OBL3 - Operating Systems

By: John Ivar Eriksen

1 - Synchronisation

1.1

> The principle of process isolation in an operating system means that processes must not have access to the address spaces of other processes or the kernel. However, processes also need to communicate.
> (a) Give an example of such communication.
> (b) How does this communication work?
> (c) What problems can result from inter-process communication?

a)

Examples of communication between processec ("Inter-Process Communication", IPC) could be a web browser spawning a new process for each tab opened, a task being split up into several subtasts which are being worked as separate processes, or when multiple users/programs/processes need access to the same file or other piece of information. The latter example would be easy to picture as separate clients accessing shared information on a server.

b)

In a broad sense, the communication works by passing information to a "neutral" space, like a shared memory region, via the kernel, or more directly via pipes and sockets.

Shared memory: A process that need to communicate with other processes will establish a shared memory region in its own address space. Other processes that want to communicate with this process can then attach this shared memory to their own address space. This sharing of memory must be accepted, or agreed upon, by all the involved processes, since reading/writing to memory outside one's own space it (usually) not permitted by default by the operating system.

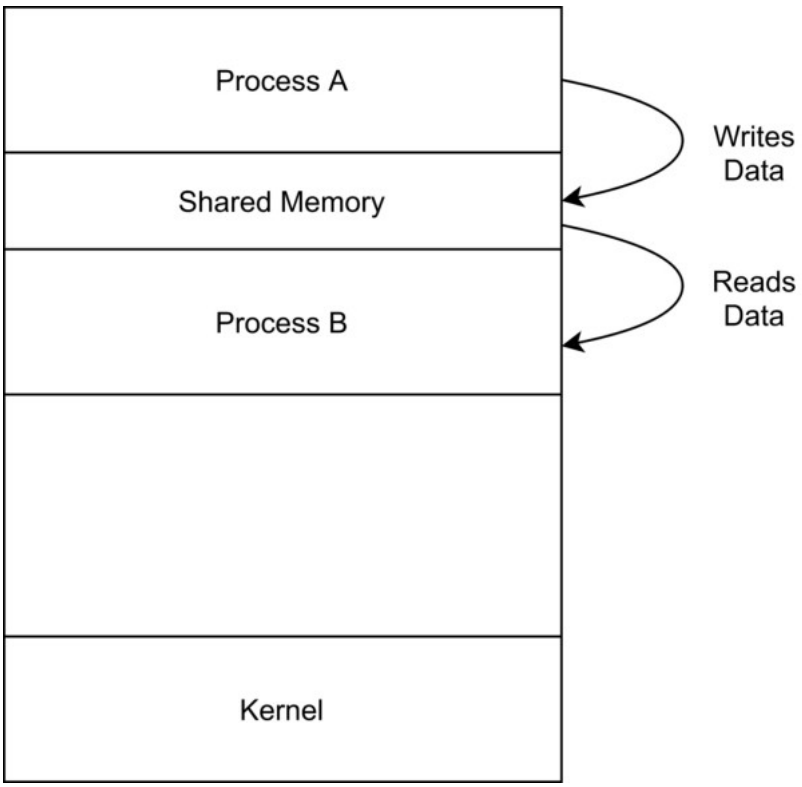

Image: sharedMemory-01.jpg

Image source: https://www.baeldung.com/cs/inter-process-communication

Message Passing/Message Queue: The processess can communicate by passing information to each other via the operating system kernel. The kernel essentially acts as a currier or mailbox for the processes. These messages can either be directed at a specific recipient, e.g. "from Process A to Process B", or they can be just sent to the kernel, where another process can find it.

