2019 Spring - System Programming Midterm 2

Virtual Memory

- Why do we use VM?
  - Use main memory efficiently (caching)
  - Simplify memory management (each process has its own virtual address space)
  - Isolate address spaces (protection: extend PTE with permission bits)
- Address Translation
  - (Only with Page Table)
    Processor $\xrightarrow{\text{VA}}$ MMU $\xrightarrow{\text{PTEA}}$ Page Table $\xrightarrow{\text{PTE}}$ MMU. If hit, done.
    (Miss) Page Fault Exception/Handler fetches page and updates PTE.
  - (TLB)
    MMU uses the VPN portion of the VA to access the TLB
- **Translation Lookaside Buffer**
  - Small cache of page table entries in the memory management unit
  - Virtually addressed cache where each line holds a block consisting of a single PTE
  - **TLB hit** eliminates the memory accesses required to do a page table lookup[1]
- Linux Page Fault Handling
  - Is the VA legal? (Segmentation Fault)
  - Is the memory access legal? (Protection)
- Memory Mapping
  - VM areas initialized by associating them with disk objects
  - Area gets its initial values from *regular file* or *anonymous file*
  - Dirty pages are copied back and forth between memory and swap file

---

[1] If k-level page table is used, we need k memory accesses...

Dynamic Memory Allocation

- Why do we use dynamic memory allocation?
  - Acquire VM at runtime (size of data unknown until runtime)
  - Manage VM of each process (heap)
- Requirements and Goals for Memory Allocators
  - Requirements
    - Can't control number or size of blocks
    - Immediate response to requests
    - Allocation only in free memory
    - Alignment (8/16 byte - 32/64 bit machine)
    - Can't move allocated blocks (No compaction)
  - Goals (Maximize)
    - **Throughput**: Number of completed requests per unit time
    - **Peak Memory Utilization**: After k requests, aggregate payload P_k: the sum of currently allocated payloads, heap size H_k, then

$$U_k = \frac{\max_{i \leq k} P_i}{H_k}$$

- **Fragmentation**: Unused memory is not available to satisfy allocate requests
  - Internal Fragmentation: Payload is smaller than block size
    - Overhead of maintaining heap data structures
    - Paddings for alignment
    - Policy decisions ...
  - External Fragmentation: Enough free memory but no single block is large enough
    - Depends on future requests
- **Implict Free List**: Free blocks linked implicitly by the size fields in the headers
  - Header + Payload + (Optional) Padding
  - Header contains size and allocation status
  - Allocation status at the LSB of the header (Useable due to alignment)
  - Linear time allocation

- **Placement Policy**: Where to allocated the requested block?
  - Trading off throughput for less fragmentation
  - First Fit: Search whole list, choose first free block that fits
  - Next Fit: Start search where the previous search left off
  - Best Fit: Search whole list, choose the block with least fragmentation
  - Allocated space might be smaller than the free space → Split (maybe)
- **Coalescing**: Merge adjacent free blocks
  - *False Fragmentation*: Large enough free memory chopped up into small free blocks, and therefore unusable
  - Immediate Coalescing: Coalesce each time a block is freed
  - Deferred Coalescing: Do it some later time (ex. when some allocation fails, when scanning the list, when external fragmentation reaches some limit)
- **Boundary Tags**
  - Replicate header into footer
  - Used for coalescing previous blocks: constant time coalescing possible
  - Internal fragmentation: extra space for footer can use a lot of memory due to alignment.
  - **Fix**. For allocated blocks, use the second LSB to contain allocation status of the previous block and remove footer. (Free blocks still need footer)
- **Explicit Free List**: Maintain list of only *free* blocks
  - Structure: Header + <u>Pointers to prev/next free block</u> + (Empty Space) + Footer
  - (First fit) Allocation time is linear in the number of *free* blocks.
  - Insertion Policy: Where to put the newly freed block?
    - LIFO: Insert at the beginning
      - Simple and constant time freeing
      - Fragmentation may be worse
    - Address Ordered: Free list blocks are always in address order
      - Fragmentation may be lower than LIFO
      - But requires search...

