CPU-I/O Cycle: Process execution consists of a cycle of:

1. CPU execution (**CPU burst time)**
2. I/O wait (**I/O burst time)**

CPU Scheduler: Selects a process from the ready queue

- **Non-pre-emptive:** Process runs until it voluntarily relinquishes CPU
- **Preemptive:** Process runs for a maximum of some fixed time

Average Waiting time calculation: Calculate wait time for each process and then average it

Scheduling Algorithms:

- **FCFS:** First come first serve
    - **Problem:** ???
- **LIFO:** Last in first out
    - **Problem:** May lead to starvation – early processes may never get CPU
- **SJF:** Shortest job first
    - Estimated CPU burst time is associated with each process. Scheduler uses these lengths to schedule the process with the shortest time
    - Optimal – gives minimum average waiting time for a given set of processes
- **Priority Scheduling:** A priority number is associated with each process
    - **Problem:** Starvation
    - **Solution:** Aging
        - As time progresses, increase the priority of the process
- **RR:** Round Robin
    - Each process gets a small unit of CPU time (**time quantum)**. After this time has elapsed, the process is pre-empted and added to the end of the ready queue
    - If there are n processes in the ready queue, and the time quantum is q, then each process gets at most q time units at once. No process waits for more than (n-1)q time units
    - If q is too large: FIFO-like behaviour
    - If q is too small: context switching

Multilevel Queue Scheduling: Uses multiple queues. Essentially the ready queue is really multiple separate queues. The processes can be classified into different groups (i.e. foreground vs background)

- Each queue has its own scheduling algorithm
- Scheduling must be done between the queues:
    - **Fixed priority Scheduling:** Serve all from foreground then from background
    - **Time Slice:** Each queue gets a certain amount of CPU time which it can Schule amongst its processes
- A process can move between queues. Separate processes according to the characteristics of the CPU bursts (i.e. if a process has too much CPU time, it will be moved to a lower

priority queue, or a process that waits too long in a low priority queue may be moved to a higher priority one)

Multiple-Processor Scheduling

- **Asymmetric Multiprocessing:** There is one processor that makes the decisions for scheduling, I/O processing, and system activities
    - Other processor(s) execute only user code
    - Simple approach to master-slave model / centralized command model
- **Symmetric Multiprocessing:** Each processor is self-scheduling, and all processors share a **common ready queue** OR each processor may have its own **private queue** of ready processes

Race Condition: Behaviour of a system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the order the programmer intended

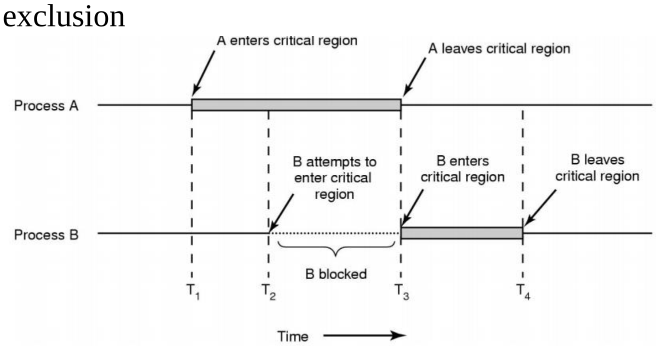
- Example: We have an application that withdraws money from the bank. Two requests for withdrawal from the same account comes to a bank from two different ATM machines
    - Assume a balance of $1000
    - A thread for each request is created
    -

Process / Thread 1	Process / Thread 2
1. Read balance: $1000	
2. Withdraw authorized for $600 (now actual balance is $400)	
CPU switches to process 2→	3. Read Balance $1000
	4. Withdraw authorized for $600 (this is unreal!!)
5. Update balance $1000-$600 = $400	← CPU switches to process 1
CPU switches to process 2→	6. Update balance $400-$600 = $-200

Critical Section: Any piece of code that accesses shared data (i.e. printer, bank account)

Mutual Exclusion: Ensures that only one thread/process accesses the critical section at a time

- Today the assumption is that the OS will provide some sort of support for mutual exclusion
-



Mutual Exclusion Solutions:

