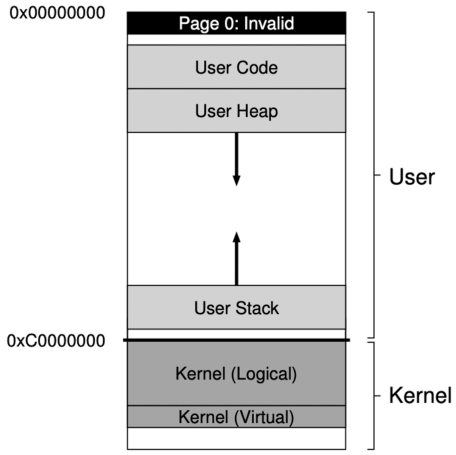
      allocated page frames number for process i = $m * \frac{s_i}{\sum s_k}$
    - priority allocation: number of page frames is proportional to the priority of the process
- Thrashing
    - thrashing happens when the system spends most of its time swapping pages in and out of memory due to the lack of memory, rather than executing user programs
    - early OS
        - working set model: keep the most frequently accessed pages of a process in physical memory
        - reduce the number of processes
    - modern OS
        - out-of-memory killer
        - reboot

Linux Memory Management

Address Space in Linux

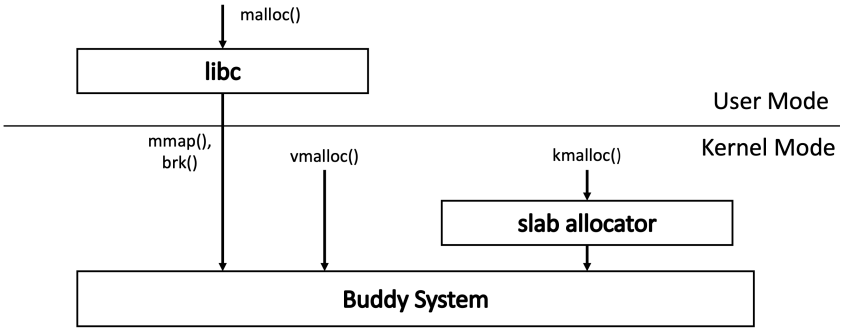
- The virtual address space of each process is split between user and kernel portions

- logical kernel
  - page tables
  - per-process kernel stacks
  - kmalloc()
- virtual kernel
  - virtually continuous memory
  - vmalloc()

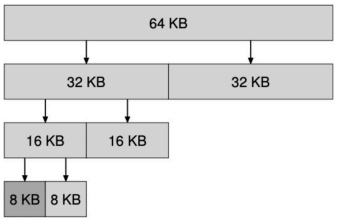


- The kernel memory is mapped into each process so that there is no need to change the page table when trapped into the kernel, no TLB flush
- The kernel code may access the user memory if need
- The kernel memory in each address space is the same

Linux Physical Memory Management



- Buddy system
  - free physical memory is a considered big space of size 2^N pages
  - when allocation, the free space is divided by two until a block that is big enough to accommodate the request is found



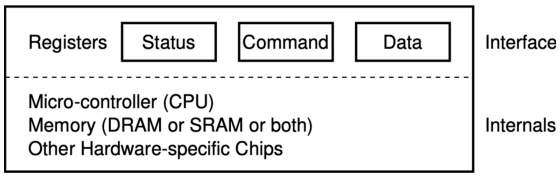
  - when free a block, the freed block is recursively merged with its buddy

    - may causes internel fragments
    - the unit for buddy system is a page, usually is 4KB, thus hard to control small pieces
- Slab allocator
    - a slab consists of one or more physically contiguous pages
    - a slab cache consists of one or more slabs
    - a slab can be in:
        - empty: all objects are free
        - partial: some objects are free
        - full: all objects are used
    - a request is first served by a partial slab, then an empty slab, then allocate a new slab from buddy system
    - no internel fragmentation
        - objects are packed tightly
        - the slab allocator returns the exact amount of memory required to represent the object
    - memory requests can be satisfied quickly
        - objects are created and initiated in advance
        - freed object is marked as free and immediately available for subsequent requests

