"Processes" items

What is process?
A process refers to an executing instance of a program. It is a unit of work within an operating system that consists of a program code, current activity, and associated resources such as memory, file descriptors, and CPU registers.

What is the difference between process and program?
A program is a set of instructions stored in a file, while a process is an instance of a program that is being executed. In simpler terms, a program is a passive entity, whereas a process is an active entity with its own state, memory, and resources.

List five process states (use the naming from our textbook and class, please)
The five process states are:

1. New: The process is being created or initialized.
2. Ready: The process is waiting to be assigned to a CPU.
3. Running: The process is currently being executed on a CPU.
4. Waiting: The process is waiting for an event or resource.
5. Terminated: The process has finished execution or has been terminated.

What is the ready queue? How many ready queues do you expect in the system?
The ready queue is a queue that holds the processes in the "ready" state, waiting to be assigned to a CPU for execution. The number of ready queues in the system typically depends on the specific scheduling algorithm being used. Commonly, there is one ready queue per CPU or one global ready queue for all CPUs in a multi-core system.

What is the I/O queue? How many I/O queues do you expect in the system?
The I/O queue is a queue that holds the processes that are waiting for I/O operations to complete. It represents the processes that have issued I/O requests and are currently blocked until the I/O operation finishes. The number of I/O queues in the system depends on the design of the operating system. There can be multiple I/O queues, with each queue associated with a specific I/O device or type.

What happens to a process when it requests an I/O operation (like file reading) in the classic "blocking" scenario?
When a process requests an I/O operation in the classic "blocking" scenario, it enters the "waiting" state. The process is blocked and cannot proceed with its execution until the requested I/O operation completes. The operating system typically removes the process from the CPU and places it in the I/O queue associated with the requested device.

Where does the process go once the I/O request is complete?
Once the I/O request is complete, the process moves from the "waiting" state to the "ready" state. It is then placed back in the ready queue, waiting to be scheduled and assigned to a CPU for execution.

Understand what happens to processes in reality when process states change (don't be fooled by process "movements" in typical visualizations).
In reality, when a process changes its state, it does not physically move between different areas of the system. The movement of processes in visualizations is a conceptual representation to understand their state transitions. In reality, the process's data and control structures are updated to reflect the new state, and the operating system manages the scheduling and allocation of resources accordingly.

Process Control Block (PCB). What typical components can you see there?
In the Process Control Block (PCB), you can typically find the following components:

1. Process state: Indicates the current state of the process (e.g., running, waiting, ready).
2. Program counter: Holds the address of the next instruction to be executed.
3. CPU registers: Stores the values of CPU registers associated with the process.
4. Memory management information: Includes details about the process's memory allocation, such as base and limit registers.
5. Process identification: Contains unique identifiers for the process, such as process ID (PID).
6. Scheduling information: Contains data related to process scheduling, such as priority or scheduling queue pointers.
7. I/O status information: Stores the state of I/O operations associated with the process, such as open files or pending I/O requests.
8. Accounting information: Keeps track of resource usage and statistics for the process, such as CPU time used or amount of memory allocated.

Context switch. Why does context switching have a small performance overhead associated with it? How does that performance overhead affect system calls?
Context switching refers to the process of saving the current execution context of a process and restoring the context of another process. It has a small performance overhead due to several reasons:

1. Saving and restoring CPU registers: During a context switch, the operating system needs to save the values of CPU registers for the currently running process and restore the registers for the incoming process. This involves memory accesses and register transfers, which incur some overhead.
2. Memory management: Context switches may involve updating memory mappings and memory protection mechanisms to ensure the incoming process has access to its required memory. This operation adds to the overhead.
3. Cache flushing: When switching between processes, the contents of CPU caches may need to be flushed or invalidated, as the new process may have different memory access patterns. This cache management introduces additional overhead.

The performance overhead of context switching affects system calls because system calls require a transition from user mode to kernel mode, which involves a context switch. When a process invokes a system call, it triggers a switch from user space to kernel space, resulting in the overhead associated with saving and restoring the execution context.

What is POSIX?
POSIX (Portable Operating System Interface) is a set of standards that define the application programming interface (API), as well as the command line interface (CLI), for compatibility between different operating systems. It provides a standardized interface for developers to write portable software that can run on various POSIX-compliant operating systems, such as Unix and Linux. POSIX specifies functionalities related to file management, process management, interprocess communication, and more.

What is fork()?
In the context of operating systems, the fork() system call is used to create a new process by duplicating the existing process. The new process, known as the child process, is an exact copy of the original process, known as the parent process, except for a few values that are changed to reflect the different process IDs. After forking, both the parent and child processes continue execution from the point of the fork() call but with different process IDs.

What is exec()?
The exec() system call is used to replace the current process's memory space with a new program. It loads a new executable file into the current process's address space and starts its execution from the beginning. The exec() call is commonly followed by one of the letter variants, such as execve(), execvp(), or execl(), which provide different ways of specifying the program to be executed and its arguments.

What is wait()?
The wait() system call is used by a parent process to suspend its execution and wait for the termination of its child process. When a parent process calls wait(), it allows the operating system to reclaim the resources associated with the child process and obtain information about its termination status. The parent process resumes execution once the child process has terminated.

Layout of a process in memory: text section, data section, heap, and stack.
A process in memory typically has the following sections:

1. Text section (also known as code section): It contains the executable code of the program, including the instructions and constants. This section is usually read-only and shared among multiple instances of the same program.
2. Data section: It includes initialized global and static variables used by the program. This section is typically read-write and includes both initialized and uninitialized data.
3. Heap: The heap is a dynamically allocated memory region used for dynamic memory allocation. It grows and shrinks as needed during program execution and is managed by functions like malloc() and free(). The heap is typically used for data structures that require flexible memory allocation, such as linked lists or dynamically sized arrays.
4. Stack: The stack is a region of memory used for storing local variables, function call information, and return addresses. Each function call creates a stack frame that contains the necessary information for that particular function's execution. The stack operates in a

last-in-first-out (LIFO) manner, meaning the most recently pushed data is the first to be popped.