- **Peterson's Solution:** Restricted to 2 processors
    - 2 variables are shared, **int turn** and **int flag[2]**
    - The variable turn indicates whose turn it is to enter the critical section
    - The flag array is used to indicate if a process is ready to enter the critical section (flag[i] = true implies that process P_i is ready)
    - ```
      Process 1
      flag [0] = false; flag[1] = false;
      {
        flag[0] = true;
        turn = 2;
        while (flag[1] && turn == 2);
            critical section
            flag[0] = false;
            remainder section
       }
      ```
      ```
      Process 2
      flag [0] = false; flag[1] = false;
        {
        flag[1] = true;
        turn = 1;
        while (flag[0] && turn == 1);
            critical section
            flag[1] = false;
            remainder section
        }
      ```
- **Disabling Interrupts:** Guarantees that there will be no process switch
    - Process disables all interrupts before entering its critical section
    - Process enables all interrupts just before leaving the critical region
    - CPU is switched from one process to another only via clock/interrupts
    - ```
      bank_example (account, amount_to_withdraw)
      {disable(interrupts);
        1.   balance = get_balance(account);

        2.   if (balance => amount_to_withdraw)
                     withdraw_authorized();
             else
                     withdraw_request_denied()

        3.   balance = balance - amount;

      enable(interrupts);
      }
      ```
    - **Disadvantages:**
        - Gives the power to control interrupts to user
        - Does not work in the case of multiple CPUs. Only the CPU that executes the disable instruction is effected

- **Test and Lock Instruction (TSL):**
  - Many computers have a TSL REGISTER, LOCK instruction
    - Reads LOCK into register REGISTER
    - Stores a nonzero value at the memory location LOCK
    - The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing the memory until TSL instruction is done
  - **Disadvantages:** Requires **busy waiting,** which wastes CPU cycles that some other process might be able to use

```
enter_region
  TSL REGISTER,LOCK
  CMP REGISTER, #0
  JNE enter_region
    Critical Section
```

- If lock value != 0
  - JNE causes control to go to the start of enter_region
- If lock value is zero then enter CS

  -
- **Semaphores:** A process synchronization technique supported by the OS. Used to ensure mutual exclusion and send signals from one process to another
  - Allows multiple processes to cooperate by using signal. Semaphore is an integer variable with the following three operations:
    - **Initialize:** Semaphore (S) is initialized to a positive value
    - **Decrement:** (down operation) decrements the semaphore value
    - **Increment:** (up operation) increments the semaphore value
  - A process can enter a critical section only if S is positive
  - Two types of semaphores:
    - **Binary:** Allows only ONE process to be in critical section at a time
      - Initialized to 1
      - Also referred to as **mutex**
    - **Counting:** Allows multiple processed to be in critical section at a time
      - Initialized to N where N is the maximum processes that can be in critical section simultaneously

- Use down before entering a critical section
- Use up after finishing with a critical section
- Example: Assume S is initialized to 1.
-
```
    S = 1;
while (true)
{       down (S);
        critical section
        up(S);
        remainder section;
        }
```

  -

Deadlock: Two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes

- **Conditions:**
  - Resource allocation cycle
  - Mutual exclusion
  - Hold and wait
  - Non pre-emptive resource
- **Avoidance:**
  - Avoid cycles in the resource allocation graph
  - Avoid mutual exclusion
  - Avoid hold & wait
  - Block a process that is requesting a large amount of resources
- **Recovery:**
  - Abort all deadlock processes
  - Back up all deadlock processes to the previous safe state and then restart
  - Selectively abort processes until deadlock is broken

Memory Management: Memory management requires

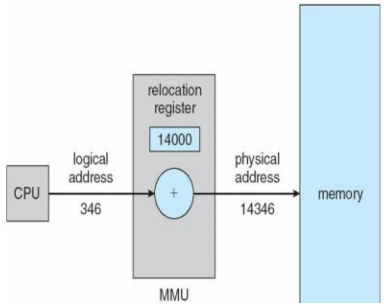
- Allocate memory to processes when needed
- Keep track of what parts of memory are in use
- Deallocate memory when processes are done

Memory Hierarchy

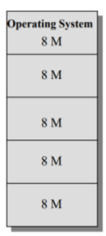
- Registers
- On-chip Cache (fast memory between the CPU and main memory so that CPUs don't have to wait for main memory)
- Main Memory
- Disk

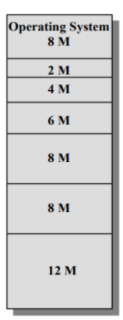
Address Binding: Memory references in the cod (**virtual** or **logical**) must be translated to actual physical memory addresses

Memory Management Unit (MMU): A hardware device that does run-time mapping from virtual to physical addresses

- The **base** register holds the physical memory address
- The **limit** register specifies the range
- These registers can be loaded only by the OS
- Ensures the user program doesn't access anything beyond its range
- **Relocation register:** Value is added to every address generated by a user process
- 

