**Mutex vs Semaphore vs Spinlock**

**Similarity**

– All of these are used for synchronization

**Difference**

Mutex provides **one person to access a single resource at a time**, **others must wait in a queue**. Once this person is done, the guy next in the queue acquire the resource.  
So access is serial, one guy after other. Too aggressive.

**Semaphore are useful if multiple instances (N) of a resource is to be shared among a set of users**. As soon as all N resources are acquired, any new requester has to wait. Since there is no single lock to hold, there is as such **no ownership of a semaphore.**

Spinlock is an **aggressive mutex**. In mutex, if you find that resource is locked by someone else, you (the thread/process) switch the context and start to wait (non-blocking).  
Whereas **spinlocks do not switch context and keep spinning**. As soon as resource is free, they go and grab it. In this process of spinning, they consume many CPU cycles. Also, on a uni-processor machine they are useless and perform very badly (do I need to explain that?).

OS/kernel perspective, what happens when a program is executed?

The machine instructions generated by a high-level language will be appropriate for the calling conventions for libraries providing those calls you make, including any system calls (albeit these are usually wrapped in a userspace library somewhere, so specifics about how to make a system call might not be necessary).

Additionally, it will be appropriate for the targetted instruction set architecture, with a few exceptions (care must be taken for example, about assumptions regarding pointer sizes, primitive types, structure layouts, class implementations in C++ etc.).

The file format will dictate the necessary hooks/publically visible functions and data to enable the operating system to execute your code as a process, and to bootstrap the process to the required state. If you're familiar with development for C/C++ under Windows, the concept of subsystem dictates the level of bootstrapping, resources provided, and entry point signature (normally main(int, char \*\*) on most systems).

There are some good examples of how the choice of high-level language, instruction set architecture, and executable file format might affect the ability to run a binary on any given system:

Assembly languages must code for a specific ISA. They use instructions that are specific to a family of CPU types. These instructions may work on other families of CPUs, if those CPUs support the given instruction set. For instance x86 code will work to a degree, on an amd64 operating system, and definitely work on an amd64 CPU running an x86 operating system.

C abstracts much of the specifics of an ISA. A few obvious exceptions include **pointer sizes** and **endianness**. Various well-known interfaces, will be provided to an expected level via libc, such as printf, main, fopen, and others. These include the expected register and stack states in order to make these calls, enabling C code to work on different operating systems and architectures without change. Other interfaces can be provided, either directly or by wrapping platform-specific into the expected interface to increase the portability of C code.

Python, and other similar "virtualized" languages operate at yet another level of abstraction, and again with a few exceptions, for instance features that don't exist on particular platforms, or character encoding differences, can run without modification on numerous systems. This is achieved by providing a uniform interface for many different ISA and operating system combinations, at the expense of performance and executable size.

Here are the general steps:

1> **Shell forks and execs the new process**. Shell waits for the completion of the process using wait4() unless forked process is a daemon (is a session leader and kills the parent and continues with the child after exec)

2> fork() actually creates clone of the shell process itself. This address space is overwritten by exec call. Exec call starts off by reading the first few pages of the executable from the disk. This operation involves file system, block layer, page cache and the device driver (if backend is disk, then disk device driver).

3> Kernel needs to be compiled with different binary file handlers. Some handlers present in Linux kernel are ELF, a.out, script etc. Header of the executable says what kind of executable it is. If kernel is compiled with the right kind of binary format handler, then the read data is handed over to the handler.

4> For example, if handler is an ELF handler, then the header has the location of the text in the file. Few pages of the text are read from disk (basically mmapped) along with all the directly linked libraries. Now the process is ready to start as its text and required libraries are in memory. Rest of the text is read using demand paging when needed.