What are the properties of the heap (it is fragmented, full of holes)? How does it affect typical dynamic memory allocation, like using the operator new?

The heap has the following properties:

1. Fragmentation: As memory is allocated and deallocated dynamically, the heap can become fragmented. Fragmentation refers to the division of the available memory into small, non-contiguous blocks, making it challenging to find a large enough contiguous block for allocation. Fragmentation can be either external (free memory scattered throughout the heap) or internal (free memory blocks surrounded by allocated blocks).
2. Full of holes: The heap can contain free memory blocks interspersed with allocated blocks, creating holes or gaps. These holes are spaces between allocated blocks that are not being used.

These properties of the heap can affect dynamic memory allocation, such as using the operator new, in the following ways:

1. Fragmentation may lead to inefficient memory utilization: As the heap becomes fragmented, it becomes harder to find contiguous blocks of memory for allocation requests. This can result in wasted memory space due to small gaps between allocated blocks that are too small to be utilized effectively.
2. Increased allocation time: Finding a suitable block of memory in a fragmented heap can be time-consuming. The allocator needs to search for available memory blocks, potentially traversing a fragmented data structure. This search process can increase the allocation time compared to a non-fragmented heap.
3. Increased memory management overhead: The heap management algorithms need to handle fragmentation and maintain data structures to track free blocks and allocated blocks. This overhead can impact memory allocation and deallocation performance.

"IPC" items

What is IPC?
IPC stands for Interprocess Communication. It refers to the mechanisms and techniques used by processes or threads running on a computer system to communicate and synchronize with each other. IPC allows processes to share data, exchange messages, and coordinate their activities.

What are two general classes of IPC (shared memory and message passing)?
The two general classes of IPC are:

1. Shared Memory: Shared memory IPC involves processes or threads accessing a common memory region that is shared among them. By sharing this memory, processes can communicate by reading and writing data to the shared memory region.
2. Message Passing: Message passing IPC involves processes or threads sending and receiving messages to communicate with each other. Messages can be sent directly between processes, typically through system calls or communication libraries provided by the operating system.

What is the idea of shared memory? What are the advantages and disadvantages of shared memory?
The idea of shared memory is to allow multiple processes to access the same memory region. Instead of copying data between processes, they can directly read from and write to the shared memory.

Advantages of shared memory include:

1. Fast Communication: Since processes can directly access the shared memory, communication between them can be faster compared to other IPC methods.
2. Efficient Data Sharing: Shared memory enables efficient sharing of large amounts of data between processes without the need for data duplication.

Disadvantages of shared memory include:

1. Synchronization: As multiple processes can access shared memory simultaneously, synchronization mechanisms such as locks or semaphores are required to prevent conflicts and ensure data integrity.
2. Complexity: Shared memory communication requires careful management to avoid issues like race conditions and deadlocks, which can be complex to handle correctly.

What is the idea of message passing? What are the advantages and disadvantages of message passing?
The idea of message passing is to allow processes to communicate by sending and receiving messages. Processes can exchange information and synchronize their activities through message passing.

Advantages of message passing include:

1. Simplicity: Message passing provides a simpler and more straightforward communication model compared to shared memory. Processes only need to send and receive messages without concerns about shared data or synchronization.
2. Isolation: Message passing ensures that processes remain isolated from each other. They can communicate and share information in a controlled manner, reducing the risk of unintended interference.

Disadvantages of message passing include:

1. Overhead: Message passing may introduce overhead due to the need to package and unpack messages, as well as the associated system calls or library functions for message passing operations.
2. Limited Data Size: Message passing is typically suited for exchanging smaller amounts of data. When large data sets need to be shared, message passing might become less efficient compared to shared memory.

Can shared memory, as a kind of IPC communication, be used for two processes running on two different computers? Why do you think so?

No, shared memory IPC cannot be used for two processes running on two different computers directly. Shared memory relies on processes having access to the same physical memory space. In a distributed computing environment where processes are running on different computers, they have separate memory spaces that are not directly accessible by other machines.

To enable communication between processes on different computers, other IPC methods such as message passing or network communication protocols need to be used. These methods allow processes to exchange messages or data over a network, facilitating communication between separate systems.

What is a network?

A network refers to a collection of computers and other devices interconnected to facilitate the exchange of data and resources. Networks enable communication between computers, allowing them to share information, access shared resources such as printers or servers, and collaborate effectively.

What is an IP address? How does a classic IP address (IPv4) look like?

An IP address is a numerical label assigned to each device connected to a network that uses the Internet Protocol for communication. It serves as a unique identifier for devices within a network. IP addresses allow devices to send and receive data over the internet or local networks.

The classic IP address format is known as IPv4 (Internet Protocol version 4). An IPv4 address consists of four sets of numbers separated by periods, where each set can range from 0 to 255. For example, an IPv4 address might look like: 192.168.0.1.

Why is an IP address alone not enough for networking communications?
While an IP address is essential for identifying and addressing devices on a network, it is not sufficient for networking communications on its own. Networking communications often require additional information and protocols to establish connections, manage data transmission, and ensure reliable delivery.
IP addresses primarily handle the routing of data packets across networks, determining the source and destination of the data. However, to establish communication between devices, protocols such as TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) are used, which provide additional functionality like error checking, flow control, and connection management.

What is a networking port?
In networking, a port is a numerical identifier used to distinguish between different processes or services running on a single device. It allows multiple applications or services to utilize network resources simultaneously. Ports are associated with specific protocols and are used to direct incoming network traffic to the appropriate application or service.

What are well-known ports?
Well-known ports refer to a range of port numbers that are standardized and commonly used for specific network services. These ports are standardized by the Internet Assigned Numbers Authority (IANA) to ensure consistent usage across different systems and networks.
Well-known ports have numbers ranging from 0 to 1023, and they are associated with widely used protocols and services. For example, port 80 is commonly used for HTTP (Hypertext Transfer Protocol), port 21 for FTP (File Transfer Protocol), and port 22 for SSH (Secure Shell). Well-known ports are reserved for specific services to allow easier identification and configuration of network applications.