Memory Allocation Techniques

- **Contiguous Memory Allocation:** Each process is contained in a single section of memory that is contiguous
    - **Fixed Partitioning:** Any program. No matter how small, occupies an entire partition
        - Leads to **internal fragmentation**
        - No modern OS uses fixed partitioning
        - 

    - **Dynamic Partitioning:** Partitions are a variable length and number. Processes are allocated to the closest possible match
        - Leads to **external fragmentation**
        - **Compaction** is required to obtain a large contiguous block
            - Shift processes so they are contiguous and all free memory is in one block
        - 

Dynamic Partitioning Placement Algorithm

- Each process can be assigned to the smallest partition within which it will fit
- Processes are assigned in such a way as to minimize wasted memory within a partition
- Operating system must decide which free block to allocate to a process
- **Best-fit Algorithm:** Choose the block that is closest in size to the request
    - This has worst overall performance
    - The smallest block is found for a process
- **First-fit Algorithm:** Starts scanning memory from the beginning and chooses the first available block that is large enough

Paging: A memory management technique that avoids external fragmentation and compaction

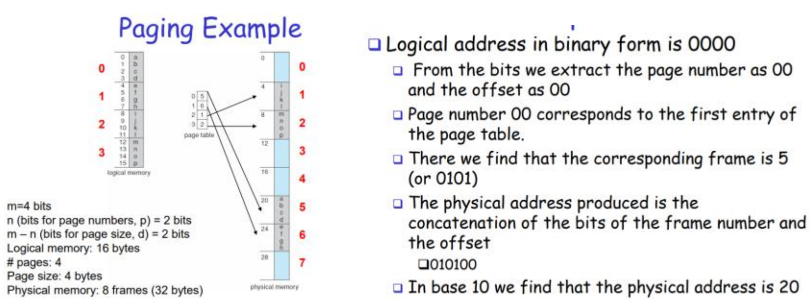
- Partition physical memory into small equal-size chunks and divide a process's logical memory into the same size chunks
- The chunks of a process are called **pages** and chunks of memory are called **frames**
- When a process is to be executes, it pages are loaded (from a file system) into any available memory frames
- A OS maintains a **page table** for each process that contains the frame location for each page of a process

Address Translation Scheme: Logical address generated by CPU is divided into:

- **Page number (p):** Used as an index to a page table which contains the frame number
- **Page offset (d):** Determines the location in the page which is needed to determine the location in the frame
- If the number of logical address bits is **m** and the number of bits to represent page numbers is **n**:
    - Logical address space = 2^m
    - Page numbers = 2^n
    - Page size is 2^{m-n}

Paging Hardware:

- The CPU issues a logical address
- The hardware extracts the page number **p** and the page offset **d**
- The page number **p** is used to index the page table (the entry of the page table consists of the frame number **f**)
- The actual address is the concatenation of the bits that make up **f** and **d**
-



Page Table Implementation: The simplest approach is to have the page table implemented as a set of dedicated registers. Not feasible to keep page tables in registers. Because page tables can be very large

- Each process has a page table
- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- During a context switch, changing page tables requires changing PTBR
- Solution: Use a special fast-lookup hardware cache called associative memory or **translation look-aside buffers (TLBs)**

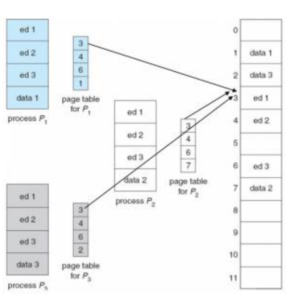
Translation Look-Aside Buffers (TLB): The TLB contains only a few of the page-table entries. When a logical to physical address is requested by the CPU its page number is presented to the TLB

- Address translation (p,d)
    - If **p** is in associative memory, get frame number out
    - Otherwise get frame number from page table in memory
- **TLB hit:** If found the frame number is immediately available
- **TLB miss:** If a page number is not in the TLB
    - The page table is consulted
    - The page number and frame number is added to the TLB
    - If the TLB is full then one of the entries is replaced (discussed later)
- **Hit Ratio:** Percentage of time that a particular page is found
- **Miss Ratio:** Percentage of time that a particular page is not found (100 – hit ratio)
- **Effective Access Time:** EAT = hit ratio * TLB hit time to get data + miss ratio * TLB miss time to get data
    - i.e. let Hit ratio = 80%
    - Therefore, Miss ratio = 20%
    - TLB hit time to get data = 120
    - TLB miss time to get data = 220
    - Effective access time = 0.80 * 120 + 0.20 * 220

Page Protection: Memory protection implemented by associated protection bit with each frame

