## Hardware/software interface layer models

1) Function driver – Bus driver – Intermediate software layer – Intermediate hardware driver – Bus devices – function device.

2) One common method to classify device (resp. driver) layers is by deciding whether devices of a layer interconnect other devices. If yes, these types of devices are referred to as bus devices.

## What is the difference between a process and a thread

Process:

1) A process is an instance of a program in execution.

2) Each process is an independent entity to which system resources (CPU time, memory, etc.) are allocated.

3) Each process is executed in a separate address space.

4) One process cannot access the data of another process, unless inter-process communication tools have to be used, such as files, sockets, etc.

Thread:

1) A thread uses the same stack space of a process. A process can have multiple threads.

2) Multiple threads share part of their states. Typically, one allows multiple threads to read and write the same memory. One thread can access other threads’ stack (I don’t’ think this is possible)

## How to measure the time spent in a context switch?

1) Time\_Stamp(P2\_1)-Time\_Stamp(P1\_1)

2) Context switch time = T-sum of all processes’ (execution time + wait time)

## What happen to stack after a function is called? (hits: 2) explain what is saved in thread context switch and compare with process context switch (hits:2)

\_\_cdecl convention

Out: the arguments are pushed in;

The return address (current EIP) is pushed in;

in: the stack frame pointer was pushed in;

The stack frame pointer pointed to the current stack pointer;

The stack pointer moved upwards to reserve a buffer space for temps

A bunch of general purpose registers were pushed in;

Some local variables are defined in the reserved buffer;

Do the work;

Pop the general purpose register;

Let the stack pointer points to the stack frame pointer

Pop the old stack frame pointer.

Out: clear the parameters;

Return;

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

There are 2 cases viz. Swapping/Non-Swapping:

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" in the cached memory lines, but is neither in L1 or higher level caches.

3) CPU have to find the moby looking at TLB. Since there is no TLB entry present, (first stop the instruction) MMU (Memory management unit) raises TLB exception, otherwise known 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 haven’t 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.

## What do you mean by address space of a process what essentially it is?

Address space of a process is the memory allocated to the process by the OS for completing its operation. the process gets exclusive rights to that address space unless the process wants to share something with some other process...this address space as allocated by OS is essentially belongs to Virtual Memory that gets mapped to the real one....

1> Stack

2> Heap

3> Data segment

4> Code segment

## Write a code to overflow a stack.

1) Create a pointer;

2) In an endless loop, keeps increase the pointer and fill the content with zero.

Recursively calling

## How linker resolved dynamic binding?

1) A dynamic linker loads (copies from persistent storage to RAM) and links (fills jump tables and relocates pointers) the shared libraries needed by an executable at run time, that is, when it is executed.

2) The term ‘dynamically linked’ means that the program and the particular library it references are not combined together by the linker at link time. Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references.

## Explain what is done in compilation phase and what is done in linking phase and where does assembler comes into picture if at all?

Compilation phase: Produces assembly code[Instruction sets defined for the Core].

Assembler: Convert assembly instructions into opcode-offset format ..object file[usually a relocatable code].

Linker: Link different object file via resolving references and consolidate them into an executable file.

## How to choose cache eviction?

A cache eviction algorithm is a way of deciding which element to evict when the cache is full. There are Least Recently Used (LRU) policy, and Least Frequently Used (LFU) policy.

Page replacement algorithms

Page replacement algorithms decide which memory pages to page out (swap out, write to disk) when a page of memory needs to be allocated. Paging happens when a page fault occurs and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

The theoretically optimal page replacement algorithm: when a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future. For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds. This algorithm cannot be implemented in the general purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used.

Not recently used: Least Recently Used (LRU), page replacement algorithm is an algorithm that favours keeping pages in memory that have been recently used.

First-in, first-out: The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little book-keeping on the part of the operating system.

Second-chance: It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out.