- Explicit Free List: **Comparison to Implicit Free List**
  - Allocation is linear in the number of **free** blocks, not **all** blocks
  - More complicated to free block and insert (doubly linked list)
  - Extra space needed for the prev/next pointer, increases internal fragmentation
- **Segregated Free List**: Different free lists for different size classes
  - Different free lists for different size or range of sizes
  - Allocation Process (Segregated Fits)
    * Search for free block in the appropriate free list
    * If block found, split and place fragment on appropriate list (optional)
    * If not found, keep searching next larger class of free list
    * Still not found: request additional heap memory (call `sbrk`), allocate into the new memory and insert remaining block into appropriate size class
    * Freeing: Free and coalesce, insert into appropriate size class
  - **Advantages**
    * Higher throughput ($\log$ time for 2^k size classes)
    * Better memory utilization
      · First-fit search $\approx$ best-fit search of entire heap
      · Extreme case: Giving each block its own size class is equivalent to best-fit
- **Garbage Collection**: Automatic reclamation of heap storage
  - Garbage: allocated blocks not needed by the program
  - Certain blocks cannot be used if there are no pointers to them
  - Memory as a **graph**
    * Node: Each block
    * Edge: Each pointer
    * Root Node: Locations not in the heap the contain pointers to heap (ex. registers, locations on stack, global variables)
    * A node is **reachable** is there exists a path from any root to that node
    * Non-reachable node is garbage
- **Mark & Sweep** Garbage Collection

- Mark: Start at roots and set mark bit on each reachable block
- Sweep: Scan all blocks and free blocks that are not marked

- Conservative[2] Mark & Sweep in C
  - Pointers in C can point to the middle of the block
  - Use a balanced binary tree to keep track of all allocated blocks, with start of block address as key
  - Balanced-tree pointers can be stored in header (two additional words)

- Memory Bugs
  - Dereferencing illegal pointers
  - Reading uninitialized memory
  - Overwriting memory (Stack/Buffer overflows)
  - Referencing nonexistent variables
  - Double free
  - Referencing freed blocks
  - Not freeing blocks (memory leaks)

---

[2] C language does not tag memory locations with type information...

Network Programming

- Client Server Model
  - A **server** process and one or more **client** processes
  - Server manages some **resource**, provides **service** for clients
  - Client requests activate server
    1. Client sends request
    2. Server handles request
    3. Server sends response
    4. Client handles response
- Computer Networks
  - **Network** is a hierarchical system organized by geographical proximity
    * LAN(local area network): spans building/campus
    * Ethernet: Most popular LAN technology
  - **Ethernet Segment** consists of a collection of hosts connected by wires to a hub
    * Spans small areas: room, floor in a building
    * One end attached on a host, other end attached to a **port** on the hub
    * Each Ethernet adapter has 48-bit address (MAC address)
    * Host sends **frames**(chunk of bits) to any other host on the segment
    * Each frame includes some fixed number of **header** bits (identify source and destination), frame length, payload(data)
  - **Bridged Ethernets**: Multiple Ethernet segments connected into larger LANs
    * Bridges: set of wires and small boxes (...)
    * Spans entire buildings or campuses
    * Make better use of the available wire bandwidth than hubs
    * Clever distributed algorithm: Learn which hosts are reachable from which ports and then selectively copy frames from one port to another only when it is necessary
  - Multiple incompatible LANs can be connected by specialized computers called **routers** to form an **internet**
    * Each router has an adapter (port) for each network that is connected to

        * WAN(wide area network): Routers can also connect high-seed point-to-point phone connections
    - **Protocol Software** running on each host and router
        * Possible to send bits across incompatible LANs and WANs
        * Smooths out the differences between the different networks
        * Implements a **protocol** that governs how host and routers cooperate to transfer data
        * Provides **naming scheme**
            · Defines a uniform format for host addresses
            · Each host/router is assigned at least one internet addresses that uniquely identifies it
        * Provides **delivery mechanism**
            · Defines a uniform way to bundle up data bits into **packets**
            · Packet consists of header (packet size, address of src/dest) and payload (data bits from src)

- Transferring internet Data (Encapsulation)

    1. Client on host A copies data from the client's VA space into kernel buffer (system call)
    2. Protocol SW on A creates *LAN1 frame* by appending an internet header and a LAN1 frame header to the data (**Encapsulation**)
    3. LAN1 adapter copies the frame to the network
    4. Router's LAN1 adapter reads the frame from the wire and passes it to the protocol SW
    5. Router fetches the destination address from the *internet packet header* and uses it as an *index into a routing table* to determine where to forward the packet. Remove LAN1 frame header and append LAN2 frame header, pass the result to adapter
    6. LAN2 adapter copies the frame to the network
    7. Host B's adapter reads the frame from the wire and passes it to the protocol SW
    8. Protocol SW on B strips of packet header and frame header. Eventually copies the resulting data into the server's VA space when a read system call is invoked