What is a networking socket?
A networking socket is an endpoint for communication between two devices over a network. It is a combination of an IP address and a port number that allows applications to establish network connections and exchange data. Sockets provide an interface for sending and receiving data between applications running on different devices.

What is a server (like in "web server" or "email server")?
In the context of networking, a server refers to a computer or a software application that provides services or resources to other devices or applications on a network. Servers are designed to listen for incoming requests from client devices and respond to those requests by providing the requested service or resource. Examples of servers include web servers, email servers, file servers, and database servers.

What is a client?
A client is a device or a software application that accesses services or resources provided by a server. Clients initiate requests to servers and receive responses containing the requested information. Clients can be computers, smartphones, tablets, or any other device that can connect to a network and interact with server applications.

What is a communication protocol?
A communication protocol is a set of rules and standards that define the format, structure, and sequence of messages exchanged between devices in a network. Protocols ensure that devices can communicate effectively and understand each other's messages. They define how data is transmitted, how errors are handled, how devices establish connections, and how they terminate connections. Common examples of communication protocols include TCP/IP, HTTP, FTP, and SMTP.

What is a URL?
URL stands for Uniform Resource Locator. It is a string of characters that provides the address or location of a resource on the internet. URLs are commonly used to specify the web addresses of websites, web pages, files, or any other resource accessible over the internet. A typical URL consists of a protocol identifier (such as "http://" or "https://"), followed by the domain name or IP address of the server, and optional additional path or query parameters.

What is DNS?
DNS stands for Domain Name System. It is a decentralized hierarchical naming system that translates human-readable domain names (such as example.com) into IP addresses used by network devices to locate resources on the internet. DNS servers maintain a distributed database that maps domain names to IP addresses, allowing users to access websites, send emails, and perform other network activities using domain names instead of numerical IP addresses.

What is RPC?
RPC stands for Remote Procedure Call. It is a communication protocol that allows a program or process to execute a procedure or function on a remote computer or server as if it were running locally. RPC enables distributed applications to interact and share resources transparently across a network. The calling program sends a request to the remote server, which executes the requested procedure and sends back the results. RPC abstracts the complexities of network communication, allowing remote procedure calls to be made in a manner similar to local function calls.

"Threads" items

What is a thread of execution (a.k.a. just thread)?
A thread of execution, commonly referred to as a thread, is a sequence of instructions that can be scheduled and executed independently within a process. Threads are the smallest units of execution within a program and allow for concurrent execution of multiple tasks within a single process. Threads share the same memory space and resources of the process they belong to, enabling efficient communication and coordination.

Why might we want to create new threads of execution instead of new processes?
Creating new threads of execution instead of new processes can be beneficial in various scenarios:

1. Resource Efficiency: Threads within a process share the same memory space, file descriptors, and other resources. Creating a new thread is generally faster and consumes less system resources compared to creating a new process, which requires duplicating the entire process's memory space.
2. Communication and Coordination: Threads within the same process can easily communicate and share data through shared memory. This allows for efficient coordination and exchange of information between threads, simplifying concurrent programming.
3. Responsiveness: Threads can be used to perform concurrent tasks within a process, allowing for better responsiveness. For example, a user interface thread can handle user input while another thread performs background tasks, ensuring that the application remains responsive to user interactions.

Why might we want to create new processes instead of new threads?
Creating new processes instead of new threads can be advantageous in certain situations:

1. Isolation: Processes provide a higher level of isolation compared to threads. Each process has its own memory space, resources, and protection mechanisms. This isolation ensures that a failure or issue in one process does not affect other processes.
2. Security: Processes provide a higher level of security as they are isolated from each other. A vulnerability or malicious activity in one process is less likely to compromise the security of other processes.
3. Parallelism on Multi-core Systems: Processes can be scheduled to run on different processor cores, enabling true parallelism and leveraging the full power of multi-core systems. In contrast, threads within a process may have limitations in achieving parallel execution on multiple cores due to factors like shared resources and synchronization.

What do threads belonging to the same process share? What don't they share?

Threads belonging to the same process share the following:

1. Memory Space: Threads within a process share the same memory space, allowing them to directly access and modify the same variables and data structures.
2. File Descriptors: Threads within a process share the same file descriptors. This means that multiple threads can read from or write to the same files or network connections.
3. Process Context: Threads within a process share the same process context, including things like environment variables, signal handlers, and process ID.

Threads belonging to the same process do not share:

1. Register State: Each thread has its own register state, including the program counter, stack pointer, and register values specific to the thread's execution.
2. Stack Space: Each thread has its own stack space to store local variables and function call information.

User threads vs. kernel threads:

User threads are managed entirely by user-level libraries or the application itself, without direct intervention from the operating system kernel. User-level threads provide flexibility and control to the application but can suffer from limitations such as being blocked by a single thread blocking operation.

Kernel threads, on the other hand, are managed by the operating system kernel. Each kernel thread is represented as a separate entity in the kernel and can be scheduled independently. Kernel threads are generally more robust and can handle blocking operations efficiently. However, the creation and management of kernel threads involve overhead due to kernel involvement.

In some systems, user threads are implemented on top of kernel threads, allowing the benefits of both approaches. The user-level thread library manages the user threads, while the kernel handles the scheduling and execution

"Synchronization" items

What are synchronization problems?
Synchronization problems refer to programming issues that arise when multiple threads or programs attempt to access shared resources or exchange data concurrently. These problems occur due to the non-deterministic nature of thread execution, leading to unexpected and undesirable outcomes.

What synchronization hazards do you know?
Two common synchronization hazards are:

1. Race condition: A race condition occurs when the behavior of a program or system depends on the relative timing or interleaving of events. It arises when multiple threads or processes access and manipulate shared resources concurrently, resulting in unpredictable outcomes. Race conditions can lead to data corruption, incorrect results, or program crashes.
2. Deadlock: Deadlock occurs when two or more threads or processes are unable to proceed because each is waiting for a resource that the other holds. In a deadlock situation, none of the threads or processes can make progress, causing the system to become unresponsive. Deadlocks can arise when there is a circular dependency among resources or when locking mechanisms are not properly managed.

