Process: program in execution. Can be **I/O Bound** or **CPU Bound**

- Each process has its own copy of data
- **Cascade termination**: when a parent terminates, its children are forcibly terminated

Activision Record: function params, local vars, return address pushed to stack when a function is called

PCB: contains info about a process. Used in **context switching** when current state saved to PCB and new state is loaded

Interprocess Communication through **shared memory** (shared region of memory) or **message passing** via communication link

- **Direct communication** requires naming recipient/sender. **Indirect communication** has messages are sent to **mailboxes**
- Messages might also be buffered

Threads share text/code section, data section, and OS resources (e.g. open files and signals)

- Challenges include **identifying task**, **load balancing**, **data splitting**, **data dependency**
- **Implicit threading** done through **thread pools**, **fork join**, **OpenMP**, **Grand Central Dispatch**
- **Concurrency** has multiple tasks making progress. **Parallelism** has multiple tasks working simultaneously
- Issue of semantics where should `fork()` copy all threads or just one thread in the context of if `exec()` will be called

Benefits of multithreading: **responsiveness**, **resource sharing**, **economy**, **scalability**

Data parallelism performs same operation on subsets of the data whereas **task parallelism** distributes tasks across multiple cores

User threads are mapped to **kernel threads** through **many-to-one**, **one-to-one**, or **many-to-many** models

Asynchronous threading: parent and child threads run concurrently. **Synchronous**: parent waits for children to terminate

Signal: notifies process that an event has occurred, received **synchronously** or **asynchronously**

- For multithreaded programs, can deliver signal to target thread, every thread in process, or thread assigned to receive all signals

Thread cancellation: terminating a thread **asynchronously** (immediately) or **deferred** (let it terminate on its own)

Thread Local Storage (TLS): copy of certain data unique to each thread

Lightweight Process (LWP): schedules user thread to run on an attached kernel thread

- Kernel thread notifies user thread of events through **upcalls**

Cooperating Process affect other processes and can share same logical address space or share data through IPC mechanisms

- May result in **race conditions** where several process manipulate the same data concurrently, creating varying outcomes

Critical section: code that is accessing data shared by other processes. Consists of **entry**, **exit**, and **remainder section**

- Must satisfy **mutual exclusion**, **progress**, **bounded waiting**
- In single-core environment, disable interrupts. For multicore environments, use **preemptive** or **nonpreemptive kernels**
  - **Preemptive** can lead to race conditions. **Nonpreemptive** prevents race conditions from happening

Peterson's Solution: 2 processes share `turn` and `flag` vars: whose turn it is to enter critical section and if the process is ready

Mutex Lock: protects critical sections and prevents race conditions by having processes `acquire()` and `release()` the lock

Busy wait: process that try to enter their critical section are continuously calling `acquire()`, wasting CPU cycles

- **Spin locks** avoid context switching so are preferred for short busy waits

Semaphore: integer variable accessed using `wait()` (decrement) and `signal()` (increment). Either **counting** or **binary semaphore**

- To avoid busy wait, `wait()` can suspend the process if `semaphore <= 0`. It will restart once another process executes `signal()`

Monitor: uses abstract data type to define operations with mutual exclusion and variables that contain the data type state

- Ensures only one process at a time can be active within the monitor

Liveness: properties system must satisfy to ensure progress. Possible issues

- **Deadlock** that depends on **mutual exclusion**, **hold and wait**, **no preemption**, **circular wait**
- **Priority Inversion**: higher priority process waiting for a local process to release a resource, but is preempted by another process

Livelock: thread attempts an action that constantly fails

Deadlock Prevention involves ensuring one of the 4 characteristics cannot hold

Deadlock Avoidance ensures actions result in a **safe state** (resource allocation graph with claim edges or **Banker's Algorithm**)

Deadlock Detection uses wait-for graph. **Deadlock Recover** considers **selecting a victim**, **rollback**, and **starvation**

Multiprogramming: load multiple programs into memory and always have a process running selected by the **CPU Scheduler**

- **Dispatcher**: involves context switching (**dispatch latency**), jumping to user mode, jumping to location in user program

Scheduling Criteria: **CPU utlization**, **throughput**, **turnaround time**, **waiting time**, **response time**

Scheduling Algorithms: **FCFS** (may cause **convoy effect**), **SJF** (optimal), **Round Robin**, **Priority**, **Multilevel Feedback**

Process Contention Scope (PCS): competition for CPU takes place between threads in the same process

System Contention Scope (SCS): competition for CPU takes place between all threads in the system

Multiprocessor scheduling: **Asymmetric** (one processor does all scheduling) or **Symmetric** (each process does its own scheduling)

Memory Stall: processor spends time waiting for data to become available (e.g. **cache miss**)

Load Balancing done through **push** and **pull** migration

Threads have **processor affinity** with the processor they are working on, creating **warm cache**

Memory addresses are determined by **base** and **limit registers**

- **MMU** handles binding logical addresses to physical addresses

Dynamic Loading: routine is not loaded until it is called, minimizing total memory use

Contiguous Memory Allocation: each processes is contained in a contiguous section of memory (**first**, **best**, **worst** fit)

- Can have **external** (solved with **compaction**) or **internal fragmentation**.