- **Global IP Internet**

- Most famous example of an internet
- **TCP/IP Protocol** family
    * IP (Internet Protocol): Provides basic naming scheme and unreliable delivery capability of packets from host to host
    * UDP (Unreliable Datagram Protocol): Uses IP to provide unreliable datagram delivery from process to process
    * TCP (Transmission Control Protocol): Uses IP to provide **reliable** byte streams from process to process <u>over connections</u>
- Accessed via a mix of Unix file I/O and functions from the **sockets interface**

- Programmer's View of the Internet
    - Hosts are mapped to a set of 32-bit **IP addresses**
    - Each IP address is mapped to an identifier called Internet **domain names**
    - A process on one Internet host can communicate with a process on another Internet host over a **connection**
- **IP Addresses**
    - 32-bit address (IPv4) stored in an IP address struct `in_addr`
    - Stored in *network byte order* (big-endian byte order)
    - Unix provides functions that convert between host byte order and network byte order (`htonl`, `htons`, `ntohl`, `ntohs`)
    - Humans use **dotted decimal notation**
    - Application programs convert between IP addresses and dotted decimal strings using `inet_pton`/`inet_ntop`[3]
- **Internet Domain Names**
    - Numbers are hard to remember - defines **domain names**, and a mechanism that maps domain names to IP addresses
    - Domain names form a hierarchy, represented as a tree, subtrees referred to as subdomains
    - The mapping is maintained in a huge worldwide distributed database called **DNS** (domain name system), consisting of host entries.

[3] `p` for presentation, `n` for network

- **Internet Connections**
  - Clients and servers communicate by sending streams of bytes over **connections**
    * Point-to-Point: Connects a pair of processes
    * Full-Duplex: Data flows in both directions at the same time
    * Reliable: Stream of bytes are received in the same order it was sent
  - A **socket** is an endpoint of a connection
  - Socket address: `IPaddress:port` pair
  - **Port** is a 16-bit int that identifies a process
    * Ephemeral Port: Assigned automatically by client kernel when client makes a connection request
    * Well-known Port: Associated with some *service* provided by a server (ex. 80-http, 22-ssh ...)
  - A connection is uniquely identified by the socket address of its endpoints (**socket pair**) `(clientaddr:clientport, serveraddr:serverport)`
- **Sockets Interface**
  - Set of system level functions used in conjunction with Unix I/O to build network applications
  - **Socket** is an endpoint of communication to the kernel, a *file descriptor* to an application, that enables read/write from/to the network
  - Clients and servers communicate with each other by reading and writing to socket descriptors
  - Difference between regular file I/O: How the application "opens" the socket descriptors
  - **Socket Address Struct**
    * `sockaddr`, `SA`: Generic socket address for arguments to `connect`, `bind`, `accept`
    * `sockaddr_in`: Internet specific socket address (IPv4)
  - `socket` **function** (= application buffer allocation)
    * `int socket(int domain, int type, int protocol);`
    * Clients and servers use `socket` to create *socket descriptor*
    * Returns non-negative descriptor, only partially opened and cannot yet be used for reading and writing

    - This function is *protocol specific*. Use `getaddrinfo`
- `connect` **function** (= set a connection to a server)
    - `int connect(int clinetfd, const SA *addr, socklen_t addrlen);`
    - Client establishes a connection by calling `connect`
    - Attempt to establish a connection with server at socket address `addr`
    - `addrlen` is `sizeof(sockaddr_in)`
    - If successful, `clientfd` is ready for read/write
    - Resulting connection is characterized by the socket pair (`x:y, addr.sin_addr:addr.sin_port`) (`x`: Client's IP, `y`: ephemeral port that uniquely identifies the client process on the client host)
    - Using `getaddrinfo` is the best practice
- `bind` **function** (= bind socket to service)
    - `int bind(int sockfd, SA *addr, socklen_t addrlen);`
    - Asks the kernel to associate the server's socket address with a socket descriptor
    - Process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`
    - Writes to `sockfd` are transferred along connection whose endpoint is `addr`
- `listen` **function** (= tells the kernel that this will be a server socket)
    - `int listen(int sockfd, int backlog);`
    - Tells the kernel that the descriptor will be used by a server instead of a client
    - Converts `sockfd` from an active socket to a *listening socket* that can accept connection request from clients
    - `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before it starts to refuse requests
- `accept` **function** (= receive connection request)
    - `int accept(int listenfd, SA *addr, int *addrlen);`
    - Server waits for connection requests from clients by calling `accept`
    - Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`
    - Returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines

- **Connected vs. Listening Descriptors**
  - Listening Descriptor
    - Endpoint for client connection **requests**
    - Created once and exists for lifetime of the server
  - Connected Descriptor
    - Endpoint of connection between client and server
    - **New descriptor created each time** the server accepts a connection request
    - Exists only as long as it takes to service client
  - Distinction needed: Allows for concurrent servers that can communicate over many client connections simultaneously
- **Host and Service Conversion**
  - `getaddrinfo` **function**
    - Modern way to convert string representations of hostnames, host addresses, ports and service names to SA
    - Re-entrant, portable protocol-independent (IPv4/v6 both OK)
    - `int getaddrinfo(const char *host, char *service,`
      `const struct addrinfo *hints, struct addrinfo **result);`
    - Given host and service, returns `result` that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.
    - Client walks the list trying each socket address in turn, until the calls to `socket` and `connect` succeed
    - Server walks the list until calls to `socket` and `bind` succeed.
  - `addrinfo` struct
    - Contains arguments that can be passed directly to `socket` function
    - Points to a socket address struct that can be passed directly to `connect`, `bind`
  - `getnameinfo` **function**
    - `int getnameinfo(const SA *sa, socklen_t salen, char *host,`
      `size_t hostlen, char *serv, size_t servlen, int flags);`
    - Inverse of `getaddrinfo`, converts socket address to the corresponding host and service

    * Also re-entrant and protocol-independent

- **Sockets Helper**
  - `open_clientfd`
    * `int open_clientfd(char *hostname, char *port);`
    * Establish a connection with a server
  - `open_listenfd`
    * `int open_listenfd(char *port);`
    * Server creates listening descriptor that is ready to receive connection requests

- `accept` Illustrated
  1. Server blocks in `accept`, waiting for connection request on `listenfd`
  2. Client makes connection request by calling and blocking in `connect`
  3. Server returns `connfd` from `accept`, client returns from `connect`
  4. Connection between `clientfd` and `connfd` established

- **Web Servers**
  - Clients and servers communicate using `HTTP`
  - Web servers return *content* to clients (MIME)
  - Static Content: Contents stored in files and retrieved in response to an HTTP request
  - Dynamic Content: Content produced on-the-fly in response to an HTTP request
  - Web content is associated with some file managed by the Web server
  - **URL**(universal resource locator): unique name for each file
  - Clients use *prefix* to infer protocol, server, port
  - Servers use *suffix* to find file on the system or determine if request is for static/dynamic content

- **HTTP Request**
  - A *request line*, followed by zero or more *request headers*
  - `<method> <uri> <version>`
  - `method`: `GET`, `POST` ...

- `uri`: URL for proxies, URL suffix for servers (uniform resource identifier)
- `version`: HTTP version of request
- Headers: `<header name> : <header data>`

- **HTTP Responses**
  - A *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line `\r\n` separating headers from content
  - `<version> <status code> <status msg>`
  - `status code`: numeric status
  - `status msg`: corresponding English text
  - Headers: `<header name> : <header data>`

- **Serving Dynamic Content**
  1. Client sends request to server
  2. URI contains `/cgi-bin`, server assumes dynamic content
  3. Server creates child process and runs the program identified by the URI
  4. The child runs and generates dynamic content
  5. Server forwards the content to the client

  - Request arguments appended to the URI. List starts with `?`, separated by `&`
  - Arguments are passed to the child in `QUERY_STRING`
  - Server uses `dup2` to redirect child's `stdout` to its connected socket

- **CGI**(Common Gateway Interface)
  - Original standard for generating dynamic content (now replaced)
  - Defines a simple standard for transferring information between the client, server, the child process