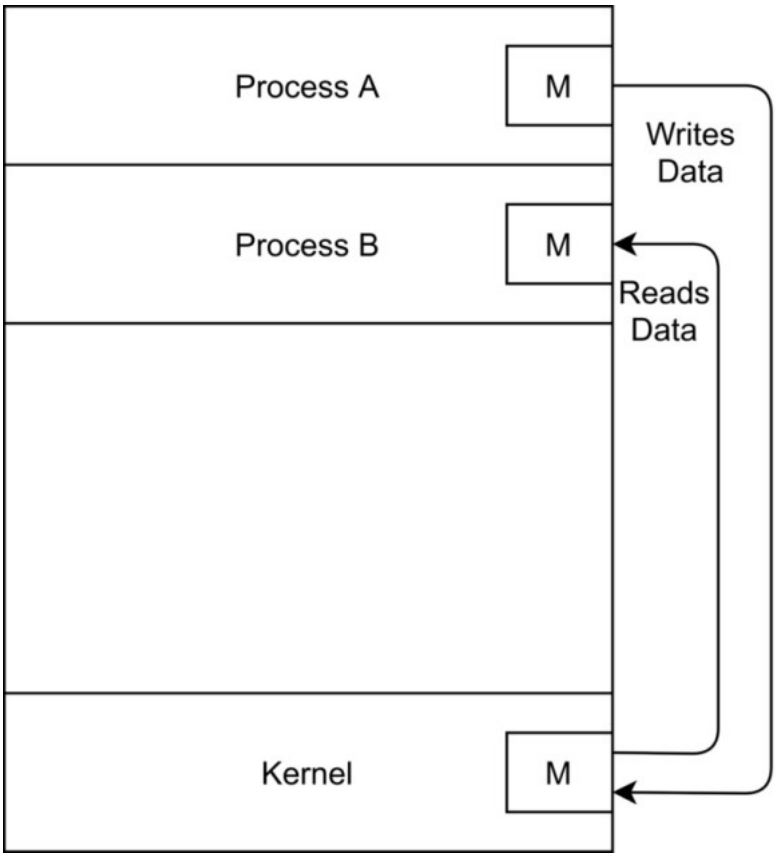

Image: messagePassing-01.jpg

Image source: https://www.baeldung.com/cs/inter-process-communication

Pipes: One-way communication between a parent and a child porcess.

Sockets: Usually used on networks, but works locally too. Sends data to a different process on a computer on the network, or to the same computer (i.e. sends it to itself).

Mutex and Semaphore: Locking mechanisms to control access to resources by locking out or blocking other processess from accessing it.

c)

Potential problems that can arise are:

- **Deadlocks:** A state where each process is waiting for the other to release resources, and where none of them will be releasing said resources.
- **Race Conditions:** If two rocessess are trying to access the same shared resource at the same time, the outcome may be unpredicatble behavior on part of the system. Which process gets access to the resource is down to timing, as they're both attemtping to be the first to access it. In other words, it becomes a race to get to the resource first.
- **Good Old Security Risk:** If not carefully handled, this communication can result in inadvertently giving unauthorized access to sensitive data during communication.
- **Overhead:** IPC consumes system resources, which may affect system performance, especially if handled poorly.
- **Complexity:** System/software designs become more complex, thus increasing workload in terms of both system load and maintenance of the software

1.2

> What is a critical region? Can a process be interrupted while in a critical region? Explain.

Asumption: "critical region" is what has been referred to by the lecturer and the curriculum text book as "critical section". It seems that these two terms are used interchangeably. Mostly, although sometimes not.

> In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. One way to do so is known as a critical section or critical region. Wikipedia

Critical Region (or Section) (p.203), refers to a sequence of code where a thread or process atomically accesses shared resources, such as data structures, network connection, peripheral device and variables. While a process is operating in this critical region/section, is must have exclusive access to the shared resources in order to avoid data corruption, race conditions or other concurrency problems.

While in a criticla section, a process can not be interrupted by another process, i.e. all other processes are locked out of being able to access the resource in question. This is achieved via various mechanisms such as Semaphores, lock/mutex (p.198), and in some cases by simply disabling interrupts while a process is in critical section. This prevents deadlocks or race conditions, and allows for atomic operations which ensure that the data is not corrupted by having several processes working on it and that the data is consistent with the single process.

1.3

> Explain the difference between busy waiting (polling) versus blocking (wait/signal) in the context of a process trying to get access to a critical section.

Polling (p.205, c.5.4) is where a process is continiously checking the shared state for changes, or try applying the objective of the process. This is much like kids in the backseat asking "are we there yet, are we there yet, are we there yet" ad nausem. In other words, it's like a while-loop that runs and check the shared state until it has achieved its goal. This consumes resources unnecessarily. This is calles "busy waiting", since the process is being very active in asking if it can proceed while also waiting. This method may also actually prevent the process on which it is waiting to finish because it's continiously (attempting) to acquire a lock on the shared state.