What is a shared resource?
A shared resource refers to a resource or data entity that can be accessed and utilized by multiple threads, processes, or programs concurrently. It can be anything from variables, files, database connections, network sockets, to hardware devices. When multiple entities attempt to access a shared resource simultaneously, proper synchronization mechanisms need to be implemented to ensure correct and consistent access to the resource. Without appropriate synchronization, conflicts and synchronization hazards, such as race conditions and deadlocks, can occur.

What is a race condition (a.k.a. data race)?
A race condition, also known as a data race, is a situation that occurs in multithreaded or concurrent programming when the behavior of a program depends on the relative timing or interleaving of operations on shared data. It arises when multiple threads or processes access and manipulate shared data concurrently without proper synchronization, leading to unexpected and incorrect results.

Explanation of how a race condition can happen at the assembly/CPU level:
At the assembly/CPU level, a race condition can occur due to the interleaved execution of instructions from multiple threads or processes. CPUs execute instructions in parallel or in an overlapping manner to improve performance. When multiple threads access shared data concurrently, the following sequence of events can lead to a race condition:

1. Load: The CPU loads the value of a shared data variable into a register.
2. Modify: The CPU performs calculations or manipulations on the value stored in the register.
3. Store: The CPU writes the modified value back to the shared data variable.

If multiple threads execute these instructions concurrently, their order of execution can lead to inconsistent or incorrect results. For example, if two threads simultaneously load the same value, modify it independently, and then store the modified value back, the final value of the shared data variable depends on the timing of the threads' operations, leading to non-deterministic outcomes.

Can a race condition happen if only one party modifies the data and others are just reading? Can a race condition happen if everybody is just reading?
A race condition typically occurs when at least one thread is performing a write (modification) operation on shared data. If multiple threads are only reading (performing read operations) and no write operations are involved, a race condition is unlikely to occur. Reading data concurrently does not lead to inconsistent or incorrect results as long as the data remains unchanged. However, it is worth noting that race conditions can still occur if there is a mix of read and write operations. For example, if one thread is writing to a shared data variable while other threads are reading from it simultaneously, race conditions can arise if proper synchronization mechanisms are not in place.

What is the critical section problem?
The critical section problem refers to the challenge of coordinating access to shared resources or critical sections of code in a multithreaded or concurrent environment. A critical section is a section of code or a region where shared data is accessed and modified. The goal of the critical section problem is to ensure that only one thread or process executes its critical section at a time, preventing concurrent access and potential race conditions.

What are the three properties of a good solution to the critical section problem?
A good solution to the critical section problem should satisfy the following three properties:

1. Mutual Exclusion: Only one thread or process can be executing its critical section at any given time. This property ensures that concurrent access to shared resources does not occur, preventing race conditions.
2. Progress: If no thread is executing its critical section and there are threads that want to enter their critical sections, one of those threads should be able to enter without being indefinitely delayed. This property prevents deadlock situations where no progress is being made.
3. Bounded Waiting: There exists a bound on the number of times other threads can enter their critical sections after a thread requests access to its critical section but before that request is granted. This property ensures that no thread is starved or unfairly delayed in accessing its critical section indefinitely.

The idea of locking shared resources:
Locking shared resources involves the use of synchronization mechanisms, such as locks or mutexes, to coordinate access to critical sections of code or shared data. When a thread wants to access a shared resource, it first acquires the lock associated with that resource. Acquiring the lock grants the thread exclusive access to the resource, preventing other threads from accessing it concurrently. Once the thread is done with the resource, it releases the lock, allowing other threads to acquire it.

What does it mean if a function is hardware implemented?
If a function is hardware implemented, it means that the function's implementation is built directly into the hardware circuitry of the processor or computer system. Instead of being implemented as software code executed by the CPU, the function is executed through dedicated hardware circuits. This allows for faster and more efficient execution of the function, as it can be performed with a single special CPU instruction rather than a sequence of general CPU instructions.

What is the test_and_set instruction and what does it do?
The test_and_set instruction is a low-level atomic instruction provided by some computer architectures. It is used to implement synchronization mechanisms for accessing shared resources. The test_and_set instruction atomically performs two operations: it reads the value of a memory location and sets that memory location to a new value. Importantly, the test_and_set operation is performed as an indivisible, uninterruptible operation. It ensures that no other thread or process can access or modify the memory location between the read and write operations.

Simple solution of the critical section problem using the test_and_set instruction:
In this solution, a lock variable is used to control access to the critical section. The lock variable is initially set to false (unlocked). When a thread wants to enter its critical section, it executes the test_and_set instruction on the lock variable. If the lock variable is already set (true), indicating that another thread is inside the critical section, the requesting thread spins in a loop until the

lock becomes available (false). Once the lock is obtained, the thread enters its critical section, performs the necessary operations, and then releases the lock for other threads to acquire.

What is suboptimal in this solution?
The simple solution using the test_and_set instruction is suboptimal because it does not provide the "bounded waiting" property. This means that a thread requesting access to its critical section may experience indefinite delays if other threads continually acquire and release the lock before it gets a chance. This situation can lead to starvation, where a thread is unfairly denied access to its critical section for an extended period. A good solution to the critical section problem should ensure that every thread eventually gets an opportunity to enter its critical section, preventing starvation.

Solution of the critical section problem using the test_and_set instruction with the "bounded waiting" property:
To achieve the "bounded waiting" property in the critical section problem using the test_and_set instruction, we can use a modification known as the "test_and_set with waiting" solution. This solution introduces the concept of a turn variable or flag, which indicates whose turn it is to access the critical section. Each thread or process that wants to enter its critical section checks the turn variable before attempting to acquire the lock using the test_and_set instruction. If the turn indicates that it is not their turn, the thread enters a waiting state and repeatedly checks the turn variable until it becomes their turn.
By implementing this waiting mechanism, each thread is guaranteed to eventually access its critical section, ensuring that there is a bound on the waiting time and preventing starvation.