I / O

Device

- Interface
    - status register
    - command register
    - data register
- Internal structure



Polling

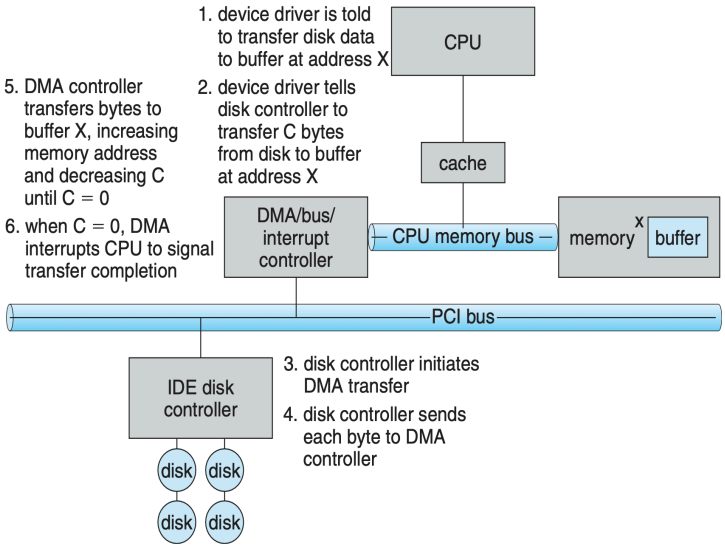
- Frequently check the status of IO devices
- To write a byte of data to device
    - OS repeatedly reads the status register, waiting for the device to be ready
    - OS sends some data to the data register
    - OS writes a command to the command register

    - OS waits for the device until the device is done with the request
- Polling is inefficient and inconvenient
- If CPU switches to other tasks, the data may be overwritten
- Polling works better for faster devices

Interrupt

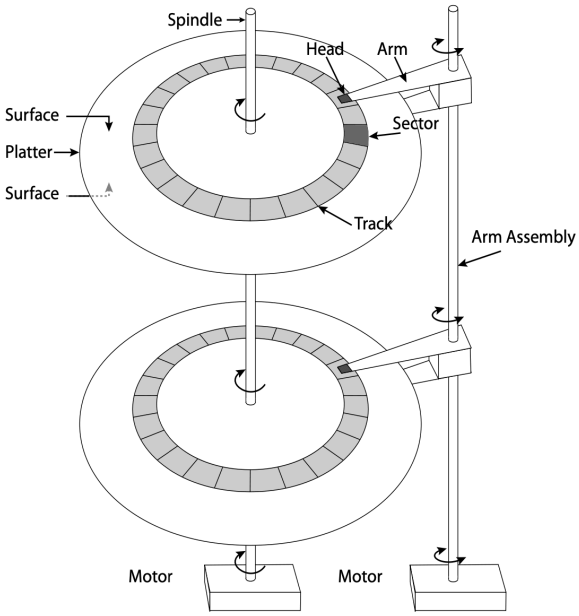
- The device raises a hardware interrupt, causing the CPU to jump into the OS at a predetermined interrupt handler
- Interrupt works better for slow devices
- Hardware support
    - interrupt-request line
    - interrupt-controller hardware
- Software support
    - interrupt handler
    - a table of interrupt vectors to specify interrupt-handling routine

DMA



Storage

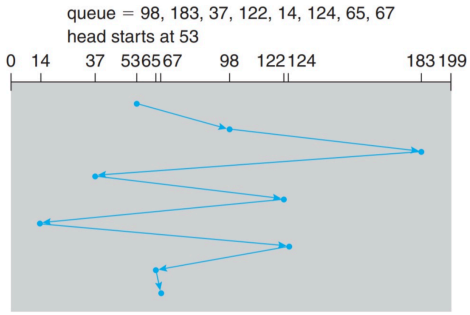
Magnetic Disk



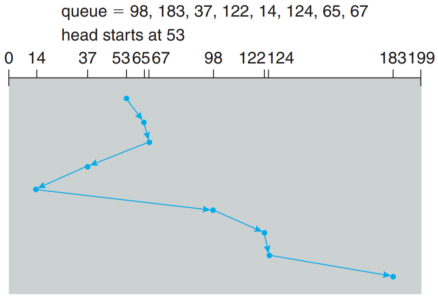
- Total time = seek time + rotation time + transfer time
- The average rotation time is $\frac{1}{RPM} \div 2$, which means the time to rotate half the circle
- The transfer time can be ignored because it is usually less then 1ms