Blocking, on the other hand, utilize a *condition variable* ("CV") to let the process wait without being "active". It uses three methods for controlling processes: `wait`, `signal` and `broadcast`. The CV is a condition that has to be met for the process to proceed. "Blocking" in this context means that the process is on `wait`, i.e. put on a "waiting list" and put to sleep, thus prevented from trying to lock on to the shared state as long as the condition is not met. The `signal` is a call that takes one thread/process off the waiting list, and marks it as eligible to run by putting it on the scheduler's ready-list. `broadcast` is essentially the same as `signal`, only it takes all threads off the waiting list, marking them as eligible to run. This is a more resource effective approach than polling, as it lets the scheduler enfore priority and fairness, while also not wasting CPU resources on the constant polling.

1.4

> What is a race condition? Give a real-world example.

A "race condition" is when processes are trying to access the shared state at the same time. Since none is given priority, or locked out, which process gets to access the resources is down to timing, literally "who gets there first". As the name suggests, it's a race, and the results of the program execution is down to who wins this race.

A real-world example could be a banking system. Given an account with a balance of 1000 [NOK/€/$/£], then simultaneously initiate two separate transactions, withdrawing 900 and 800. They both check the balance, which is 1000, i.e. both transactions confirm
1000 - 900 - 800 = -700$

The shared resource here is the bank balance, where the two transactions are competing processes. They both "race" to complete their transaction, and since synchronisation hasn't been properly implemented here, they are both allowed to proceed and whoever gets there first gets to run first. Ideally, only one of the transactions (processes) should be allowed to check and modify the account balance at any given time.

1.5

> What is a spin-lock, and why and where is it used?

A spinlock is a lock that puts any thread trying to acquire it to be put in wait in a loop where it repeatedly checks if the lock is available. It's a type of "busy waiting, and can be pictured as to be "spinning" in place in a loop, getting nowhere, while looping the request to acquire lock.

As long as it's used on threads that will only be blocked for a short time, i.e. for threads that the kernel "know" will be run soon, they are efficient in that they don't cause overhead by requiring reshecduling or context switching. However, if they are kept for longer durations they risk becoming inefficient, or downright deadlocks, as they may prevent other threads from running and require to be rescheduled. As time held in spinlock increases, the likelyhood of the OS interrupting the thread goes up. This may result on other threads be left spinning, waiting for the same lock that "our" thread is unable to release. This will delay everything until the thread holding the lock can finish and release the lock.

Spinlock are most commonly used in systems where the expected wait time to acquire locks are very short. Some examples are:

- Low-Lever system code in. operating systems where code cannot be put on sleep, and where the programming language support atomic operations.
- Real-Time systems, where timing is more immportant than pure throughput. If designed well, the predictability of a well-made spinlock may be preferable to putting threads to sleep and wake them.
- Cases where we want to **avoid context switching overhead**, i.e. situations where the cost os putting a thread to sleep and waking it up is higher, or takes more time, than the cost of using a spinlock. This also presupposes a design where the spinlock is expertly implemented.
- **SMP (Symmetric Multi-Processing) systems.** According to Linux Device Drivers 3rd Edition, chapter 5, page 117, spinlocks are "by their nature, intended for use on multiprocessor systems, although a uniprocessor workstation running a preemptive kernel behaves like SMP, as far as concurrency is concerned. If a nonpreemptive uniprocessor system ever went into a spin on a lock, it would spin forever; no other thread would ever be able to obtain the CPU to release the lock. For this reason, spinlock operations on uniprocessor systems without preemption enabled are optimized to do nothing, with the exception of the ones that change the IRQ masking status. Because of preemption, even if you never expect your code to run on an SMP system, you still need to implement proper locking." (emphasis mine)

However, spinlocks seems to be much more risk than reward, and is (as far as I can tell from the litterature and "opinions" of the online community) not something that provide a lot of advantages in modern computing, as they are liable to clog up the system by having "everyone" wait for the the spinning thread if improperly implemented.

1.6

> List the issues involved with thread synchronisation in multi-core architectures. Two lock algorithms are MCS and RCU (read-copy-update). Describe the problems they attempt to address. What hardware mechanism lies at the heart of each?

(Some) issues invovled with thread synchronisation are:

- Cache coherence: When multiple cores have a common memory resources stored in their cache, an update to one of the copies would result in the two copies being incoherent with each other, even though they're supposed to be "the same".