What does busy waiting mean? Why is it inefficient? When is it still good?
Busy waiting refers to a situation where a thread or process repeatedly checks for the availability of a resource or condition in a loop without yielding or performing any meaningful work. In the context of synchronization, it occurs when a thread continuously checks the lock or condition variable to see if it has become available.
Busy waiting is inefficient because it wastes CPU cycles and system resources. While a thread is in a busy waiting state, it consumes CPU time without performing useful work. This leads to reduced overall system performance and inefficiency.
Busy waiting is still considered acceptable or good in certain scenarios when the expected wait times are short, and the cost of continuously checking the condition is lower than the overhead of performing a context switch to another process. In such cases, busy waiting can be more efficient than context switching and waiting for I/O completion.

What is a mutex? What does a process do if it needs a mutex that is busy?
A mutex (short for mutual exclusion) is a synchronization primitive used to protect shared resources or critical sections of code in concurrent programming. It provides mutual exclusion, allowing only one thread or process to acquire the mutex at a time. When a process needs a mutex that is currently busy (already acquired by another process), it pauses or waits for the mutex to become available again before proceeding further. The process typically enters a

waiting state and is blocked until the mutex is released by the thread or process currently holding it. Once the mutex becomes free, the waiting process is notified and can proceed to acquire the mutex and enter its critical section.

```
#include <iostream>
#include <map>
#include <string>
#include <chrono>
#include <thread>
#include <mutex>

std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;

void save_page(const std::string &url)
{
      // simulate a long page fetch
      std::this_thread::sleep_for(std::chrono::seconds(2));
      std::string result = "fake content";

      std::lock_guard<std::mutex> guard(g_pages_mutex);
      g_pages[url] = result;
}

int main()
{
      std::thread t1(save_page, "http://foo");
      std::thread t2(save_page, "http://bar");
      t1.join();
      t2.join();

      // safe to access g_pages without lock now, as the threads are joined
      for (const auto &pair : g_pages) {
      std::cout << pair.first << " => " << pair.second << '\n';
      }
}
=======================================================

#include <iostream>
#include <thread>
#include <mutex>
```

```
std::mutex m;//you can use std::lock_guard if you want to be exception safe
int i = 0;
void makeACallFromPhoneBooth()
{

	//The other men wait outside
	m.lock();// one man gets a hold of the phone booth door and locks it.
	//man happily talks to his wife from now....
	std::cout << i << " Hello Wife" << std::endl;
	i++;//no other thread can access variable i until m.unlock() is called
	//...until now, with no interruption from other men
	m.unlock();//man unlocks the phone booth door
}

int main()
{
	//This is the main crowd of people uninterested in making a phone call

	//man1 leaves the crowd to go to the phone booth
	std::thread man1(makeACallFromPhoneBooth);
	//Although man2 appears to start second, there's a good chance he might
	//reach the phone booth before man1
	std::thread man2(makeACallFromPhoneBooth);
	//And hey, man3 also joined the race to the booth
	std::thread man3(makeACallFromPhoneBooth);

	man1.join();//man1 finished his phone call and joins the crowd
	man2.join();//man2 finished his phone call and joins the crowd
	man3.join();//man3 finished his phone call and joins the crowd
	return 0;

}
```

Producer-Consumer problem (Bounded Buffer problem) and its solution:
The Producer-Consumer problem involves two types of threads: producers, which produce items, and consumers, which consume the items. They share a common buffer or queue of fixed size. The challenge is to ensure that producers do not produce items when the buffer is full and consumers do not consume items when the buffer is empty.

One common solution to this problem is to use a mutex and condition variables. The mutex is used to protect the critical sections where producers and consumers access the buffer. The condition variables are used to signal and wait for changes in the state of the buffer (e.g., when it becomes non-empty or non-full). Producers acquire the mutex, check if the buffer is full, and if so, wait on the condition variable. When a consumer consumes an item, it signals the condition variable to notify waiting producers. Similarly, consumers acquire the mutex, check if the buffer is empty, and if so, wait on the condition variable. When a producer produces an item, it signals the condition variable to notify waiting consumers.

Readers-Writers problem and its solution:
The Readers-Writers problem involves multiple threads where some threads act as readers that only read a shared resource, and others act as writers that both read and modify the resource. The challenge is to ensure that concurrent reads do not interfere with each other, but only one writer can access the resource at a time, preventing data corruption.
One solution to this problem is to use a combination of a mutex and a semaphore. The mutex is used to provide mutual exclusion for the writers, allowing only one writer to access the resource at a time. The semaphore is used to track the number of readers currently accessing the resource. Readers acquire the mutex before incrementing the reader count and releasing the mutex, allowing multiple readers to access the resource concurrently. Writers acquire the mutex and wait on the semaphore until all readers have finished accessing the resource. Once the last reader has finished, the writer can access the resource, update it, and release the mutex.

Dining Philosophers problem
The Dining Philosophers problem is a classic synchronization problem that involves a group of philosophers sitting at a round table with a bowl of rice and a chopstick between each pair of philosophers. Philosophers alternate between thinking and eating. To eat, a philosopher must acquire both the left and right chopsticks. The challenge is to design a solution that prevents deadlocks, where each philosopher holds one chopstick and waits indefinitely for the other.
In this problem, philosophers represent concurrent threads or processes, and chopsticks represent shared resources that they need to acquire to perform their task (eating). The challenge is to coordinate the use of these shared resources to avoid deadlocks.

Deadlock
Deadlock refers to a situation where two or more processes or threads are unable to proceed because each is waiting for the other to release a resource or terminate. It is a state of a system where progress halts because the required resources are unavailable, and none of the waiting processes can continue execution.

Three approaches to solving the Dining Philosophers problem are:

1. Resource Hierarchy: Assign a strict order or hierarchy to the chopsticks and require the philosophers to acquire them in that order. For example, each philosopher always picks

up the left chopstick first and then the right chopstick. This prevents the possibility of circular waits and eliminates the deadlock condition.

