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

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

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

- **Implicit threading** done through **thread pools** or **fork join**

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

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

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()`