- Lock contention: When multiple threads from different cores attempt to acquire the same lockm they are "contending" for it, which will consume additional system resources to sovle and get back into order.
- False sharing: Threads running on different cores may be accessing data A that, while not the same, shares a cache line/block with some other data B. If data B were to be modified, the caching protocol may require that data A reloads the entire block, even though no data *in* A was modofied. This is because the cashing systems isn't aware of whats happening specifically inside the block, only that "something was modified". Then A must bear the cost of the reloaded .
- Thread starvation: When several threads are trying to acquire a lock, a thread may be positioned such that it never gets access to the lock and resources, always having the other threads getting the lock, thus being starved.
- Deadlock: When threads are stuck in a loop of trying to acquire a lock, and no threads are able to relase the lock.
- Priority inversion: A high-priority thread may end up waiting for a low-priority thread, vioalting the priority model. This may occur when there's resourcer contention with a low-priority task that is preempted by a medium-priority task.
- Load imbalance: Some core may be given a higher workload than others, leading to inefficient use of the available resources.
- [Memory barriers]https://en.wikipedia.org/wiki/Memory_barrier(): Used to enforce ordering constraints on memory operations. This is usually only used in low-level code. They ensure that modern CPU don't perform out-of-order executions, but can cause problems with concurrency, unless very carefully and expertly implemented.

MCS - Mellor-Crummey and Scott

Problems addresses: MCS lock aimt to reduce lock contention and provide fairness for the threads. Where spinlocks can cause the so-called "Thundering herd problem" (Wikipedia), where a large number of waiting threads all rush in to try to acquire the lock when released, causes a spike in memory and CPU usage. By maintinging a linked list of waiting threads, where each threads has its own local node or flag on which to spin. This differs from the "traditional" spinlock, where all waiting threads are spinning on the same variable, which results in high contention for the lock once the threads wake up.

Whan a thread find the lock already held by some other process, it will append itself to the linked list of waiting threads and then "spin" on it's own node. When the lock is released, the next thread in the list, which is essentially a queue, will be notified and allowed to acquire the lock. This reduces contention and makes the process more fair.

Hardware mechanism: To keep maintain the linked list, MCS uses atomic operations, which are special CPU instructions, designed to perform certain tasks without being interrupted and execute as a single unit. These instructions are very close to the hardware level.

RCU - Read-Copy-Update

RCU is designed to allow for reads to occur concurrently with updates on the shared state. This allows for more processes to access the same data at the same time, since the reader does not need to acqure a lock. When a modification is made to the data, a new copy is made and updated, instead of modifying the original in place. This ensures that readers are not interrupted, which may cause issues. This means that there may/will exist (an) old version(s) and a new version of the data at the same time. Via integration with the scheduler, it is ensured that all readers complete their reads withing a grace period, after which it garbage collect the old versions, removing the outdated data and reclaiming the resources used for holding it.

Hardware mechanism: RCU uses memory barriers to ensure that once a read-side critical region has started, the read will see a consistent view of the data structure even if updates are made concurrently to said data structure. Additionally, it uses atomic instructions for publishing the updated version.

2 - Deadlocks

2.1

> What is the difference between resource starvation and a deadlock?

A deadlock refers to when multiple processes are waiting for each other to complete their work or release a lock, which results in none of them getting executed. Also known as a "Circular Wait".

Starvation is when higher priority processes keeps being exectued, effectively blocking any process with low priority. The lower priority process is "stuck" in the back of the queue, always being passed by higher priority processes.

2.2

> What are the four necessary conditions for a deadlock? Which of these are inherent properties of an operating system?

The four conditions for a deadlock are:

1. **Mutual Exclusion:** At least one resource must be held in a non-shareable mode, i.e. only one process can use the resource at any given time.
2. **Hold and Wait:** A process must be holding at least one resource while it's waiting to acquire additional resources held by other processes.
3. **No Preemption:** Resources can't be forcible taken away from a process holding it until the process releases it voluntarily after having completed it's task. Taking this resource away before the process has finished is called Preemption.
4. **Circular Wait:** When the other conditions are aligned, and we have processes waiting for each other to finish, we end up with the condition of a Circular Wait. Everyone is holding on to their resources while waiting for everyone else to release theirs.

Of these, the first three are inherent properties of the operating system.

- Mutual exclusion require locks to work in the MUTEX fashion.
- Hold and wait is a condition the OS sets for a process to procees.
- The lack of preemption is a design choice, i.e. it would be considered an inherent feature of the OS.

The fourth, Circular Wait, is a condition that arise as a consqeunece of the first three.

2.3

> How does an operating system detect a deadlock state? What information does it have available to make this assessment?