2. Arbitrator/Manager: Introduce a central entity or manager that controls the access to the chopsticks. The manager ensures that only a certain number of philosophers (less than or equal to the number of chopsticks) can be eating simultaneously. Philosophers must request permission from the manager before acquiring the chopsticks. This approach avoids deadlock by limiting the number of philosophers that can potentially hold the chopsticks at the same time.
3. Chandy/Misra Solution: This solution utilizes an additional token that circulates among the philosophers. A philosopher can only eat if they possess the token. When a philosopher finishes eating, they pass the token to their left neighbor, allowing the next philosopher to eat. This approach guarantees that only one philosopher can be eating at a time, avoiding deadlock. If a philosopher is unable to acquire both chopsticks, they simply pass the token to the next philosopher, ensuring progress and preventing starvation.

"Memory management" items

Ram memory purpose
RAM (Random Access Memory) is a type of computer memory that is used to store data and program instructions that are actively being accessed by the CPU (Central Processing Unit) or other hardware components. It serves as a temporary workspace for the computer, providing fast and temporary storage for data that is needed for immediate processing.

What is a memory address?
How much memory can you address with addresses of a given size? A memory address is a unique identifier assigned to each location in the computer's memory. It is used to access and manipulate data stored in memory. The size of a memory address determines the maximum amount of memory that can be addressed. For example, with 64-bit addresses, which are commonly used in modern systems, it is possible to address up to 2^64 memory locations, which is an extremely large address space.

What is a physical address?
A physical address is the actual physical location of data in the computer's memory. It represents the specific location where data is stored in the RAM chips. The physical address is assigned by the hardware, and it is used by the memory management unit (MMU) to translate logical addresses to their corresponding physical addresses.
Why are physical addresses inconvenient for modern programs? Physical addresses are inconvenient for modern programs because they are tied to the specific physical layout and organization of memory in the hardware. It means that programs would need to be aware of the specific hardware configuration and memory layout to work correctly. This limits the portability and flexibility of programs, as they would need to be modified or recompiled to run on different hardware configurations.

What is a logical address?
A logical address, also known as a virtual address, is an address used by a program or process to reference memory. It is generated by the program and represents a location in the logical address space. The logical address space is typically larger than the physical address space and provides an abstraction layer that allows programs to access memory without being concerned about the specific physical memory locations. The translation from logical addresses to physical addresses is handled by the operating system and the MMU, allowing programs to be executed in a consistent manner regardless of the underlying hardware configuration.

What is physical address space? How many physical address spaces are there in the system?

The physical address space refers to the actual physical locations or addresses in the computer's memory. It represents the range of addresses that the hardware can directly access. The size of the physical address space is determined by the number of address lines available in the hardware architecture. For example, with a 64-bit architecture, the physical address space can address up to 2^64 memory locations.

In a system, there is typically one physical address space per physical memory, meaning that each RAM module or memory device has its own unique physical address space.

What is logical (a.k.a. virtual) address space? How many logical address spaces are there in the system?

The logical address space, also known as the virtual address space, is the range of addresses that a process or program uses to access memory. It is the address space that is visible and accessible to an individual process, independent of the physical memory organization. The logical address space is typically larger than the physical address space.

In a system, there can be multiple logical address spaces, one for each process or program running on the system. Each process has its own private logical address space, allowing it to have a separate and isolated view of memory.

What kind of address does the CPU fetch with instructions?

The CPU fetches instructions using logical addresses. When executing a program, the CPU uses the program counter (PC) to fetch the next instruction from the logical address space of the currently running process.

What kind of address does the CPU send to RAM with a load/store request?

The CPU sends physical addresses to RAM with a load/store request. Before accessing the RAM, the CPU translates the logical addresses generated by the program into their corresponding physical addresses using the memory management unit (MMU). The MMU performs the necessary address translation and maps the logical addresses to physical addresses.

What kind of address arrives at RAM with the load/store request?

The physical address arrives at RAM with the load/store request. Once the CPU has translated the logical address to its corresponding physical address, the physical address is sent to the RAM. The RAM then uses this physical address to access the specific memory location and retrieve or store the data as requested by the load/store operation.

What is MMU? What is its purpose?
MMU stands for Memory Management Unit. It is a hardware component in a computer system that is responsible for managing the translation between logical addresses and physical addresses during memory access. The main purpose of the MMU is to provide memory virtualization, allowing programs to use logical addresses that are independent of the physical memory organization.
The MMU performs address translation by utilizing memory mapping techniques and page tables to map logical addresses to their corresponding physical addresses. It ensures that each process has its own isolated address space and provides memory protection and efficient memory allocation.

What is paging?
Paging is a memory management scheme used by the operating system to manage memory in a computer system. In paging, the logical address space of a process is divided into fixed-size blocks called pages. Similarly, the physical memory is divided into fixed-size blocks called frames. The operating system uses the MMU to map the pages of a process to the available frames in physical memory.

What is a page? What is a frame?
A page is a fixed-size block of the logical address space of a process. It represents a unit of memory that can be allocated and managed by the operating system. Pages are typically of equal size and are used to divide the logical address space into manageable units.
A frame, on the other hand, is a fixed-size block of the physical memory. It represents a unit of physical memory that can store data and instructions. Frames are typically of equal size and are used to divide the physical memory into manageable units that can be allocated to pages.

How do the sizes of pages and frames relate?
The sizes of pages and frames are typically the same in a computer system. This means that a page and a frame have the same fixed size. The equal size allows for efficient mapping of logical pages to physical frames. When a page is mapped to a frame, the entire page is stored within the frame, and any remaining space within the frame is left unused.
Having the same size for pages and frames simplifies the memory management process, as it allows for direct mapping and efficient allocation of memory.

What is page offset?
The page offset, also known as the page displacement, is the lower portion of a logical address or a physical address that specifies the offset within a page or a frame. It represents the position or displacement of a specific memory location within a page or a frame.
The page offset is used by the CPU and the MMU to determine the exact memory location within a page or a frame that needs to be accessed. It is typically determined by the number of bits required to represent the size of a page or a frame. For example, if the page or frame size is 4KB (2^12 bytes), then the page offset will require 12 bits to represent the displacement within the page or frame.

What is a page table?
A page table is a data structure used by the operating system to keep track of the mapping between logical pages and physical frames in a computer system. It acts as a translation table and provides the necessary information to the MMU for address translation during memory access.
Each process has its own page table, which is created and managed by the operating system. The page table contains entries that map logical page numbers to corresponding physical frame numbers or page table entries.