Clock: Clock is a more efficient version of FIFO than Second-chance because pages don't have to be constantly pushed to the back of the list, but it performs the same general function as Second-Chance. The clock algorithm keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location. If R is 0, the new page is put in place of the page the "hand" points to; otherwise the R bit is cleared. Then, the clock hand is incremented and the process is repeated until a page is replaced.

## make file

1) writing makefile is an art.

2) it gives the targets, the dependence, and the commands

3) if any file of the dependence is newer than the target file, the command will be executed.

## What is difference between named pipes and unnamed pipes?

Unnamed pipes are created used and destroyed by a set of processes.   
They have two ends one for writing and one for reading.   
They can link only two related processes. When its use is over they are automatically reclaimed.   
  
Named pipes or FIFO has a name and it remains in system till it is removed explicitly. It can link two totally separate processes as it outlives a process

# Memory Management (paging)

1. Memory Management techniques

First fit:

1) The allocator keeps a list of free blocks (known as the [free list](http://www.memorymanagement.org/glossary/f.html#free.list)) and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request.

2) If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block.

3) Memory location: This is not fast for allocation or recycling, but supports efficient merging of adjacent free blocks (known as coalescence [ˌkəʊə'lesns])

4) Increasing size: This is equivalent to the best fit algorithm, in that the free block with the "tightest fit" is always chosen.

5) Decreasing size: This approach encourages external fragmentation, but allocation is very fast.

6) Increasing time since last use: This is very fast at adding new free blocks, because they are added to the beginning of the list. It encourages good locality of reference (where blocks used together are not spread throughout memory), but can lead to bad external fragmentation.

Buddy system:

1) The allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence (see below for example), such that any block except the smallest can be divided into two smaller blocks of permitted sizes.

2) When the allocator receives a request for memory, it rounds the requested size up to a permitted size, and returns the first block from that size's free list.

3) If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.

4) When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size (coalescence). To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the "buddy" of any free block can be calculated from its address.

5) The rounding typically leads to a significant amount of wasted memory, which is called internal fragmentation.

Suballocators:

1) A suballocator obtains large blocks of memory from the system memory manager and allocates the memory to the application in smaller pieces.

To take advantage of special knowledge of the application's memory requirements that cannot be expressed to the system memory manager;

2) Many applications have one or two sizes of block that form the vast majority of their allocations. One of the most common uses of a suballocator is to supply the application with objects of one size. This greatly reduces the problem of external fragmentation. Such a suballocator can have a very simple allocation policy.

3) There are dangers involved in making use of special knowledge of the application's memory requirements. If those requirements change, then the performance of the suballocator is likely to be much worse than that of a general allocator. It is often better to have a memory manager that can respond dynamically to changing requirements.

1. What is MMU made up of? (With reference to data structures)

A current page table register, page tables of the current running processes: dictionary, hash table.

Where the memory map for the process is stored and is it in hardware or in software

Memory of a process is nothing but the virtual address space of that process. So the map of it, as in which physical page it resides, is stored in Page Table (single level or multi-level). Page Table is software, a part of the kernel.

1. Why we need virtual memory?

1) Virtual memory was designed to provide automated storage allocation. It gives the application an illusion of a very large amount of available memory.

2) In a virtual memory system, only the most-often used portions of a process’s address space actually occupy physical memory; the rest of the address space is stored on disk until needed.

3) Each running process generates addresses for loads and stores as if it has the entire machine to itself.

1. What is a page table?

1) Page table maps virtual page numbers (VPNs) to physical frame numbers (PFNs), physical memory is divided into frames to hold pages.

2) One page has only one frame, but one frame can be referenced by many pages, these frames are shared memories.

3) It is a dictionary.

1. How do you implement TLB in Operating System? (hits: 2)

1) Translation look-aside buffer caches the most frequently used page table entries in hardware, so a memory reference completes at the speed of hardware.

2) The TLB is first searched, if a miss happens, the page table was searched the entries was inserted it into the TLB.

1. What is page fault? Write a pseudo code for page fault handler

1) A page fault (sometimes #pf or pf) is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in physical memory.

2) In the typical case the operating system tries to handle the page fault by making the required page accessible at a location in physical memory or terminates the program in the case of an illegal access.