Disk Scheduling

- FIFO
  - first in, first out
  - fair but may cause very long seeks

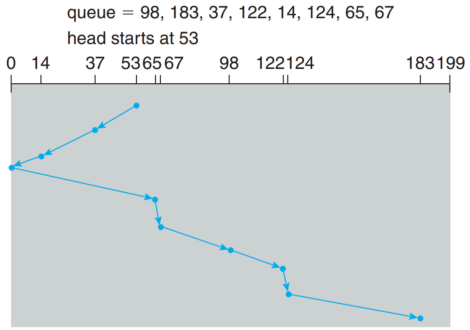


- SSTF
  - shortest seek time first
  - may cause starvation
  - common and has a natural appeal

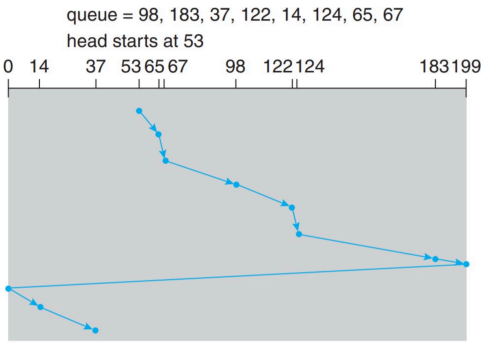


- SCAN

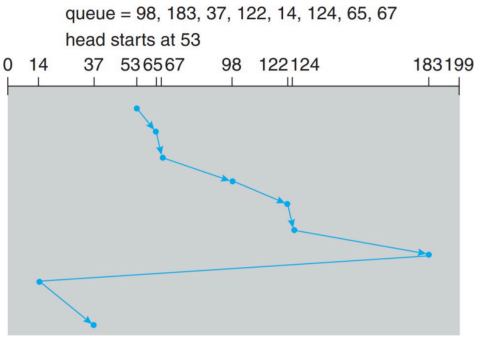
    - like an elevator
    - needs to touch the ends of the disk
    - data at the end of the disk wait the longest
    - performs better at a heavily loaded disk



- C-SCAN
    - the head moves from one end of the disk to the other and serves requests; when it reaches the other end, it returns to the beginning of the disk without serving
    - more uniform wait time



- Look / C-Look
    - a variant of SCAN / C-SCAN
    - the arm only goes as far as the last request in each direction, then reverses direction immediately without first going all the way to the end of the disk



File System

Contiguous Allocation

- Each file or process is allocated a single contiguous block of memory
- It causes external fragmentation, and it uses a defragmentation process to solve it as the process is expensive

- Application
  - ISO 9660
  - CD-ROM (e.g. iso image)

Linked Allocation

- Each file or process is divided into blocks, and each block contains a pointer to the next block, forming a linked list
- No external fragmentation, but may have internal fragmentation
- Dynamic growth of files is supported
- Accessing a specific block can be slower because the system may need to traverse the pointers sequentially
- File Allocation Table (FAT)
  - structure

    the root directory is a table:

Filename	Attributes	First block number	Size

    both the FATs are the same:

Block	Next block

    the directory entry describes a file or a sub-directory under a particular directory:

Filename	Attributes	First block number	File size

    the filename must be 7 characters filename + 3 characters extension; the block number is 4 bytes (32 bits); the file size is 4 bytes
  - steps of reading a file
    - reads the root directory and retrieves the first block number
    - reads the FAT and moves on to the next block
    - stops until the next block number is -1
  - steps of appending a file
    - locates the last block
    - starts writing to the non-full block
    - allocates the next block through FSINFO
    - updates the FATs, FSINFO, and the file size
  - steps of deleting a file