The OS can detect a deadlock state by employing algorithms and techniques that monitor the relatinship between resources and processes:

Resource Allocation Graph (RAG) algorithm: A graph that represents the relationship between processes and resources, i.e. how many resources are allocated to each process. The processes and resources are represented as the vertices, while the edges represent the allocation or assignment/request relationship between them. This way it can determine how much resources are allocated, and how much will be needed in the future. If a cycle in the graph occurs, then this would be identified as a deadlock. This only work if all resources exist in a single instance. If not, then Banker's algorithm is used to determine the presence of a deadlock.

Banker's Algorithm: Keeps track of the resource allocation required for each process to complete, and the total available resources on the system. The OS will typically "know" the current allocation, future requests, available resources and the wait-for-relationships (which processes are waiting for resources, and for what they are waiting). Using this data, the OS can make a *safe sequence*, i.e. is the sequence in which the porcesses can be run without suffering a deadlock.

Wait-for-graph: Akin to Resource Allocation Graph, but more specific in that it only represent processes and their dependencies on each other for resources. The graph will "show" only the Resource vertices, and only vertices that are waiting for (dependent upon) one another.

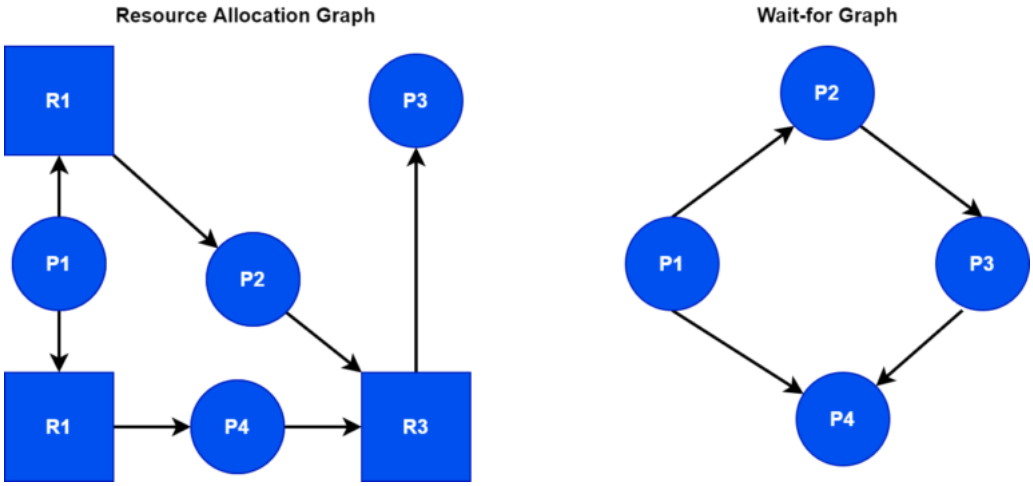

Image: RAG-and-WfG.png

RAG and Wait-for-graph, image source: https://www.baeldung.com/cs/os-deadlock

Resource Allocation Tables: Tables where the current state of resource allocation is maintained. It will monitor which resources are allocated to which processes, and what resources are currently available for use by other processes. A deadlock would show up something like this in the table:

Process	Holds	Waits for
P1	R1	R2
P2	R2	R1

Where P1 = Process 1, P2 = Process 2, R1 = Resource 1, and R2 = Resource 2.

To detect and assess a deadlock, the system typically have access to (among others) the following information:

- **Resource allocation state:** The OS is aware which resources are currently allocated to which processes. Overview is often maintained in the Resource Allocation Table, or a similar structure.
- **Resource request queue:** The OS keeps track of which processes have requested which resources and are currently waiting for them. This queue can help identify cycles of dependencies.
- **Available resources:** What resources are free and available is known by the OS.
- **Process state:** The OS knows the state of each process, i.e. if it's waiting, running, etc. This can be used by deadlock detection algorithms.
- **Resource characteristics:** "Metadata" about each available resouce. Whether it's shareable, how many available instances, etc. This is useful for more advanced methods and algorithms for detecting and avoiding deadlocks.
- **Priority information:** Info about process priority (high, medium, low) can be used for predicting and preventing deadlocks.
- **Historical data:** Historical data detailing usage patterns can be used to recognise and prevent conditions which may lead to deadlocks.
- **Wait-for-graph:** The OS may construct and maintain a wait-for-graph (mentioned above, too) which represents dependencies between processes and resources. Cycles here would be a predictor of a potential deadlock.

