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 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**