Paging: break physical memory into **frames** and logical memory in **pages** that are loaded into memory when executing

- Pages stored in a page table that contains **PTBR** and **PTLR** values
- Page table entries have a **valid-invalid** bit to see if they are already in memory

Translation Look-aside Buffer (TLB): used to speed up page lookup. **TLB Miss** requires normal page lookup

Reentrant Code: code shared between processes, allowing them to execute the same code (each process has its own registers and data)

Page table can be organized as **hierarchical PT**, **hashed PT**, **inverted PT** (one entry per real frame)

Entire processes can be **swapped** in/out of **backing store**. Same idea can be applied to pages, resulting in **paging**

Virtual Memory separates logical and physical memory

Demand Paging involves only loading pages into memory when they are needed. Can result in **pages faults**

- Need to terminate current instruction, find a free frame, read desired page into it, and then restart the instruction
- **Swap Space**: secondary memory used to hold pages not in main memory
- **Free-frame List**: pool of free frames
- Need to consider **frame allocation algorithms** and **page replacement algorithms**

Copy On Write: parent and child share these pages until either process writes, then need to copy the page

Page Replacement: Free a **victim frame** by writing its content to swap space. Then perform page-fault routine

- Can use a **dirty bit** to reduce number of page transfers required
- **FIFO** (can result in Belady's anomaly), **Optimal** (replace page that will not be in the longest period), **LRU**, **LFU**, **MFU**
- Can involve **global replacement** or **local replacement**

Allocation of frames: need enough frames to hold all pages a single instruction can reference (otherwise results in **thrashing**)

- **Equal Allocation** and **Proportional Allocation** (based on process size)
- Thrashing avoided using **locality model**: pages used together are loaded simultaneously (approximated by **working set model**)

Non Uniform Memory Access (NUMA): main memory that is not created with equal access time

Memory Compression: compress several frames into a single frame, reducing memory use

Kernel Memory is implement as a memory pool because kernel requests vary in size

- **Buddy System** (allows for easy **coalescing**), **Slab Allocation** (kernel objects stored in **cache** and **slabs**)

Other considerations: **prepaging**, **page size** (smaller page has better memory use but requires more pages), **TLB reach**

Magnetic Drive (HDD) data access is done by moving head. **Flash Drive** (SSD) data access is done electronically

Accessing secondary storage involves **transfer rate** and **random access time** (**seek time** and **rotational latency**)

- Computers access secondary storage via **host-attached storage** or **network attached storage**

Data transfer is done on a bus through **host controllers** (computer side) and **device controller** (on storage device)

- Computer places command into host controller that sends is received by device controller to operate on hardware

HDD Scheduling optimized by ordering I/O requests: **FCFS, SCAN, C-Scan**

RAID is used to parallelize read/writes and improve data storage reliability through **redundancy** and **striping**

Device Drivers: uniform device access interface to I/O subsystem

Daisy Chain: device A is connected to device B is connected to device C that is connected to a port

Memory Mapped I/O: device control registers are mapped to address space of the processor

- Managed using **data-in**, **data-out**, **status**, **control registers**
- **Polling**: host and controller handshake when interacting (busy-wait on device status and interrupt notifies CPU of I/O completion)

CPU maintains an **interrupt-request line** that triggers an **interrupt handler routine** on **signal detection**

Direct Memory Access (DMA): host writes a DMG control block into memory with source and destination of transfer

- Rather than **Progammed I/O**: CPU feeds data into controller one byte at a time

2 types of **open-file tables**: **system wide file table** and **per process file table** (points to system wide table)

File type gives operating system hints of how to handle it (system must support code for this)

2 types of file access: **sequential access** and **direct access** (file broken into fixed size **records**)

- Can also use table and index into various blocks/records

Directory structures: **single level**, **2 level** (each user has its own UFD) with entry from (MFD), **tree structured**, **acyclic**

- Acyclic allows directories to share files through **links** that point to another location. System ignores links when traversing

For protection, system will duplicate files and manages **Access Control List (ACL)** that specifies access permissions

Memory Mapped Files: part of virtual address space is logically associated with file for faster access

- Maps disk blocks to a page in memory. First access causes a page fault, subsequent read/write from memory
- Multiple processes can map the same file, allowing for data sharing

Ways of allocating files:

- Contiguously: minimizes disk seeks and gives fast access but leads to external fragmentation. Can **defragment** but it is costly
  - Can also use **extents**: chunk of contiguous space linked together
- **Cleaner**: copies in-use files to end of log and defragments the log
- **Linked**: each file is a linked list of blocks scattered on disk (only useful for sequential seek and also issue of **reliability**)
- **File Allocation Table (FAT)**: beginning of each storage has a table that points to 1st block of files (speeds up random access)
- **Indexed**: all block pointers stored in an **index block**
  - **Linked Scheme** (links several index blocks) and **Multilevel index** (1st index block points to 2nd level index block)

Sticky Bit: indicates file should be kept in memory after the program terminates

Free Space Management: done using **bit vector** or **linked list** (of free blocks)

Recovery ensures system crash doesn't cause inconsistencies: **Log Structured File System** (all metadata/transactions written to logs) or **snapshot** (view of file system at a specific time)

Virtual File System (VFS): allows clients to access different types of file systems in a uniform way

LDT defines base and limit registers for segments. Since all users are in range $2-4$ GB, they can use same LDT entry

User can't access APIC since it needs protection so user can't change interrupt handling via APIC

Base/Limit registers require a contiguous allocation of memory

Hard link: 2 files with same inode. **Symbolic link**: store linked file name

`Alloc_Pageable_Page` needs virtual address to clear page table entry bit to say it's no longer in memory

Even though kernel is not pageable, page fault can still occur in a system call when the kernel page is not in memory

Needed to copy kernel page directory to user page directory since user may need to run in kernel mode for system calls

To make page table pageable, need to allocate page table from pageable memory, and on page fault, check if page table is present

Seek doesn't require disk head moving, only a pointer update