Jacob Chmura

Introduction

Definition 1. An **operating system** is the interface between user and hardware. It is concerned with the allocation of resources (memory, processors, devices), memory management, I/O , file systems, and privacy.

Types of Operating Systems

- Batch Operating System: does not interact with computer directly. The operator takes similar jobs, groups into batches, and executes.
- Time Sharing Operating System: each task is given some time to execute. After this time interval, OS switches to next task.
- Distributed Operating System: multiple computers interconnected through network, each with their own process power and memory work together giving the appearance of a single computer.
- Real Time Operating System: strict time requirements on task, often very small.

Types of Parallelism

- Multiprogramming: multiple programs in memory at the same time for execution. Initially, these processes or jobs or kept in the job pool, waiting for allocation to main memory and CPU. As soon as one Job goes for an I/O task, the OS interrupts that kob, and choosing another job from the job pool, giving resources.
- Multiprocessing: a computer using more than CPU at a time.
- Multithreading: a thread is a basic unit of CPU utilization. In multi-threading, a single process can have multiple code segments running concurrently within the context of that process. They share parent process resources but execute independently.

Types of Memory

- Primary memory (RAM and ROM)
  - RAM: volatile memory (static random access memory, dyamic random access memory). SRAM is used for cache, and DRAM is used for main memory.
  - ROM: non-volatile read only memory (PROM, EPROM, EEPROM)
- Secondary memory (hard drive, CD)

Boot Process

The *bootloader* is pulled into memory and started, and is tasked with starting the real operating system.

Functions of BIOS

- Power on self test: initialize devices, create device tables, test RAM, CPU, secondary storage
- Master boot record: a special sector of disk storing bootloader code
- Init: looks for file /etc/inttab for run-level of the system (system halt, single user mode, multiuser without network, multiuser with network

System Structure

Definition 2. The **kernel** is the part of the operating system that manages system resources, and is the first program to load up after the bootloader.

Kernel Mode vs. User Mode A CPU can execute certain instruction only when it is in kernel mode: *privileged instructions*.

The OS puts the CPU in kernel mode when it is doing so, and then puts it in back in user mode when complete.

Systems calls are implemented in the form of *interrupts* which causes the mode bit to kernel mode in the process.

User programs that attempt to execute illegal instructions or to access forbidden memory also generate interrupts which are trapped by the handler, to which the OS reports error.

Definition 3. A **microkernel** is one type of kernel, where the *user services* and *kernel services* are implemented in different address spaces.

Kernel IO Subsystem

- I/O Scheduling: maintain a wait queue of the request for each device. When an application issue a blockin IO system call, teh request is placed in the queue for that device.
- Buffering: a *buffer* is a memory area that stores data being transferred between two devices. We use buffering because:
  - Cope with speed mismatch between producer and consumer of a data stream
  - Adapt to data that has different data-transfer size
  - Support copy semantics for the application
- Caching: a *cache* is a region of fast memory that holds a copy of data.
- Spooling: a *spool* is a buffer that holds the output of a device such as a printer, that cannot accept interleaved data streams

Definition 4. A **monotholic kernel** is another classification of a kernel, but unlike a microkernel, the user services and kernel services are implemented under the same address space.

Process Management

Processes

Definition 5. A **process** is an instance of a program in execution.

Process memory is divided into four sections

- Text section: comprises the compiled program code, read in from non-volatile storage when program is launched
- Data section: stores global and static variables, allocated and initialized prior to execution of main
- Heap: used for dynamic memory allocation, managed via cllas to new, delete malloc, free
- Stack: local variables, freed when variables go out of scope. If the stack every meats the heap, either a stack overflow error will occur, or call to malloc will fail.

When processes are swapped out of memory and later restored, additional information like the program counter and value of all program registers are kept.

Processes may be in one of five states:

- New: process is in stage of being created
- Ready: process has all the resources available that it needs to run, but the CPU is not currently working on the process instructions
- Running: the CPU is working on the process instructions
- Waiting: the process cannot run at the moment, and it is waiting for the required resources.
- Terminated: process complete

Process Control block

Each process has a *process control block*, PCB, which stores the following process specific information

- Process state: running, waiting etc
- Process ID, and parent process id
- CPU registers and program counter: these need to be saved and restored when swapping processes in and out of CPU
- CPU scheduling info: priority informaiton and pointers to scheduling queues
- Memory management information: page tables and segment tables
- Accountign information: user and kernel CPU time consuemd, account numbers, limit
- I/O Status information: devices allocated open file tables

Process Scheduling

The two main objectives of the scheduling system are to keep the CPU busy at all times and deliver fast response times for all programs.