        - deallocates all the blocks involved and updates the FATs and FSINFO
        - changes the first byte of the directory entry to _ (0xe5)
    - on DOS, a block is called as a cluster

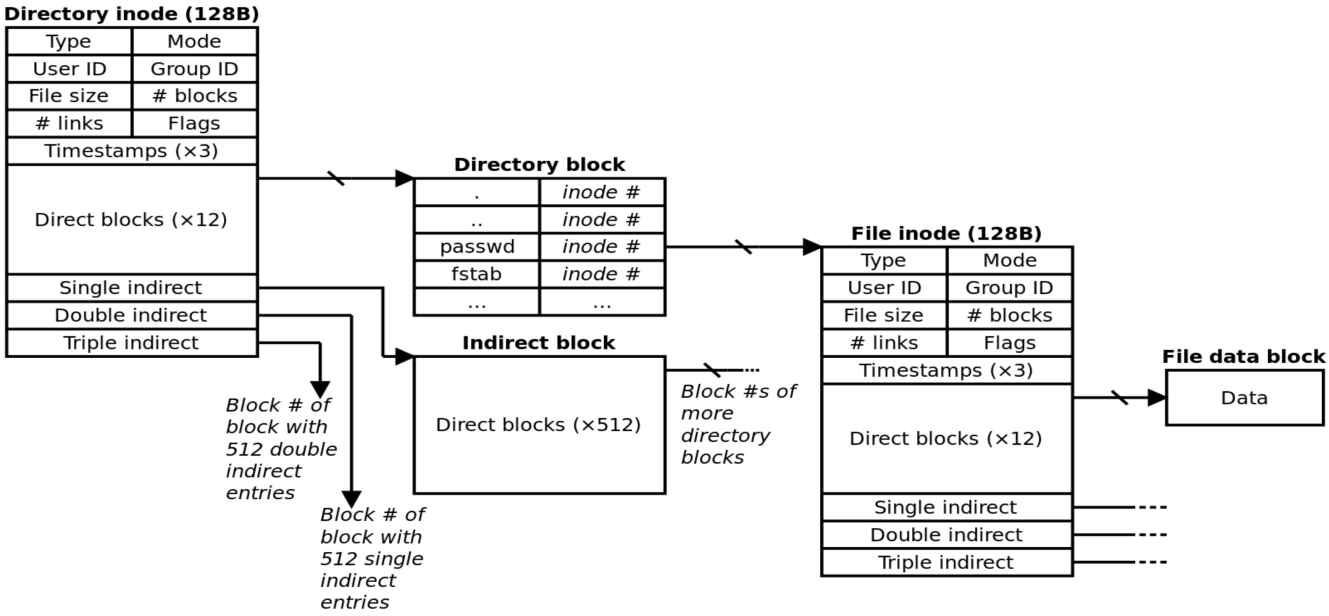
	FAT12	FAT16	FAT32
Cluster address length	12 bits	16 bits	28 bits (4 reserved bits)
Number of clusters	2^{12}	2^{16}	2^{28}

      file system size = cluster size * number of clusters

      the largest FAT32 file size is $2^{32} - 1$ bytes
    - use LFN added to the normal entry to support longer filenames
    - application
        - CF card
        - SD card
        - USB driver

iNode Allocation

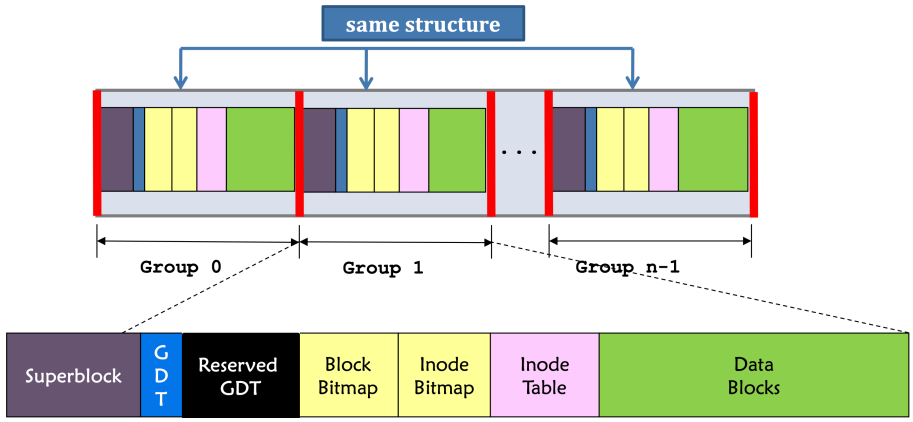
- All pointers of a file are located together
- Each directory or file has one inode