The nearly complete story

1) In the instruction fetch stage, TLB will be checked, if there are the page entries, fetch the instruction from the physical memory address. If no entries cached in TLB, page table will be searched. If the page table indicates that the page looking for is not currently on the physical memory, page fault happens.

2) Memory management unit (MMU) will raise an interrupt to the CPU. When the CPU detects the Interrupt signal before its instruction fetch cycle completes, it aborts the instruction cycle and acknowledge the interrupt. A page fault handler will be implemented.

3) DMA will raise an interrupt when it completes copying the page, and the CPU will suspend the current running program and make a subroutine call to the DMA interrupt handler.

Page fault handler:

1) Save the context from the CPU into the head of the Ready Queue. (A bunch of pushes)

// this is needed because the current program is no longer ready to run.

2) Perform a disk read operation using the location on disk information in the page table.

3) Remove the head of the Ready Queue and insert it into the DMAQ

// The Operating System must also introduce an additional variable in the PCB that can be used to indicate the reason for the DMA disk request, because an ordinary disk read (to obtain disk data is handled differently than a disk read to fetch a missing page).

4) Update current page table register of MMU to point to the page table of the program that is the new head of the Ready Queue

// this will switch the MMU to the page table from the program at the head of the ready queue.

5) Restore the context using the head of the Ready Queue

DMA handler:

1. Save context from CPU into the PCB at the head of the Ready Queue

2. Remove the PCB from the DMA Queue

3. Insert this PCB at the head of the Ready Queue

//This process is the NEW current running process!

Make the page table of this process the current page table by updating the current page table register in MMU \*\*\*\*\*\* (added for paging)

4. If (program had a page fault) {

Update page table entries in MMU

Specifically, you must:

1. Set the valid bit of the fetched page to VALID

2. Enter the page number for the fetched program page into the page entry

3. If you replaced an existing page, you must also invalid the corresponding page entry.

Usually, you replace one of YOUR own pages and do not "steal" a frame from another process)

}

5. Restore the context using the PCB at the head of the Ready Queue.

1. What is paging?

In computer operating systems, paging is one of the memory-management schemes by which a computer can store and retrieve data from secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. The main advantage of paging over memory segmentation is that it allows the physical address space of a process to be noncontiguous.

1. What is thrashing?

1) In computer science, thrashing occurs when a computer's virtual memory subsystem is in a constant state of paging, rapidly exchanging data in memory for data on disk, to the exclusion of most application-level processing.

2) This causes the performance of the computer to degrade or collapse. The situation may continue indefinitely until the underlying cause is addressed.

3) In virtual memory systems, thrashing may be caused by programs or workloads that present insufficient locality of reference: if the working set of a program or a workload cannot be effectively held within physical memory, then constant data swapping, i.e., thrashing, may occur.

1. Why do we use circular link list in place of any balanced binary search tree in storage allocator? One drawback is that to free() a chunk of memory allocator has to search the link list and then if found that address then release , so why not tree to reduce this search and merge?

1) It might be true for some allocators, but it's not true in general.

2) You can still find a specific memory block (node) by knowing its memory address. So, you'd have an O(1) access time.

3) Linked list fits for the FIFO policies.

1. From which type of memory does shared segment come?

heap, since global

# Pipeline (CPU)

1. What are differences between ARM and Intel CPU on high level architecture?

1) ARM has a number of RISC features, such as a large register set, fixed-length instructions, and purely load-store architecture. In comparison, 32-bit x86 has six registers that are nominally general-purpose, although a lot of instructions require the use of specific registers. It uses a variable-length instruction coding, and a number of instructions take a memory address as one of the operands.

2) The ARM instruction set also has some decidedly non-RISC features. The most well-known is predicated instructions, which execute conditionally based on the state of one of the condition registers. In a typical implementation, they execute anyway, but the results are stored only if the condition codes are matched.

3) Another well-known feature of the ARM instruction set is the addition of a barrel shifter in the pipeline, which operates on the second operand of most instructions, optionally shifting or rotating them by an arbitrary amount.