All processes are stored in the *job queue*, and processes that are ready are put in the *ready queue*, and processes waiting for a device are put in the *device queue*.

A **long term scheduler** is a typical batch system that runs infrequently, whereas **short term scheduler** runs very frequently and must be very quick to swap one process out of the CPU.

A good scheduling system will select a good process mix of CPU-bound and I/O bound processes.

Whenever an interrupt arrives, the CPU must do a *save-state* of the currently running process, then switch into kernel mode to handle the interrupt, and then do a *state restore* on the interrupted process. A **context switch** occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This involves restoring all the registers and program counters as will as the process control block.

Operations on processes

Process may create processes through system calls *fork* or *spawn*. The process that does this is the parent of the newly created child.

Each process is given a unique identifier, the PID, and holds the parent process id PPID as well.

The initializing process scheduler has PID = 0.

The parent process may:

- Wait for the child process to termiante before procedding, by making a wait() system call, which causes a block.
- Run concurrently with teh child, continuing to process without waiting.

The address space of the child may be:

- An exact duplciate of the parent, sharing the same program and data segments. (each will have their own PCB, program counter, registers and PID). This is the behaviour of fork() system call
- The child process may have a new program loaded into its address space with all new code and data segments. This is the behaviour of calling exec() system call after fork.

Processes may request their own terminated by making the exit() system call. The return int is passed to parent if it is doing a wait().

A process may also be terminated by the operating system if it cannot supply the resources, by invoking the kill command.

The unix nohup command allows a child to continue executing after its parent is exited.

When a process is terminated, all system resources are freed. Process which are trying to termianted but which cannot because their parent is not waiting are called *zombies*. These are eventually inherited by init as orphans and killed off.

Interprocess Communication

Independent processes operating concurrently on systems are those that cannot affect or be affected by other processes

Cooperating processes are those that can affect or be affected by other processes

- Information sharing: many processes may need access to same file
- Speed: problem can be solved faster if it can be broken down into sub-tasks running simultaneously
- Modularity, convinience

There are two types of inter-process communication: *shared memory* and *message passing*

Shared memory is faster once it is set up since no system calls are required and access occurs at normal memory speeds. This is proffered when large amounts of information must be shared quickly on the same computer. A typical example is a producer-consumer model with a shared buffer queue.

Message passing requires system calls for every message transfer and is slower, but works well across multiple computers. Message passing is preferable when the amount and or frequency of data trasnfers is small.

There are three distinct issues that need to be resolved in message passing systems:

- Direct or indirect communication: in direct communication the sender must know the name of the receiver to which it wants to send message. In symmetric communication, the receive must also knoew the name of the sender (but not in asymmetric communication). In indirect communcation, we use ports. Multiple process can share the same port.
- Synchronous communication is blocking, asynchonours communication is non-blocking
- Buffering: messages are passed via queues which can have zero capacity (senders must block until receivers accept), bounded capacity (pre-defined finite capacity queue, so senders must block if queue is full until space is available), or unbounded capacity (senders never forced to block).

Exampled of IPC systems

- POSIX Shared memory
  1. One of the process needs to allocate sharde memory using shmget()
  2. Any process which wishes to use shared memory must attach the sharted memory to their address space using shmat()
  3. The process may access the shared memory using the void pointer returned by shmat
  4. When a process no longer needs a piece of shared memory, it can be detached using shmdt
  5. The process that orgiinalyl allocated the shard memory can remove it from the system using shmctl

Communication in Client Server systems

A **socket** is an endpoint of communication. Two processes communicating over ntwork often use a pair of connected sockets as a communication channel. A socket is identified by an IP address concaetenated with a port number.

Communication via sockets may be:

- Connection oriented TCP: emulated telephone connections. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent, re-send packets if necessary, and arrange the received packets in the proper order before delivering.
- Connection-less UDP: emulates individual telegrams. There is no guarentee that any particular packet will get through undamaged or at all, and no guarentee that the packets will get delivered in any order. These are much faster than TCP but applications must implement their own error checking and recovery procedures.

Remote procedure calls (RPC) make procedure calls that lie on a remote machine. Implementation involves *stubs* on either end of the connection.

The local process calls the stub, the RPC system marshals the parameters to the procedure call, and transmits them to the remote system. On the remote side, the RPC daemon accepts hte parameters and does the work. The results are packaged and send by the RPC system back to the local system.