3 - Scheduling

3.1 - Uniprocessor scheduling

3.1a

> (a) When is first-in-first-out (FIFO) scheduling optimal in terms of average response time? Why?

FIFO scheduling is optimal in systems where the processes have similar (or at least predicable and known) execution times, and arrive at the same time or in ascending order of execution times (shortest first).
This require little management by the scheduler and OS, and there is minimal overhead from context switching, reorganization of the process queue or scheduling.
If the environment has no preemption, FIFO also ensures that there is no starvation, since every process will be run eventually. Even if stuck in a "convoy", a short job behind a long job will eventually get run.

3.1b

> (b) Describe how Multilevel feedback queues (MFQ) combines first-in-first-out, shortest job first, and round robin scheduling in an attempt at a fair and efficient scheduler. What (if any) are its shortcomings?

MFQ (c.7.1.5, p.215 digital)

- **Responsiveness (SFJ):** Run shorter jobs first, as in SJF.
- **Low Overhead (FIFO):** Minimize the numnber of preemtive operations, as done with FIFO. This mininmize the time spent on making scheduling decisions.
- **Starvation-Freedom (Round Robin):** As in Round-Robin, all tasts should make progress towards their goal.
- **Background Tasks:** Defer, i.e. delay, system maintenance tasks, so that they're not interfering wtih user work.
- **Fairness:** Assign non-background processes their approximate min-max fair share of the processor resources.

MFQ uses multiple Round Robin queues, with different priority levels and time quantum. Tasks with higher priority levels preempt tasks with lower priority, and tasks at the same level of priority are scheduled with Round Robin. Additionally, higher priority levels have shorter time quanta than lower levels.
If a task is waiting for I/O, it will yield the processor to another task, but will stay in their level, not being penalized, but may even be bumped up a level.

A new task will be given the top priority level, and each time the task has used up its time quantum it will drop a level. Tasks are moved between levels to favor short tasks over long ones.

To ensure starvation freedom and max-min fairness, especially with many I/O-bound processes which will get presedence over compute-bound tasks (due to the stay-at-level or bump when waiting for I/O), the MFW scheduler monitors all processes to ensure that they all receive a fair share of the resources.
At each level of priority, the Linux kernel will maintain two queues. Tasks that have received their fair share will only be scheduled if all other processes at that level also have received their fair share. Tasks that have not gotten their fair share will periodically have their priority bumped up, while those who have received more than their fairs share will be reduced in priority.

The shortcomings of MFQ is that is more complex to implement than the simpler "stand-alone" algorithm, like FIFO, SJF and Round Robin.
It requires tuning paramters in order to work properly and efficiently, e.g. the crieteria for promoting and demoting tasks, and time quantum for each level.
The continious monitoring of tasks fo ensure fairness, and managing multiple queues, can introduce overhead, despite it being designed to *reduce* overhead.
In some scenarions, MFQ may not be able to guarantee optimal response times and CPU utilization. Some such scenarios are:

- **Bursty workloads:** Swithing between long I/O bursts and long CPU bursts. Results in the process jumping up and down the priority ladder, wasting time and CPU utilization.
- **High turnover of shorter tasks:** Many short tasks coming continiously will start at high priority, potentially starving longer, compute-bound, tasks in lower priority queues.
- **Misjudging job length:** MFQ uses past bevarior to predict future behavior. A process that "suddenly" acts in a different way than before, e.g. used to be I/O-bound but is now CPU-bound, may have it's priority midjudged and be sub-optimally prioritized.
- **Prioritizing I/O-bound tasks over Compute-bound tasks:** While often a good practice, favouring I/O-bound tasks too much may lead to delay or starvation for CPU-bound tasks, especially if I/O-bound tasks keeps getting bumped up the priorituy ladder.
- **Overhead from priority adjustments:** Constantly minoritoring, adjusting priorities, and moving processes between queues can introduce overhead. With high rates of context switching, this may affect CPU utilization and response times.
- **Fixed time quantum issues:** The fiexed time quanta for the different priority levels may favor some workloads over others. E.g. if the time quantum for the high-priority queue is too short, it may lead to context switching before tasks are able to complete their work.
- **User-manipulation of the scheduler:** A skilled user would be able to design their processes in such a way as to "game" the MFQ system, at thus keep their process at a higher level of priority. This would be relevant in multi-user environments.