Diffence between thread and process:

address space: separate vs shared

comm: via IPC vs via memory

Explain how memory management works: **Virtual Memory** Management

- Virtual memory, Page Fault

“volatile”: **skipping cache** and no compiler optimization (no moving)

A variable should be declared volatile “whenever its value could change unexpectedly”

1. Memory-mapped I/O (peripheral registers)

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application (ex) spin-lock in pthreads

“static” keyword

1. static variable

2. static function

Linux Kernel Memory Allocation -> ref

Semaphores, Mutex -> CA slides and ref

- MUTEX in assembly

- mutex vs binary semaphores

- semaphores, mutex, spin locks

- mutexes and spin locks in interrupts

- top halves and bottom halves in interrupt

How would you implement semaphores / mutexes in interrupt handlers ? Should we implement mutexes in interrupt handlers ? Can we sleep an interrupt ?

Spinlock and semaphore and its usage.

How to avoid interrupt coming on same CPU again and again? How to avoid deadlock in multiprocessor interrupt path for Linux.

Mutexes in interrupt Handler? Can we sleep an interrupt? (NO)

Semaphores cause tasks to sleep on contention, which is unacceptable for interrupt handlers. Basically, for such a short and fast task (interrupt handling) the work carried out by the semaphore is overkill. Also, spinlocks can't be held by more than one task.

The problem is that interrupt handlers (IH) are triggered asynchronously and in unpredictable way, out of the scope of any other activities running in the system. In fact, IHs run out of the scope of concept of the threads and scheduling at all. Due to this all mutual exclusion primitives which rely to the scheduler are unacceptable. Because they usage in the IH can dramatically increases the interrupt handling latencies (in case of IH running in the context of low priority thread) and is able to produce deadlocks (in case of IH running in the context of thread which hold the lock).

How does removing page table entry from one processor affects the performance in a mulch-processor system?

Basically when the page table entry is remove the TLB gets flushed and due to which if the other processor has a page table entry that was flshed by other processor then this other processor will have to get updated with the latest page table entry.

What are read-write semaphores and how does it degrade performance in a mulch-processor environment.

Similar to the above question when one process A on a CPU 0 acquires a write lock on a semaphore variable then if other process B on CPU 1 has acquired a read lock (something like this) then there is flushing required of the cache! (Did not understand what the interviewer explained :) )

Check the following list:

1) Bit manipulation

2) C macro questions

3) Find the error in the C code

4) Control Systems

5) Security - knowledge of vulnerabilities and threat models

process scheduling and cache management

System Level Programming

1. kernel mode vs user mode

2. what is printk()? (why using it?)

why padding is needed in struct

difference between interrupt and exception

Linux kernel memory allocation techniques?

paging, segmentation, dma

Does virtual memory solve the problem of fragmentation? If yes how or if no then how?

Difference between pointer and reference

1) pointer can be reinitialized and reference cant. and hence pointer can be incremented / decremented but reference cant.

2) pointer can be zero initialized but reference cant.

in a below program statement

int x = 3;

will 3 be fetched into the cache after executing the above line of code? It depends!

A. If there's optimization at all, it's fairly likely that 3 will never even make it into a register. Rather, the compiler will recognize that x has a value of 3 and will substitute 3 for the next use of x, possibly with, eg, an **add-immediate** instruction that doesn't first place the value in a register. Or the compiler may optimize x into a **register** and so the value of x will never be stored in memory and hence never go through cache.

And some processors have what's known as a "store through" cache, meaning that if x is assigned a storage location the value may be placed into that location without first/simultaneously being placed in storage cache.

So we can most definitely say that the value 3 might possibly appear somewhere in cache. Sometimes.

Here's pseudo-code for a ver simply reader/writer lock using a mutex and a condition variable. The mutex API should be self-explanatory. Condition variables are assumed to have a member wait(Mutex&) which (atomically!) drops the mutex and waits for the condition to be signaled. The condition is signaled with either signal() which wakes up *one* waiter, or signal\_all() which wakes up all waiters.

read\_lock() {

mutex.lock();

while (writer)

unlocked.wait(mutex);

readers++;

mutex.unlock();

}

read\_unlock() {

mutex.lock();

readers--;

if (readers == 0)

unlocked.signal\_all();

mutex.unlock();

}

write\_lock() {

mutex.lock();

while (writer || (readers > 0))

unlocked.wait(mutex);

writer = true;

mutex.unlock();

}

write\_unlock() {

mutex.lock();

writer = false;

unlocked.signal\_all();

mutex.unlock();

}

That implementation has quite a few drawbacks, though.