A common example of a system based on RPC calls is a networked file system.

Pipes are the simplest channels of communication between processes. We can have unidirectional or bidirectional pipes. Bidirectional pipes may be half-duplex or full duplex. Some pipes require parent-child process relationships, some pipes work over a network.

The pipes are accessed as files, using standard read() and write() system calls.

Named pipes support bidirectional communication, communicating between non-parent child related process, and persistence after process termination. Multiple processes can share a named pipe, typically one reader and multiple writers.

Threads

A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, a set of registers and a thread id. Traditional processes have a single thread of control (one program counter, and one sequence of instruction that can be carried out at any given time).

- Useful when a process has multiple tasks to perform independently of the others
- Responsiveness: one thread may provide rapid response while other threads are blocked
- Resource sharing: by default threads share common code, data and other resources which allow multiple tasks to be performed in a single address space
- Economy: creating and managing threads is much faster then multiprocessing
- Scalability: a single threaded process can only run on one CPU, no matter how many are available, whereas the execution of a multi-threaded application may be split amongst available processors

Multi-threading models

User threads are supported above the kernel, and *kernel* threads are supported within the kernel of the OS.

In a *many-to-one model* many user level threads are all mapped onto a single kernel thread.

In a *one-to-one* model created a separate kernel thread to handle each user thread.

Implicit threading

Thread pools: creating new threads every time one is needed and then deleting it when it is done is inefficient. An alternative solution is to create a number of threads when the process first starts, and put those threads in a thread pool.

Threads are allocated form the pool as needed, and returned to the pool when nolonger needed.

Open MP is a set of compiler directive available in C, C++ that instruct the compiler to automatically generate parallel code where appropriate.

Process Synchronization

In a number of cooperating processes, each has a *critical section of code*

- Only one process in the group can be allowed to execute in the critical section at any one time

A solution to the critical section problem must satisfy the three conditions:

1. **Mutual exclusion**: only one process at a time can be executing in their critical section
2. **Progress**: if no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the process not in the remainder sections can participate in the decision, and the decision cannot be postponed indefinitely

3. **Bounded waiting**: their exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section, and before that request is granted.

Petersons Solution to the critical section problem

We store two shared data items:

- int turn: indicated whose turn it is to enter into the critical section
- bool flag[2]: indicated when a process *wants to* enter into their critical section

Most apis provide API equivalent of *mutex locks* which involve *acquire()* a lock prior to entering a critical section, and *release()* it when exiting.

The acquire step will block the process if the lock is in use by another process, and both the acquire and release operations are atomic.

A *spinlock* is a type of lock that CPU sits and spins while blocking the process, and wastes CPU cycles.

A more robust alternative to simple mutexes is to use *semaphores* which are integer variables for which only two atomic operations are defined, the *wait* and *signal* operations.

Binary semaphores can take on one of two values, and an be used to solve the critical section problem, and can be used as mutexes on systems that do not provide a separate mutex mechanisms.

Counting semaphores can take on any integer value, and are usually used to count the number of remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can enter a critical section. When the counter gets to zero, the process blocks until another process frees resources and increments the counting semaphore with a signal.

An alternative approach to spinlock is to block a process when it is forced to wait for an available semaphore, and swap it out of CPU. In this implementation, each semaphore needs to maintain a list of processes that are blocked and waiting, so that one of the processes can be woken up and swapped back in when the semaphore becomes available.

Classic Synchronization problems

Bounded buffer problem: generalization of producer-consumer wherein access is controlled to a shared group of buffers of a limited size.

```
do {
   . . .
   // produce an item in nextp
   . . .
   wait(empty);
   wait(mutex);
   . . .
   // add nextp to buffer
   . . .
   signal(mutex);
   signal(full);
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