- For block size = 2^x bytes and address length = 4 bytes (32 bits),
    - number of blocks = $12 + (2^x/4)^1 + (2^x/4)^2 + (2^x/4)^3$
    - max file size = number of blocks * block size
    - number of block addresses = 2^{32}
    - file system size = number of block addresses * block size

Extended File System 2 / 3

- Disk layout
  - the file system is divided into block groups, and every block group has the same structure

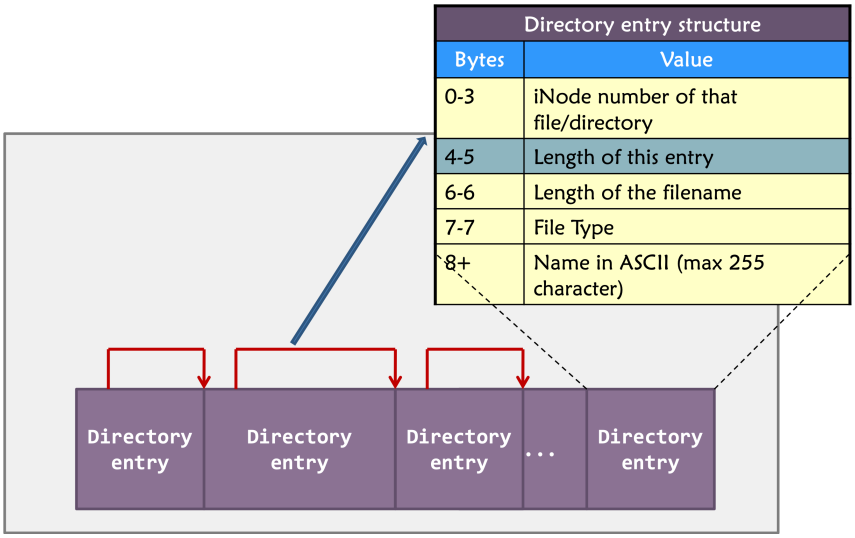


Superblock	Stores FS specific data. E.g., the total number of blocks, etc.
GDT – Group Descriptor Table	It stores: - The locations of the **block bitmap**, the **iNode bitmap**, and the **iNode table**. - Free block count, free iNode count, etc...
Block Bitmap	A bit string that represents if a block is allocated or not.
iNode Bitmap	A bit string that represents if an inode (index-node) is allocated or not.
iNode Table	An array of inodes ordered by the inode #.
Data Blocks	An array of blocks that stored files.

  - the number of files in the file system is fixed
  - groups for performance and reliability
    - performance: group inodes and data blocks of related files together (spatial locality)
    - reliability: superblock and GDT are replicated in each block group
- Directory
  - inode structure for one file

iNode Structure (128 bytes long)	
Bytes	Value
0-1	File type and permission
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count (will discuss later)
...	...
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
...	...
108-111	Upper 32 bits of file sizes in bytes

  - directory entry in a directory block



file deletion is just an update of the entry length of the previous entry

the file name is stored in the directory entry, while the file attributes are stored in the inode

- Link
  - hard link (like soft copy)
    - a hard link is essentially an additional directory entry for an existing file which points directly to the inode of the file
    - no additional data storage is required other than the directory entry itself, as the data blocks are shared
    - hard links to a file share the same inode number, meaning they refer to the same physical data on the disk
    - the file can be accessed through two different pathnames
    - deleting one hard link does not delete the actual file until all hard links are deleted
    - there is a link count value in the inode, which is increased by 1 if a new link to it occurs or decreased by 1 if a link is removed
    - the data blocks and the inode will be deallocated if the link count reaches 0
  - soft / symbolic link (like shortcut)
    - a soft link is a special type of file that contains a path to another file
    - when a soft link is created, it gets a new inode which points to a new data block that stores the pathname of the target file
    - soft links have their own inode and data block, which contains the pathname of the target file
    - additional data storage is required for the soft link itself, specifically for storing the path to the target file
    - if the target file is deleted, the soft link becomes a dangling (broken) link, which points to a non-existent file