Wakes up all waiters whenever the lock becomes available

If most of the waiters are waiting for a write lock, this is wastefull - most waiters will fail to acquire the lock, after all, and resume waiting. Simply using signal() doesn't work, because you *do* want to wake up everyone waiting for a read lock unlocking. So to fix that, you need separate condition variables for readability and writability.

No fairness. Readers starve writers

You can fix that by tracking the number of pending read and write locks, and either stop acquiring read locks once there a pending write locks (though you'll then starve readers!), or randomly waking up either all readers or one writer (assuming you use separate condition variable, see section above).

Locks aren't dealt out in the order they are requested

To guarantee this, you'll need a real wait queue. You could e.g. create one condition variable for each waiter, and signal all readers or a single writer, both at the head of the queue, after releasing the lock.

Even pure read workloads cause contention due to the mutex

This one is hard to fix. One way is to use atomic instructions to acquire read or write locks (usually compare-and-exchange). If the acquisition fails because the lock is taken, you'll have to fall back to the mutex. Doing that correctly is quite hard, though. Plus, there'll still be contention - atomic instructions are far from free, especially on machines with lots of cores.

Conclusion

Implementing synchronization primitives correctly is **hard**. Implementing efficient and fair synchronization primitives is **even** **harder**. And it hardly ever pays off. pthreads on linux, e.g. contains a reader/writer lock which uses a combination of futexes and atomic instructions, and which thus probably outperforms anything you can come up with in a few days of work.

In Linux, we use virtual address.So each process will think it has 4 GB

memory space even if the real memory is only 2GB. Now suppose we do not have

MMU and programmer use real physical address in their program. We only have

small size of physical memory. How can we design the system?

You can design the system in 2 ways :

1. Implement the virtual memory yourself and use File IO operations if and when you need more memory than existing memory. You need to swap some of your data to a File to create space for new Objects.

2. You try to clean memory just like a garbage collector does and remove unwanted objects from time to time. This is inferior to 1 and you run the risk of your program crashing because of lack of memory.

You can use a combination of 1 & 2.

-------------------------------------------------------

In the new mobile phone, we can either choose to use a 1GHz solo core or

500MHz duo core processor. What tradeoffs should we consider?

1- Single 1Gig processor

Pros: High CPU frequency means shorter execution cycle, faster interrupt handler (better real-timeness, higher internal timer resolution) and less power consumption compared to a dual core due to the fact that there are less transistors on he chip to use power which results in higher battery life. Usually CPU clock rate also can be reduced by software to get further reduction in power consumption when Mobile is idle.

The silicon chip also would be smaller in size and would weight lighter due to reduction in number of populated transistors which makes it more suitable for small sized mobile phones and more portable.

cons: High frequency means higher radio interference in PCB level so it make PCB design more complicated to ensure level of the inducted noise remains in an acceptable level.

If the application is heavily multi-tasked then overhead of the CPU context switching can be also a problem by means of reduction in overall CPU performance.

1- Dual core 500Mhz processor

Pros: lower frequency, lower noise generation, lower chance for interfere with other devices in critical applications such as Airplane , simpler PCB design which means reduced cost of manufacturing, providing the real parallel processing which increases processing power in heavily muti-threaded applications. In critical applications having two CPUs may alos be considered as 1oo2 CPU redundancy so as to make sure if one CPU failed there might be still another CPU to take control and keep software running.