4) One of the advantages that x86 has over most RISC chips is instruction density. x86 instructions are variable-length, which means that common instructions typically have a shorter encoding and so take up less space in instruction cache.

The cost of supporting multiple instruction sets is increased complexity of the instruction decoder. The ARM instruction decoder takes a 32-bit word and just needs to test a few bits to know where to dispatch the instruction. The x86 decoder needs to read the bits in sequence, find breaks between instructions, and so on. On something like the Atom, the decoder can account for around 20% of the total power consumption.

The ARM solution is to add a second instruction decoder. Thumb code was introduced as a subset of ARM operations. The CPU is always in ARM, Thumb, or Thumb-2 mode (or, occasionally, in one of a few less-common modes). Because switching between modes requires an explicit instruction, an ARM chip needs three or more instruction decoders. However, each decoder is quite simple, and the chip needs to power only one at a time, combining the power efficiency of simple decoders with the instruction cache-usage of variable-length decoders.

1. Why branching is a bad thing for a processor? How does it slow the process down?

1) CPU always works in a fetch/decode/implement 3 depth pipeline mode.

2) Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

3) When the CPU is implementing the first instruction, and decoding the second instruction, it is also pre-fetch the third instruction. When branch happens, the pre-fetched instruction may not be the correct one. So the pipeline is stopped for many cycles to remove the incorrect instructions in the pipeline, and reload the right ones.

1. What is (BTB) branch target buffer? (hits:2)

1) BTB caches the recent used target addresses, indexed by the PC address.

2) It is accessible at the end of instruction fetch stage, so we could take the cached target address for the next instruction fetch.

3) The predictors decide whether take the branch or fall through.

4) 1-bit or 2-bit predictors are local predictors. It changes stages according to the history of one branch.

5) Pattern History Table (PHT) is the global predictor; it is indexed by the branch history of the global pattern register (GHR). GHR is FIFO register record the branch patterns.

6) “Gshare” predictor uses GHR “XOR” the PC to index the PHT.

1. Pipeline Forwarding

Results go to next instruction not from the stage 3 not stage 5.

# Scheduling

1. Multilevel feedback queue

Design requirements:

1) Give preference to short jobs.

2) Give preference to I/O bound processes.

3) Separate processes into categories based on their need for the processor.

Multiple FIFO queues are used and the operation is as follows:

1) A new process is positioned at the end of the top-level FIFO queue.

2) At some stage the process reaches the head of the queue and is assigned the CPU.

3) If the process is completed it leaves the system.

4) If the process voluntarily relinquishes control it leaves the queuing network, and when the process becomes ready again it enters the system on the same queue level.

5) If the process uses all the quantum time, it is pre-empted and positioned at the end of the next lower level queue.

6) This will continue until the process completes or it reaches the base level queue.

1) At the base level queue the processes circulate in round robin fashion until they complete and leave the system.

2) Optionally, if a process blocks for I/O, it is 'promoted' one level, and placed at the end of the next-higher queue. This allows I/O bound processes to be favored by the scheduler and allows processes to 'escape' the base level queue.

3) In the multilevel feedback queue, a process is given just one chance to complete at a given queue level before it is forced down to a lower level queue.

1. What is priority inversion?

1) Priority inversion is a problematic scenario in **scheduling** in which a high priority task is indirectly preempted by a medium priority task effectively "inverting" the relative priorities of the two tasks.

2) L, M, and H represent low, medium, and high priority tasks. L is using a shared exclusive-use resource with H, so H stops and wait for L. When properly designed, L relinquishes the resource promptly enough so H’s priority use is not hindered excessively. Surprising behavior happens, when M runs before H, since it does not need the shared resource.

1. In a multi-threaded process, if one thread is busy on I/O will the entire process be blocked?

If the process has only one thread, then yes.

If the process has multiple threads, then normally no if the operating system supports multithreading.