```
do {
   wait(full);
   wait(mutex);
   . . .
   // remove an item from buffer to nextc
   . . .
   signal(mutex);
   signal(empty);
   . . .
   // consume the item in nextc
   . . .
}while (TRUE);
```

Figure 5.10 The structure of the consumer process.

Figures 5.9 and 5.10 use variables next_produced and next_consumed

Reading-writer problem: reader processes read the shared data and never change it, writer process may change data. There is no limit to how many readers can acess the data simultaneously but when a writer accesses the data, it needs exlusice access.

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```


Figure 5.12 The structure of a reader process.

Dining philosohpers problem: acces of limited resources among a group of processes in a deadlock-free and starvation free manner.

Atomic DB Transactions

Db ops need to carry out *atomic transactions* in which the entire trasnaction must either complete, or not occure at all (rollback).

Log based recovery: before each transaciton is conducted an entry is writen to alog on stable storage (each transaction has a unique serial number, first entry is start, every data changin entry specifies transaction number, old value and new value, final entry is commit). All transactions are idempotent. After a crash, any transaction which has a commit recorded in the log can be redone from log info. Any which has start but not commit can be undone.

Log based recovery can be slow, so often one esablishes checkpoints, whereas we periodically write all data to after checkpoint to stable sotrage, and then write a checkpoint entry to a log.

CPU Scheduling

A scheduling system allows one process to use CPU while the other is waiting for I/O, thereby making full use of otherwise lost CPU.

Preemptive Scheduling

CPU Scheduling take place under one of four conditions

1. When a process switches from the running state to waiting state
2. When a process switches from the running state to ready state
3. When a process switches from waiting state to ready state
4. When a process terminated

Conditions 1,4 have no choice and must select a new process.

Conditions 2,3 we can continue the running process, or select a different one.

If scheduling takes place only under conditions 1,4, the system is **non-preemptive**: if a process starts running it keeps running.

Otherwise, the system is said to be **preemptive**

Dispatcher

The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. It involves switching context, switching to user mode, jumping to the proper location in the newly loaded program.

Scheduling Criteria

- CPU Utilization: want high CPU usage not too waste cycles, but not always running at 100%
- Throughput: number of processes completed per unit time
- Turnaround time: time required for a process to complete, from submission time to completion
- Waiting time: how much time process spends in ready queue waiting for CPU
- Response time: time taken in an interactive program from the issuance of a command to start of a response of that command

Scheduling Algorithms

- First come first serve: FIFO Queue
- Shortest Job first serve: best scheduling but must estimate runtime
- Round robin serve: serve process with timer, and swap to back of queue if not complete by the end time, using a circular queue
- Multi level queue: use a set of queues, each implementing a different scheduler,and assigning jobs to each queue based on its type

Thread Scheduling

Contention scope: refers to the scope in which threads compete for CPU. **Process contention scope** occurs in many-to-one and many-to-many threads, when competition occurs between threads that are part of the same process. **System contention scope** involves the system scheduler scheduling kernel threads to run on one or more CPUs. Systems implementing one-to-one threads use only system contention scope.

MultiProcessor Scheduling

If a process where to switch from one processor to another, the data in the cpu cache would need to be invalided and re-loaded from memory, making the cache useless. Therefore, **symmetric multiprocessing** systems attempt to keep processes on the same processor via **processor affinity.**

Load Balancing

Push migration involves a sperate process that runs periodically and moves processes from heavily loaded processors onto less loaded ones

Pull migration involves idle processes taking processes from the ready queues of other processors.

There are two ways to multi-thread a procesor:

1. Course grain multithreading switched between threads only when one threads block, say on a memory read
2. FIne grain multithreading occurs on smaller regular intervals, say on the boundary of instruction cycles

Deadlocks

A set of processes is **deadlocked** when every process in the set is waiting for a resource that is currently allocated to another process in the set and can which can only be released when that other waiting process makes progress.

There are four conditions that are necessary to achieve deadlock

1. Mutual exclusion: at least one resource must be held in a non-sharabe mode
2. Hold and wait: a process must be simultaneously holding at least one resource and waiting for at least one resource currnetly being held by some other process
3. No premption: once a process is holding a resource then that resource cannot be taken away from that process until the process voluntarily realises it
4. cicular wait: a set of process must exist such that every process is waiting for the next in a circular ordering

Deadlock prevention

We can prevent at least one of the four conditions

- Mutual exclusion: shared resources such as read-only files do not lead to deadlocks
- Hold and wait: don't allow a process to request for more resources while holding others
- No premption: pre-empt in some cases
- Circular wait: order resource and require all processes request resourecs in strictly increasing order

We need to store additional resources allocation state: number of available and allocated resources, and max requirements of all processes in the system

Memory Management

Main Memory

Background

From memory chip point of view, all memory accesses are equivalent.

The CPU can only access its registers and main memory.