- One protection bit can define a page to be read-write or read-only

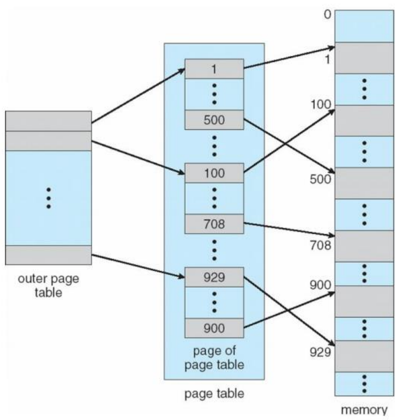
Shared Pages:



Page Table Structure: Typically systems have large logical address spaces

- Page table could be excessively large
- i.e. 32-bit logical address space, ∴ the number of possible address in the logical address space is 2^{32}
- Page table may consist of up to 1 million entries (very large)

Two-Level Page Table: Solution to large page tables. A page table is also paged. Need to be able to index the outer page table

- 

Virtual Memory: This is basically logical address space. Processes use a virtual (logical) address space

- Every process has its own address space
- Only part of the virtual address space is mapped to physical memory at any time

Demand Paging: Bring a page into memory only when it is needed.

- Why? Less I/O needed and less memory needed
- We need to distinguish between pages that are in memory and the pages that are on disk
- A valid-invalid bit is part of each page entry
    - When bit is ‘valid’, associated page is in memory
    - When bit is ‘invalid, the page is on the disk

Page Fault: Happens if a process tries to access a page that was not brought to memory

Page Replacement: Determine what page entry to replace when page table is full

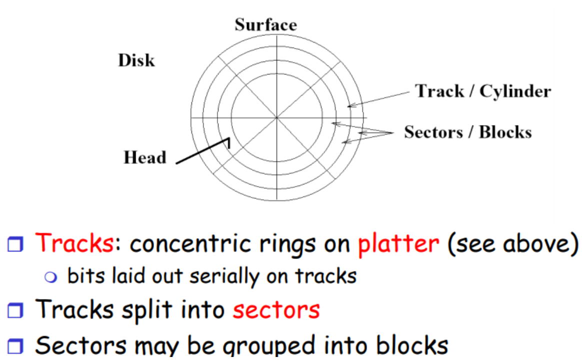
- Designing an appropriate algorithm is important since disk I/O is expensive
- **Optimal Page Replacement:** Replace page needed at the farthest point in future (i.e. replace the page that will not be used for the longest period of time)
    - Lowest page fault rate
    - Easy to describe but impossible to implement (OS has no way of knowing when each of the pages will be referenced next)
- **FIFO Page Replacement:** Maintain a linked list of all pages. Each page is associated with the time when that page was brought into memory. Page chosen to be replaced is the oldest page
    - Implementation is a FIFO queue
        - Head points to the oldest page
        - Tail points to the newest page
- **Least Recently Used (LRU) Page Replacement:** Associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time
    - Most OS's use LRU
- Other algorithms:
    - Least Frequently Used (LFU)
    - Most Frequently Used (MFU)

Long-term Information Storage: Three essential requirements:

- Must store large amounts of data
- Information stored must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently

Mass Storage Structures:

- **Magnetic Disks:** Provide the bulk of secondary storage for modern computer systems
    - **Layout:**



- **Magnetic Tape:** Was used as an early secondary story medium
    - Slow compared to magnetic disks & memory
    - Can hold large amounts of data
    - Read data sequentially
    - Mainly used for backup

Disk Pack: Multiple Disks: Think of disks as a stack of platters. Use both sides of platters (rather than 1 side)

- Two **read-write heads** at each end of arm
- **Cylinders:** Same track on each surface

Disk Operations Cost: Access time composed of:

- **Seek time:** Time to position head over correct track
- **Rotation time:** Time for correct sector to rotate under disk head
- **Transfer time:** Time to transfer data
- Usually seek time takes the longest

I/O Bus: A disk drive is attached to a computer by a set of wires called an I/O bus. Types of I/O bus include:

- Enhanced Integrated Drive Electrics (EDIE)
- Advanced Technology Attachment (ATA)
- Serial ATA (SATA)
- Universal Serial Bus (USB)
- Small Computer Systems Interface (SCI)

I/O Controllers: The data transfers on a bus are carried out by special electronic processors called controllers

- **Host controller:** Controller at the computer end of the bus
- **Disk controller:** Controller that is built into each disk drive

Network Attached Storage (NAS): Storage made available over a network rather than over a local connection

- Often a set of disks (storage array) are placed together
- NFS and CIFS are common protocols

Disk Scheduling: The disk accepts requests, what sort of disk arm scheduling algorithm is needed?