How many page tables are there in a system?
In a system, there can be multiple page tables, one for each process running on the system. Each process has its own distinct page table that maintains the mapping between logical pages and physical frames specific to that process.

What is TLB? What is its purpose?
TLB stands for Translation Lookaside Buffer. It is a hardware cache used by the MMU to accelerate the translation of logical addresses to physical addresses. The TLB stores a subset of the page table entries, specifically the recently accessed or frequently used mappings.
The purpose of the TLB is to provide a faster lookup mechanism for address translation. It caches the most commonly accessed translations, reducing the need to access the full page table for every memory access.

What is TLB hit? What is TLB miss?
A TLB hit occurs when the translation for a given logical address is found in the TLB cache. It means that the required address translation is readily available in the TLB, and the MMU can quickly retrieve the corresponding physical address.
A TLB miss occurs when the translation for a logical address is not found in the TLB cache. In this case, the MMU needs to access the full page table to find the necessary translation. It takes additional time and results in a performance penalty compared to a TLB hit.

Is it necessary to have the whole program in memory to run it?
No, it is not necessary to have the entire program in memory to run it. In most modern operating systems, programs are loaded into memory in a demand-driven manner. Only the required portions of the program, such as code segments and data structures, are loaded into memory when needed.
This approach is known as demand paging or demand loading. The operating system brings in the necessary pages of the program into memory as they are accessed, and if a page is not currently in memory, a page fault occurs.

What is a page fault?
A page fault is an exception that occurs when a program attempts to access a page that is not currently in memory. It happens when the requested page is not present in the physical memory or has been swapped out to secondary storage.
When a page fault occurs, the operating system handles it by bringing the required page into memory from secondary storage (such as the hard disk) and updating the page table to reflect the new mapping. Once the page is in memory, the program can resume its execution. Page faults are a normal part of virtual memory management and allow for efficient utilization of memory resources by bringing in pages on demand.

"CPU Scheduling" items

What is CPU scheduling about?
CPU scheduling is a key aspect of operating systems that involves determining the order in which processes are allocated CPU time. It is responsible for efficiently managing the execution of multiple processes, as the CPU can only execute one process at a time. CPU scheduling aims to maximize CPU utilization, throughput, and responsiveness while minimizing waiting times and overall system latency.

What does the choice of a specific CPU scheduling algorithm depend on?
The choice of a specific CPU scheduling algorithm depends on various factors, including the characteristics of the system, the workload, and the desired performance goals. Some factors that influence the selection of a CPU scheduling algorithm include:

1. System characteristics: The number of available CPUs, the type of tasks being executed (interactive or batch), and the level of multiprogramming in the system.
2. Workload characteristics: The distribution of CPU burst times, the frequency of I/O operations, the priority levels assigned to processes, and the variability of resource requirements.
3. Performance goals: The desired metrics for evaluating the efficiency and effectiveness of the scheduler, such as response time, throughput, fairness, and resource utilization.

What is CPU burst? What is I/O burst?
CPU burst refers to the amount of time a process spends actively using the CPU, executing its instructions. It represents the period when the process is utilizing the CPU to perform computational tasks.
On the other hand, I/O burst refers to the period during which a process is waiting for I/O operations to complete, such as reading from or writing to a disk or waiting for user input. During this time, the process is blocked, and the CPU remains idle.

Why can it be beneficial to schedule short CPU burst processes to the CPU before long CPU burst processes?
Scheduling short CPU burst processes before long CPU burst processes can provide benefits in terms of responsiveness and overall system performance. Short CPU burst processes typically complete their execution quickly and release the CPU, allowing other processes to run sooner. This leads to reduced waiting times and improved system throughput.
By prioritizing short CPU burst processes, the system can provide a more interactive and responsive experience to users, as their tasks are executed promptly. Additionally, it can help in achieving better CPU utilization by keeping the CPU busy with frequent context switches between short burst processes.

What does it mean a CPU scheduling algorithm is preemptive or non-preemptive?
A CPU scheduling algorithm can be classified as preemptive or non-preemptive, based on whether it allows for the interruption of a running process before it completes its CPU burst.
A preemptive scheduling algorithm has the ability to preempt or interrupt a currently executing process and allocate the CPU to another process. Preemption can occur when a higher-priority process becomes available or when a process voluntarily yields the CPU. Examples of preemptive scheduling algorithms include Round Robin and Shortest Remaining Time First (SRTF).
On the other hand, a non-preemptive scheduling algorithm does not interrupt a running process until it voluntarily releases the CPU or blocks for I/O operations. Once a process is allocated the CPU, it keeps the CPU until it finishes its execution or enters a waiting state. Examples of non-preemptive scheduling algorithms include First-Come, First-Served (FCFS) and Shortest Job Next (SJN).

What does it mean a CPU scheduling algorithm is starvation free?
A CPU scheduling algorithm is considered starvation-free if it guarantees that every process in the system will eventually get a chance to execute, regardless of its priority or arrival time. In other words, no process should be indefinitely delayed or prevented from running due to the scheduling algorithm's behavior.
Starvation can occur when a process with lower priority or longer burst time is continuously bypassed by higher priority or shorter burst time processes, leading to unfairness and reduced overall system performance. A starvation-free scheduling algorithm ensures that every process has a fair chance to execute and prevents indefinite postponement.

First-Come-First-Served (FCFS) algorithm
The First-Come-First-Served algorithm is a non-preemptive CPU scheduling algorithm where the process that arrives first is allocated the CPU first. It follows a simple concept of serving processes in the order of their arrival time. Once a process starts executing, it continues until it completes or performs I/O operations.
Advantages of FCFS:

1. Simplicity: FCFS is easy to understand and implement since it follows a straightforward rule of serving processes based on arrival time.

Disadvantages of FCFS:

1. Convoy effect: If a long CPU burst process arrives first, it can cause other shorter processes to wait, resulting in poor overall throughput and increased waiting times.
2. Lack of responsiveness: FCFS can lead to longer response times, especially if a CPU-intensive process arrives early, making interactive processes wait for a significant time.