Memory access to registers are very fast, usually one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick. Memory access to main memory is slow, and would be intolerable if not for intermediary fast memory cache. The cache transfers chunks of memory at a time from main memory to cache, and then CPU looks at the cache first, trying to minimize cache misses before looking into main memory.

Address Binding

When user programs refer to memory addresses with variable names, they are mapped to physical memory addresses in stages:

- Compile time: if known at compile time where a program will resied in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses.
- Load time: compiler must generate relocatable code, which references addresses relative to start of the program.
- Execution time: if program can be moved around in memory during execution, then binding must be delayed to runtime

Logical vs Physical Address space

The address generated by the CPU is a **logical address**, whereas the address actually seen by the memory hardware is a **physical address**

The logical address is also known as the **virtual address**

The **memory management unit** maps logical to physical addresses.

Dynamic linking and shared libraries

With **static linking**, library modues get fully included in executable modules, wasting both disk space and main memory usage because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code/

With **dynamic linking** only a stub is linked into the executable module, containing references to the actual library module linked at run time. This saves disk space. Also, when a program uses a routine from a standard library, and the routine changes, then the program need not be re-built

Swapping

A process must be loaded into main memory to execute. If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the **backing store**.

Modern OS no longer use swapping because it is too slow and paging is faster.

Contiguous Memory Allocation

One approach to memory manaegment is to load each process into a continuous space, dividing all available memory into equal sized partitions and assign each process to its own partition. An alternative is to use bestfit,firstfit,worstfit allocation strategies.

All these suffer from **external fragmentation** where available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.

Segmentation

Memory segmentatoin supports the segments of memory (stack,heap,code,data) by providing addresses with a segment number, and an offset from the beginning of that segment.

A **segment table** maps segment offset addresses to physical addresses.

Paging

Paging is a memory mangement scheme that allows processes physical memory to be discontinuous and which eliminated problems with fragmentation by allocating memory in equal sized blocks known as **pages**.

We divide physical memory into a number of equal sized blocks called **frames** and divide a programs logical memory space into blocks of the same size called **pages**.

The **page table** is used to look up what frame a particular page is stored in at the moment.

A virtual address consists of a page number in which the address resides, and an offset from the beginning of that page.

The physical address consists of a frame number and the offset within that frame.

Larger page sizes waster more memory, but are more efficient in terms of overheaad. We can use multi-sized page tables.

Processes are blocked from accessing anyone elses memory because all of their memory requests are mapped through their own page table. Whenevre a process is swapped in or out of CPU, its page table must be swapped in orout too, along with the instruction registers.

A **page table base register** stores page table in main memory using a single register. This makes process switching fast, but every memory access now requries *two* memory accesses - one to fetch the frame number from memory, and another to access the desired memory location.

The **translation look aside buffer** is a high speed memory device that can search the page table for a key in parallel, and worked like a cache.

Structure of the page table

Most modern computer systems support a logical address space of 2^{64}. This requires hierarhical page tables, since the page table size is too large to keep in contiuous memory.

Typically page tables are either implemented using a hash table.

Virtual Memory

Background

We saw that we can avoid memory fragmentation by breaking process memory down into smaller bites (pages), and storing the pages non-contiguously in memory.

However, the entire process still had to be stored in memory somewhere.

In practice, most real processes do not need all their pages, or at least not at all once.

Virtual memory allows one to write programs for much larger address spaces, we can improve CPU utilization and throughput, and less I/O needed for swapping processes in and our of RAM.

Moreover, virtual memory allows the sharing of files and memory by multiple processes in virtual address space. For example, process pages can be shared during a fork() system call, eliminating the need to copy all the pages of the parent process.

Demand Paging

Demand paging: when a process is swapped in, pages are not all swapped at once, rather only swapped in when the process needs them.

If a page is needed that is not originally loaded up, then a **page fault** trap is generated and then:

- Requested memory address is checked for validity
- A free frame is located
- A disk op is scheduled to bring in the necessary page from disk
- When I/O complete, the process's page table is updated with the new frame number

Copy on write

Copy on write fork is that the pages for a parent process do not have to be actually copied for teh child until one or the other of the processes changes the page. They can simply be shared between the two processes in the meantime.

Page replacement

In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load pages that are actually neede by each process at any given time, there is room to load many more processes than if we had to load in the entire process.

Page replacement finds some page in emmory that isn't begin used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it.

The frame that is swapped out is called the *victim frame*.