Concurrent Programming

- Hard!
  - Human mind is sequential, misleading notion of time
  - Considering all possibilities of **interleaving** is impossible
  - **Races**: Outcome depends on arbitrary scheduling decisions
  - **Deadlock**: Improper resource allocation preventing progress, stuck waiting for an event that will never happen
  - Livelock, starvation, fairness etc.
- **Concurrent Servers**
  - **Process**-based: Automatic interleaving by kernel, private address space for each flow
  - **Event**-based: Manual interleaving, shared address space, *I/O multiplexing*
  - **Thread**-based: Automatic interleaving by kernel, shared address space (Mixup)
- **Process Based Server**
  - Separate process for each client (`fork`)
  - Must reap all zombie children
  - `accept` process
    1. Server blocks in `accept`, waits for connection request on `listenfd`
    2. Client makes connection request by `connect`
    3. Server returns `connfd` from `accept`, *Forks child to handle client*
    4. Connection between `clientfd` and `connfd` is established
  - *No shared states between clients*
  - Both parent & child have copies of `listenfd` and `connfd`: parent should close `connfd`, child should close `listenfd` (considering `refcnt`)
  - Pros
    * Clean sharing model - file tables (o), descriptors/global var.(x)
    * Simple and straightforward
  - Cons
    * Additional overhead for process control

    - Hard to share data between processes (IPC)

- **Event Based Server**
  - Maintains a set of active connections by an array of `connfds`
  - Repeats:
    - `select` which descriptors have pending inputs
    - If `listenfd` has input, `accept` the connection
    - Add new `connfd` to array
    - Service all `connfds` with pending inputs
  - Pros
    - One logical control flow and shared address space
    - Can single step with debugger
    - No process or thread control overhead
    - Gives programmers more control over the behavior
  - Cons
    - Too complex
    - Hard to provide fine-grained concurrency
    - Cannot take advantage of multi-core (single control)
- **Thread**: Logical flow that runs in the context of a process
  - **Thread context**: Registers, Condition Codes, Stack Pointer, Program Counter, Thread ID, own stack (for local var)
  - Threads share same code, data, and kernel context (VA space)
  - Threads $\approx$ pools of concurrent flow that access the same data[4]
  - Concurrent if flows overlap in time
- **Threads vs. Processes**
  - Similarities
    - Own logical control flow
    - Can run concurrently with others
    - Context switching

[4] Processes form a tree hierarchy, where threads do not

- **Differences**
  - Threads share all code and data (except local stacks)
  - Threads are less expensive than processes (they have less context)

- **Posix Threads** (`pthreads`) **Interface**
  - Standard interface that manipulate threads from C programs
  - Creating Threads
    - `int pthread_create(pthread_t *tid, NULL, func *f, void *arg);`
    - `tid`: contains id of created thread
    - `f`: thread routine
    - `arg`: arguments for `f`
  - Terminating Threads
    - `void pthread_exit(void *thread_return);`
    - `int pthread_cancel(pthread_t tid);`
    - Terminates the thread with `tid`
  - Reaping Threads
    - `int pthread_join(pthread_t tid, void **thread_return);`
    - Blocks until thread `tid` terminates, and reaps terminated thread
    - Can only wait for a specific thread
  - Detaching Threads
    - Joinable thread: Can be reaped and killed by other threads, memory is not freed until reaped.
    - Detached thread: Cannot be reaped by other threads, memory is freed automatically on termination
    - `int pthread_detach(pthread_t tid);`

- **Thread Based Server**
  - Run only detached threads: reaped automatically
  - Free `vargp`, `close(connfd)` necessary
  - Each client handled by each peer thread
  - Careful to avoid unintended sharing (`malloc`)
  - Functions in the thread routine must be thread-safe

- Pros
  - Easy to share data between threads (perhaps too easy)
  - Efficient than processes (cheaper context switch)
- Cons
  - Unintended sharing
  - Difficult to debug

Synchronization

- **Threads Memory Model**
  - Variable shared $\iff$ Multiple threads reference the variable
  - Multiple threads run within the context of a single process
  - Threads have its own thread context (TID, SP, PC, CC, REG)
  - Share the remaining process context
- **Variable Instances in Memory**
  - Global Variables: Exactly one instance
  - Local Variables: Each thread stack has one instance each
  - Local *Static* Variables: Exactly one instance[5]
- **Concurrent Execution & Process Graphs**
  - Interleaving of any order possible; May cause errors
  - **Process Graph** depicts the discrete *execution state space* of concurrent threads
  - Axis: sequential order of instructions in a thread
  - Each point: Possible *execution state*
  - Trajectory: is a sequence of legal state *transitions* of possible concurrent execution (one set of interleaving)
  - **Critical Section** (w.r.t a shared var): load $\sim$ store instruction
  - Instructions in critical section should not be interleaved
  - **Unsafe Region**: Intersection of critical sections
  - Trajectory is *safe* $\iff$ does not pass unsafe region
  - Enforce **mutual exclusion** to **synchronize** the execution of threads so that they can never have an unsafe trajectory
- **Semaphores**: Non-negative global integer synchronization variable
  - Mainpulated by `P`, `V` operations
  - `P(s)` (= Lock)
    * If $s \neq 0$, `s--` and return (happens atomically)