Deadlock

Requirements

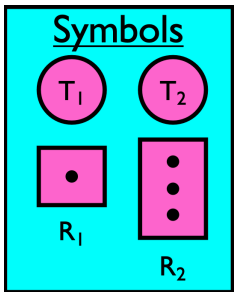
- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Solution

- Terminate the process
- Make the process preempt resources (not always work)
- Roll back actions of the process (commonly used, deadlock may happen again)

Resources-Allocation Graph

- Symbols
  - processes: $T_1, \ldots, T_n$
  - resources: $R_1, \ldots, R_m$
  - instances: each resource R_i has W_i instances
  - request edge: directed edge from T to R
  - assignment edge: directed edge from R to T

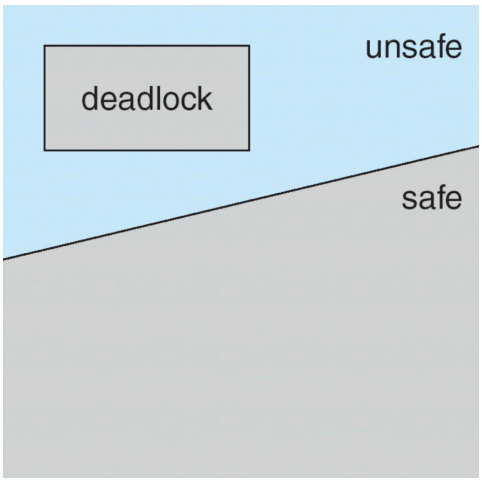


- Deadlock detection
  - each resource has only one instance: look for cycles
  - otherwise: detection algorithm
    - predefined arrays
      - available[m] indicates the number of available resources of each type
      - work[m] indicates which is a temp array of available[]
      - finish[n] indicates if a process is finished
      - allocation[n][m] indicates the number of resources of each type that each process has allocated till now
      - request[n][m] indicates how many instances the current request still needs now
    - initialization
      - work[] = available[]
      - finish[i] = (allocation[i] == 0)
    - find i such that finish[i] == false and request[i][] <= work[]

        - found: work[] += allocation[i][], finish[i] = true, go back to find another i
        - not found: if finish[j] == false for some j, process j is deadlocked

Safe State

- A system is in safe state if there exists a process sequence such that each process can be executed one by one



- Banker's algorithm to detect the state
    - predefined arrays
        - available[m] indicates the number of available resources of each type
        - work[m] indicates which is a temp array of available[]
        - finish[n] indicates if a process is finished
        - max[n][m] indicates how many instances the current request needs at most globally
        - allocation[n][m] indicates the number of resources of each type that each process has allocated till now
        - need[n][m] indicates how many instances the current request still needs at most (need[][] = max[][] - allocation[][])
    - initialization
        - work[] = available[]
        - finish[] = false
    - find i such that finish[i] == false and need[i][] <= work[]
        - found: work[] += allocation[i][], finish[i] = true, go back to find another i
        - not found: if finish[] == true, the system is in a safe state; otherwise unsafe
    - the Banker's algorithm is used to detect if the state is still safe after allocating some resources and prevent deadlock in advance, as the deadlock detection algorithm is used to detect if a deadlock exists
- Resource-request algorithm for each process i:
    - if request[i][] > need[i][], raise an error condition because the process has exceeded its maximum claim
    - if request[i][] > available[][], the process i must wait since resources are not available

- now the process i can try to allocate requested resources, update available[][], allocation[i][], need[i][], and detect the state
  - safe: the resources are allocated to process i
  - unsafe: the process i should wait, and the old arrays are restored

Welcome Reward