The goal of the **frame-allocation and page replacement algorithm** is to minimize the number of page faults.

A simple algorithm is to use FIFO replacement. But an optimal page replacement algorithm is called **MIN** which *replaces the page that will not be used for the longest time in the future.*

The estimation is often done using **LRU: least recently used** algorithm. While the FIFO uses the oldest **load** time, the LRU uses the oldest **use** time.

Allocation of Frames

The frame allocation strategy can be:

- Min number of frames: allocate the minimum required for each process
- Equal allocation: divide all available pages evenely by processes

Thrashing

If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. A process that is spending more time paging than executing is said to be **thrashing**.

To avoid thrashign we must provide processes with as many frames as they need *right now.* The **locality** model notes that process typically access memory references in a given **locality** making lots of references to the same area of memory.

The **working set model** is based on locality, and defines a **workings set window of length** δ. Whatever pages are included in the most recent δ pages are included in teh msot recent delta page references are said to be in the processes working set window.

Allocating Kernel Memory

Kernel memory is typically locked (restricted from ever being swapped out), so we need good algorithms for managing memory.

The **buddy system** allocated memory using a **power of two allocator**.

Storage Management

File System Interface

- Attributes: Name, inode number, type, location, size, protection, time, user id
- Operations: create file, write file, read file, delete file, truncate file
- Access methods: sequential access, direct access
- Directory ops: search for a file, create file, delete file, list directory, rename file, traverse file system
- File sharing: owner, group, others
- Protection: read, write, execute

File System Implementation

File systems organize storage on disk drives and can be viewed as a layered design.

- Lowest level physical devices: magnetic media, motors and controls
- I/O Control device drivers: assembly written software program which communicate with hardware using teh controller card's register
- Basic file system: works with device drivers in terms of retrieving and storing raw blocks of data
- File organization module: knows about files and their logical blocks, how they map to physical blocks on the disk
- Logical file system: all the meta data associated with a file (everything except file data itself), managing directory structure, and mapping file names to **file control block**

File systems store several data structures on disk:

- **boot control block**: the boot block about how to boot the system, generally first sector of the volume
- **Volume control block** the master file table which contains information about the partition number, number of blocks on each filesystem, and pointers to free blocks in the FCB
- **Fil control block**: contains details about ownership, size, permissions, dates in inodes
- **System wide open file table** containing a copy of the FCB for every currently open file in the system
- **Per-process open file table** containing pointer to the process open file table

Virtual File Systems

Virtual file systems provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier for files across the entire space.

- Inode: represents an individual file
- File: represents an open file
- Superblock: represents a filesystem
- Denty: represents a directory entry

Directory Implementation

Directories need to be fast to search insert, delete, with minimum wasted space.

A hash table is often used in addition to a linear list.

Allocation methods

- Contiguous: requires that all blocks of a file be kept together. Performance is fast but may lead to fragmentation, and issues when file sizes grow or cannot be estimated
- Linked allocation: disk files can be stored as linked lists, leading to no external fragmenetaiton, and allows files to grow dynamically. Random access is very slow.
- Indexed allocation: combines all of the indexes for accessing each fiel into a common block as opposed to spreading them all over the disk

Recovery

Using in memory and cached datastructures is fast but volatile memory is lost in system crash. A **consistency checker** is often run at boot time, to check disk blocks numbers, inodes fields, link counts, and directory structures are consistent. **Journaling** is a log-based transaction oriented filesystem that works like a database, guarenteeing that any given transaction either completes successfully or can be rolled back to a safe state before the trasnaction commenced.

I/O Systems

0.0.1 I/O Hardware

Devices connect with the computer via **ports**. A common set of wires connecting multiple devices is termed a **bus**.

The PCI bus connects high speed high bandwidth devices to the memory subsystem.

The Expansion bus connects slower low bandwith devices which typically deliver data one character at a time.

In **memory mapped I/O** a certain portion of a process address space is mapepd to the device, and communicaton occur by reading and writing directly to and from those areas. This is often used in graphics card.

For devices that trasnfer large quantities of data (disk controllers) it is wasteful to tie up the CPU trasnferring data in and out of registers one byte at a time. The **direct memory acces controller** handles large data trasnfer and then interuupts the CPU when trasnfer is complete. While DMA transfer is going on, CPU does not have access to the PCI bus (inclding main memory) but it does have access to its internal registers and primary and secondary caches.

Kernel IO Subsystem

Buffering of IO is performed when there are speed differences between devices, different data trasnfer sizes and to support *copy semantics*.

Caching involves keeping a *copy* of data in a faster acces locaiton than where the data is normally stored.

Spooling (simultaneous peripheral operations on line) buffers data for peripheral devices such as printers that cannot support interleaved data streams.

Protection and Security

Protection

Security