Cons:

Bigger silicon, heavier, more power consumption, slower interrupt handler ( increased interrupt response time)

------------------------------

Single core:

pros: +less complexity of HW design

+less complexity of SW design (here inside a lot of factors)

cons: -potentially higher power consumption

-in some cases higher context switching overheads

Dual core:

pros: +Allows more flexible system solutions

+In some cases can seriously increase perofrmance with high latency memory: twice less cycles to wait for memory. I had projects in which it played decisive role. It also allows to decrease memory frequency and save some power with same performance

cons: -HW design higher complexity

-requires mechanisms of resource sharing

Sequence of steps that happen in CPU, cache, TLB, VM, HDD leading to execution of “x = 7” which isn’t present in cache or sysmem nor translation in TLB. Also specify if any intrs, exceptions or faults are generated.

I have given a rough idea here. Not very clear about specific interrupts at each step. Hope this helps.

1) CPU first fetches the instruction x = 7 from the Instruction cache (when it reads this address in the PC/IR)

2) After decoding and executing the instruction, it sees that, it needs to access the memory location of variable x (which will be a virtual address)

3) Hence it issues a request to the TLB to return the physical address/tag.

Assuming the cache is Virtually indexed, it will parallely calculate the index for this virtual address.

4) Since it's a TLB miss, it accesses the Page table which resides mostly in Memory.

//Not sure what the interrupt here is?

5) But since, the translation is not found, meaning the page for this address is not in RAM, it issues a DMA request to transfer the page from Secondary storage to the RAM. It knows the address of the page on Secondary storage through the vm\_area struct for this process, which maintains the location of all the pages.

// This is done in page fault handler. Page fault is raised when this even occurs.

6) Once DMA is complete, the processor is interrupted with this event. It then updates the page table with this entry and also the TLB.

//This would be an I/O interrupt to the processor

7) Once it gets the tag, it checks if that tag matches in the cache.

8) It won't, since cache does not have this entry.

9) Hence it fetches this block (cache block) from memory and places into the cache and restarts the execution.

10) In the MEM phase of the execution pipeline, it writes the value 7 to this location in the cache.

-------------------------------

Case 1: No Swapping.

Here we assume, we have physical pages available, hence no swapping needed.

1. CPU fetches the instruction x=7 (We assume, code page is in memory) and decodes it.

2. CPU tries to find corresponding physical address of "x" by looking at TLB. Since there is no TLB entry present, (first stop the instruction) MMU (Memory management unit) raises TLB exception, otherwise know as TLB Fault.

3. Now, TLB fault handling code, visits PTE (Page table entries) to find corresponding physical page. Since, there is no physical page assigned, this causes Page Fault to be raised, which allocates a page in RAM and updates PTE.

4. Restart the instruction.

Note - We havent updated TLB entry yet. This would be done when the restarted instruction runs again, which would again raise TLB fault, but this time TLB fault handling code would find the corresponding physical page and would load the TLB entry and the instruction is restarted.

5. Thus finally, this time, there would be no TLB fault and Page fault and instruction would be successful.

Case 2 - Swapping.

In this case, RAM is out of physical pages and only way to find a physical page is to swap out a used physical page to the HDD and then use that page (the one that we swapped out) to assign during Page fault.

Rest remains the same. Only, Page Fault handling changes i.e. it involves swapping (HDD is involved). Note - We also need to update PTE of the page we swapped out to the disk, so that the process (whose page we swapped out ) knows about its page location.

=========================================================

Assuming you have three N bit unsigned integers a, b and c, what is the min number of bits you would need to store the result of a \* b + c?

Another easy way to understand this is : consider N to be 8 bits or 1 byte... An unsigned int of 1 byte can have a max value of 255. Now applying the given equation with the maximum values possible:

a\*b + c = 255\*255 + 255 = 65280

we know unsigned 8 bit value max is 255 . Similarly unsigned 16 bit value is 65536 and 65280 is less than 65536. Hence we need 16bits for storing the result or 2N bits.

-----

2N

This is how i calculated

(2^N-1)(2^N-1) + 2^N-1 = (2^2N - 1) - (2^N-1)