Shortest-Job-First (SJF) algorithm

The Shortest-Job-First algorithm is a non-preemptive or preemptive CPU scheduling algorithm that selects the process with the shortest burst time to be executed first. It aims to minimize the average waiting time by prioritizing processes with the smallest CPU bursts.

Advantages of SJF:

1. Minimized waiting time: SJF prioritizes short CPU burst processes, leading to reduced waiting times and improved overall throughput.
2. Efficient utilization: By executing shorter processes first, the CPU stays busy with frequent context switches, achieving better CPU utilization.

Disadvantages of SJF:

1. Difficulty in predicting burst times: The burst time of a process is typically not known in advance, and accurate estimations are challenging. This makes it difficult to implement SJF effectively.
2. Potential for starvation: If shorter processes consistently arrive, longer processes may face starvation and prolonged waiting times.

How can the system know the duration of the next CPU burst (that hasn't happened yet!)?

The system typically estimates the duration of the next CPU burst based on historical information or statistical analysis. Several techniques can be used to predict the duration of the next CPU burst:

1. Exponential Averaging: The system keeps a record of past CPU bursts and calculates an exponentially weighted average. This average is then used as an estimate for the next CPU burst duration.
2. Moving Average: Similar to exponential averaging, the system maintains a moving average of the recent CPU burst times. The moving average provides a more balanced estimate by considering a fixed number of past burst times.
3. Regression Analysis: By analyzing the historical data of CPU burst times, the system can employ statistical techniques such as regression analysis to predict the next CPU burst duration. These techniques try to find patterns and correlations between process characteristics and burst times.

It's important to note that these prediction techniques are not always accurate, and there can be variations and uncertainties in the actual CPU burst durations. The system continuously updates and refines its estimates based on observed burst times to improve the accuracy of future predictions.

Round-Robin algorithm

The Round-Robin algorithm is a preemptive CPU scheduling algorithm where each process is assigned a fixed time slice or quantum of CPU time. Processes are executed in a circular manner, and when a time slice expires, the CPU is preempted and allocated to the next process in line. The preempted process is then placed at the end of the ready queue to wait for its next turn.

Advantages of Round-Robin:

1. Fairness: Round-Robin provides fairness by ensuring that each process receives an equal share of CPU time. It prevents long-running processes from monopolizing the CPU and allows all processes to make progress.
2. Responsiveness: Since processes are given small time slices, the Round-Robin algorithm provides good response times, making it suitable for interactive systems.

Disadvantages of Round-Robin:

1. Poor performance with long CPU bursts: If a process has a long CPU burst, it may need multiple time slices to complete its execution, causing delays for other processes waiting in the ready queue.
2. Inefficiency with short CPU bursts: Round-Robin introduces overhead due to frequent context switches, which can be inefficient when processes have short CPU bursts.

Priority scheduling

Priority scheduling is a preemptive or non-preemptive CPU scheduling algorithm that assigns a priority value to each process. The priority value determines the order in which processes are scheduled for execution. The highest priority process is selected for execution, and processes with the same priority are scheduled using another scheduling algorithm, such as Round-Robin.

Advantages of Priority scheduling:

1. Prioritization: Priority scheduling allows processes with higher priority, such as important tasks or real-time processes, to receive preferential treatment and have quicker access to the CPU.
2. Customizability: The priority value assigned to processes can be adjusted dynamically based on factors like process characteristics, user requirements, or system policies.

Disadvantages of Priority scheduling:

1. Starvation: Lower priority processes may suffer from starvation if higher priority processes continuously arrive, as they may have limited or no opportunities to execute.
2. Lack of fairness: If priority values are not carefully assigned or managed, processes with lower priority may experience significant waiting times or may not receive sufficient CPU time.

Aging
Aging is a technique used in priority-based scheduling algorithms to address the issue of starvation. It involves gradually increasing the priority of processes that have been waiting in the system for a long time. By increasing the priority over time, aging ensures that lower priority processes eventually receive a chance to execute, preventing indefinite delays and starvation.
Priority inversion and priority donation: Priority inversion occurs when a low-priority process holds a resource that a high-priority process needs. This can result in the inversion of priorities, where the low-priority process prevents the high-priority process from executing, causing performance issues. Priority donation is a mechanism that allows a high-priority process to inherit the priority of a low-priority process holding a resource, preventing priority inversion and ensuring timely execution of critical tasks.

Real-time scheduling
Real-time scheduling is a type of CPU scheduling designed to handle tasks with strict timing requirements, typically in systems where timely response and meeting deadlines are critical. Real-time systems must ensure that tasks are scheduled and completed within their specified deadlines.

Soft Real-Time Systems
Soft real-time systems are characterized by timing constraints that are important but not critical. In these systems, meeting deadlines is desirable but not mandatory. Soft real-time applications aim to optimize overall system performance while still providing timely responses. If occasional deadline misses occur, they may not have significant consequences.
Examples of soft real-time systems include multimedia streaming, video games, and data analytics applications. In these cases, missing an occasional frame or delaying a result by a small margin may not significantly affect the overall user experience or system functionality.

Hard Real-Time Systems
Hard real-time systems are characterized by strict timing constraints and critical deadlines. Meeting these deadlines is essential for proper system operation and may have severe consequences if missed. In hard real-time systems, even a single missed deadline can lead to system failure, safety hazards, financial losses, or other serious implications.
Examples of hard real-time systems include aircraft control systems, medical equipment, industrial automation, and real-time monitoring and control systems. In these applications, precise timing is crucial to ensure the safety, reliability, and proper functioning of the system. Any delay or failure in meeting deadlines can have catastrophic consequences.
The distinction between soft and hard real-time systems lies in the consequences of missing timing constraints. While soft real-time systems prioritize overall system performance, hard real-time systems prioritize meeting critical deadlines and ensuring deterministic behavior. The design and implementation of hard real-time systems often involve strict scheduling algorithms, predictable response times, and thorough analysis of worst-case scenarios to guarantee deadline compliance.