[5] It's similar to global variables, just that its scope is limited to the function

    - If $s = 0$, **suspend** until $s \neq 0$, and the thread is restarted by a `V` operation
    - After restart, `P` decrements `s` and returns control to caller
  - `V(s)` (= Unlock)
    - `s++` and return
    - If any threads blocked in `P` are waiting, restart exactly one of those threads,[6] which enables `P` to decrement `s`.
  - **Semaphore Invariant**: $s \geq 0$

- **Semaphores for Synchronization**
  - Associate a unique semaphore **mutex** (initially 1) with each shared var
  - Surround corresponding critical sections with `P`, `V` operations
  - *Binary Semaphores*: Value is 0 or 1
  - *Mutex*: Binary semaphores for **mut**ual **ex**clusion
  - *Counting Semaphore*: Counter for set of available resources
  - Synchronization makes programs run slower
  - The semaphore invariant surrounds critical sections, which is the *forbidden region*
  - Semaphore is < 0 in the forbidden region, therefore cannot be passed by any trajectory

- **Semaphores to Coordinate Access to Shared Resources**
  - Semaphore operation to notify another thread that some condition has become true
  - Use counting semaphores to keep track of resource state
  - Mutex for protecting access to the resource

- **Producer-Consumer Problem**
  - They share a bounded buffer with n slots
  - Producer produces new items, inserts them to the buffer, notify consumer
  - Consumer consumes items, removes them from the buffer, notify producer
  - `sbuf` (shared buffer) package
  - `slots`: counts available slots in the buffer
  - *items*: counts available items in the buffer

[6] You don't know which will be restarted...

- **Reader-Writers Problem**
  - Reader threads only read object
  - Writer threads modify the object $\rightarrow$ Must have exclusive access
  - Unlimited readers can access the object
  - *First readers-writers problem* (Reader Favoring)
    * No reader should be kept waiting if writer doesn't have access
    * Reader has priority over writers
    * Starvation for writers may happen
  - *Second readers-writers problem* (Writer Favoring)
    * Once a writer is ready to write, performs write ASAP
    * Readers that arrive after a writer must wait, even if the writer is also waiting
    * Starvation for readers may happen
- **Prethreaded Concurrent Server**
  - Creating/reaping thread is expensive! Maintain a set of worker threads!
  - Server consists of main thread and a set of worker threads
  - Main thread repeatedly accepts connection from clients and places `connfd` in a bounded buffer
  - Each worker thread removes `connfd` from the buffer, services client and waits for the next descriptor
- **Thread Safety**
  - Functions called in a thread routine must be **thread safe**
  - Thread Safe $\iff$ Always produces correct results when called repeatedly from multiple concurrent threads
  - Classes of unsafe functions
    1. Functions that do not protect *shared variables*
       * Use `P`, `V` operations to synchronize
    2. Functions that keep *states across multiple invocations*
       * Modify function to be *re-entrant*
    3. Functions that return a *pointer to a static variable*

        * Rewrite function so caller passes address of variable to store result
        * Lock and copy: Lock and copy to a another private memory location to store the result (write a wrapper function)
    4. Functions that call other unsafe functions
        * Just don't call them

- **Reentrancy**
  - Function is **reentrant** $\iff$ Does not access shared variables when called by multiple threads
  - Requires no synchronization process (which is expensive)
- **Race Conditions**
  - Race when the correctness of program depends on on thread reaching point x before another thread reaches y
  - Happens usually when programmer assumes some particular trajectory
  - Avoid unintended sharing to prevent races
- **Deadlocks**
  - Deadlock $\iff$ Waiting for a condition that will never be true
  - `P` operation is a potential problem because it blocks
  - Trajectory entering deadlock region will reach deadlock state
  - Often non-deterministic
  - Fix: Acquire shared resources in the same order