3.2 - Multi-core scheduling

3.2a

> (a) Similar to thread synchronisation, a uniprocessor scheduler running on a multi-core system can be very inefficient. Explain why (there are three main reasons). Use MFQ as an example.

The reason would be the fact that a uniprocessor scheduler is designed with a single processor in mind, and would therefore not be made for actually using the multiple processors or cores available on a multi-core and -processor system. It would not have the ability to execute processes in parallel, and not have good techniques for solving resource contention.

By uni- and multi-processor system, it is understood to be a "CPU with a single or multiple cores".

To delve a little deeper:

- **Under-utilization of processor cores:**
  - By design, a uniprocessor will schedule tasks as if there's only a single core available. This means that at any given time, only a single task would be actively executed, while the other cores remain idle.
  - For singlecore-MFQ, even if there are multiple tasks in multiple queueus ready to be executed, the scheduler would only pick one task for execution, wasting the resources available in the idle cores, resulting in *one* core working like a champ, while all the others does nothing. This is under-utilization resources and will lead to poor thoughput compared to what the system is capable of.
- **Lack of parallel execution:**
  - Where multi-core systems are designed to run and execute multiple tasks in parallel, a unicore scheduler would not be able to take advantage of the paralellism available.
  - In a singlecore-MFQ model, with several short tasks at the highest piroirty level, a unicore MFQ would run them serially on a single core, instead of potentially running them simultaneously in separate cores. Since the unicore-MFQ does not use the opportunity for parallel execution, it will increase the time equired to process and finish the tasks.
- **Increased contention and overhead:**
  - In a multicore system, threads often need to communicate and synchronize with each other, especially when they have shared resources or data. If tasks are scheduled on a single core, due to a uniprocessor scheduler implementation, this communicatino or synchronization would require context switching, which adds overhead to the system.
  - In multicore MFQ, tasks that require frequent synchronization would ba able to run in parallel on different cores, thus allowing for efficient synchronization. But in a uniprocessor-MFQ, these tasks would be scheduled in a queue, one after the other, on the same core. This would increase time required to complete and add overhead for synchronization due to the frquent need to context switch.

In short, a uniprocessor scheduler would not be effective in a multi-core environment, if seen in context of the amount of available resources. That said, is that it would not be less effective in the multi-core environment than if run in a single-core environment, which does make sense. The only sense in which it is less effective, is relative to the available resources it's not using.

3.2b

> (b) Explain the concept of work-stealing.

Work stealing is a scheduling strategy used on multithreaded computer programs. It allows for executing a *dynamically multithreaded computation*. It's designed for "fork-join model" of parallell computing, where computation can be viewed as a directed acyclic graph. The graph has a single start of computation (source), and a single end of computation (sink). The nodes in the graph representes eiter a join or a fork. Forks will produce multiple "logically parallel" computations, which are called threads or strands. Edges in the grap represents serial computations.

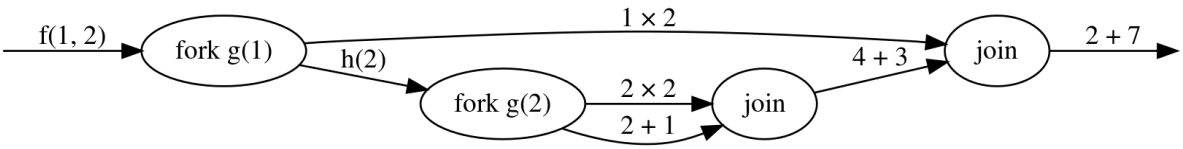

Image: Fork-join_computation.png

Image source: https://en.wikipedia.org/wiki/Work_stealing#/media/File:Fork-join_computation.svg

Basically, this means to use parallel computing to maintain a load balance among mutplie processors or threads by utilizing all available processing resources. This is done by allowing idle threads to "steal" tasks from busier threads.

Each processor in a computer system will have their own queue of work items. Each item consists of a series of instrucitons which are to be executed sequentially. Now, during execution, a work item may "spawn" a new work item that can be executed in parallel with its own work. While this new work item is initially put on the queue of the same processor as the item it was spawned from, any processer that runs out of work will be able to look at the queues of other processors and "steal" their work items. This leads to distributing the work over to idle processors (CPU cores), instead of them sitting around doing nothing.

The result is the aforementioned "dynamically mutlithreaded computation" model. Threads are created and managed dynamically during execution of a program, based on the needs of the program and available resources.

// End