- **First Come First Serve (FCFS):**
  - **Pros:** Fairness among requests
  - **Cons:** Arrival may be on random spots on the disk (long seeks)
- **Shortest Seek Time First (SSTF):** Pick the one closest on disk
  - **Pros:** Minimize seek time
  - **Cons:** Starvation
  - Most often used
- Others:
  - SCAN / C-SCAN

RAID: Redundant Array of Independent Disks. Uses multiple disks attached to a computer system

- Provides reliability and redundancy
- Improve performance via **parallelism**
- Improve reliability via **information redundancy**
- Different scheme to provide redundancy, classified as **RAID Levels:**
    - **RAID-0:** Break a file into blocks of data, stripe the blocks across disks in the system
        - Provides no redundancy or error detection
        - Loss of 1 disk means all data among both disks is lost
        - Allow to use 2 disks as 1. Used in systems where fasts reads are required but minimal data integrity is needed
    - **RAID-1:** A complete file is stored on a single disk, the second disk contains an exact copy of the file
        - Provides complete redundancy of data
        - Write performance suffers (must write data twice)
        - Most expensive implementation (need twice as much storage space)

File Attributes: Name, type, location, size, protection, creation time, etc

File Naming: Files with a certain standard structure imposed can be identified using an **extension** to their name

File Operations: Create, write, read, delete, etc

- For write/read operations, the OS needs to keep a **file position pointer** for each process

Directory Operations: File search, creating, deletion, renaming, directory listing

File Systems:

- Unix uses **Unix File System (UFS)**
- Windows **NT File System (NTFS)**, FAT, FAT32

File Control Block (FCB): Information about a file may be maintained in an FCB

- One FCB per file
- A FCB is associated with a unique id
- Consists of details about a file (permissions, dates, owners/groups, size, etc)

File System Structures:

- **System-wide Open File Table:** Contains a copy of the FCB of each open file
- **Per-process open-file table:** Contains a pointer to the appropriate entry in the system-wide open-file table

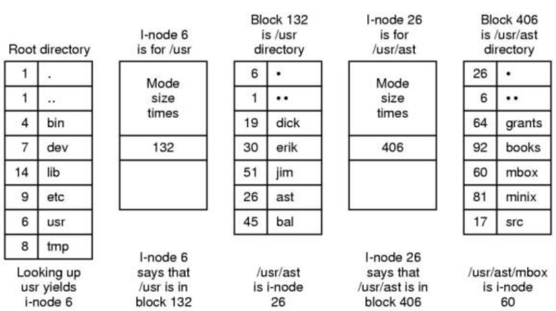
File System Layout: File systems usually are stored on disks. Most disks can be divided up into partitions

- Sector 0 of the disk is the **Master Boot Record (MBR)**: The end of the MBR contains the partition table

Allocation Methods: How to allocate space on disks for files?

- Files are a sequence of bytes, and disks are arrays of sectors (**512 bytes**)
- **Block Size:** Usually consists of a number of sectors
- File systems view the disk as an array of blocks
- **Contiguous Allocation:** Start and length are stored in the FCB
    - Easy to implement and excellent read performance
    - Causes fragmentation and will need periodic compaction (Defrag)
- **Linked List Allocation:** Each file is a linked list of disk blocks. The directory contains a pointer to the first and last block of the file. Each block contains a pointer to the next block and the last block contains a NIL pointer (-1)
    - No fragmentation
    - Random access is slow
- **Indexed Allocation:** Brings all pointers together into an index block. Index block contains all indexes of the block

Entry Lookup: The **superblock** has the location of the i-node which represents the root directory. Once the root directory is located a search through the directory tree finds the desired directory entry (block)



- The steps in looking up /usr/ast/mbox

Free Space Management: Limited disk space, so we need to reuse the space from deleted files

- **Bitmap:** One bit for each block on the disk. If the block is free the bit is 1 else the bit is 0. Makes it easy to find a contiguous group of free blocks
    - Used in Apple machines
- Uses Linked List, Indexing, or a combination of the 2

NFS: Network File System

- **NFS File Server:** Runs on a machine (that may have large disks) that has a local file system
- **NFS Client:** Runs on an arbitrary machine and access the files on machines that run NFS servers
- **Mounted:** Means that a directory can appear in the tree structure
- **Export List:** Maintained by server, list of local directories that server exports for mounting and names of machines that are permitted to mount them

NFS Mount Protocol: Following a mount request that conforms to its export list, the servers returns a **file handle** (a key for further access)

- The mount operation changes only the user's view and does affect the server side