This question can also be addressed in terms of the underlying implementation of user threads. There are different models for multihreading models, in order to implement user threads they have to be mapped to a kernel thread:

Many-to-One: Many user threads to one kernel thread

One-to-One: Each user thread is assigned to a kernel thread.

Many-to-Many: Many user threads are split on different kernel threads.

In the many-to-one case, a single block-operation (system call) within the thread can block the whole process. This disadvantage is not present in the one-to-one model.

Implementing threads in user space:

1) Blocking system calls can't be executed directly since that would block the entire process. For example the producer consumer example, if implemented in the natural manner would not work well as whenever an I/O was issued that caused the process to block, all the threads would be unable to run (but see just below).

2) Similarly a page fault would block the entire process (i.e., all the threads).

3) A thread with an infinite loop prevents all other threads in this process from running.

4) Re-doing the effort of writing a scheduler.

5) Very fast since no context switching.

Implementing Threads in the Kernel:

1) Thread-create and friends are now system calls and hence much slower than with user-mode threads. They are, however, still much faster than creating/switching/etc processes since there is so much shared state that does not need to be recreated.

2) A thread that blocks causes no particular problem. The kernel can run another thread from this process or can run another process.

3) Similarly a page fault or infinite loop in one thread does not automatically block the other threads in the process.

1. After context switch processes which has interrupted other process starts running but how does os know which process to run after running current process. Similarly in case of interrupt how does interrupt handler know where it should go back?

The next process on the ready queue?? scheduling.

# IRS

1. Software Interrupt

A software interrupt, also called an exception, is an interrupt that is caused by software, usually by a program in user mode; such as SIGFPE (Signal Floating-Point Exception), SIGSEGV (Signal Segmentation Violation).

A signal is a software interrupt delivered to a process. Default handler is written in kernel, user-defined handler is in user space, user define handlers can override the kernel ones 44. Unix signal

Signals are a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems. A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred.

Exceptions such as division by zero or a segmentation violation will generate signals (here, SIGFPE and SIGSEGV respectively, which both by default cause a core dump and a program exit).

1. Nested interrupts

1) It should be noted that good programming technique implies that you keep interrupt functions very short. When you are using short interrupt functions, interrupt nesting becomes unimportant. When you are using a Real-Time Operating System (such as Advanced RTX), the stack usage of user tasks becomes unpredictable when you allow interrupt nesting.

2) It is required to switch the CPU into the User/System Mode. Otherwise it would be impossible to call other functions within the interrupt service routine.

3) You may nest interrupts by using the macros IENABLE and IDISABLE at beginning and end of an interrupt service routine. Depending on interrupt controller, it might be required to clear the interrupt source and acknowledge the interrupt. So in reality an implementation might look like the following example:

1. What is an interrupt vector?

1) An interrupt vector is the memory address of an interrupt service function.

2) When an interrupt is generated, the Operating System saves its execution state via a context switch, and begins execution of the interrupt handler at the interrupt vector.

1. ISR, interrupt service routine, what considerations, when you use ISR

1) Data consistency

2) If it happens to often, so we need to take DMA interrupt instead.

Some device raises an interrupt -> CUP save the current context -> look for the interrupt handler through the interrupt vector table -> jump to the address and do something -> return and restore the context.

1. What is the difference between ISR & function call

ISR:

Asynchronous event that can occur any time during the execution of the program

Saves the PC, Flags and registers on the stack and disables all the interrupts and loads the address of the ISR

ISR cannot have arguments that can be passed to it

Cannot return values

Enables the interrupts

Generally small as they are taking the time of some other process

Some of ISR have their own stack

Function:

Occurs whenever there is a function call

Saves the PC and registers on the stack

Can have arguments

Can return values

No restriction on the size and duration of execution

1. What is IDT (interrupt descriptor table) and what the processor does when an exception is triggered?

1) The Interrupt Descriptor Table (IDT) is a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions.

2) Use of the IDT is triggered by three types of events: hardware interrupts, software interrupts, and processor exceptions.

1. Explain about what you cannot do in a ISR

Any Lengthy code is not allowed: wait function, sleeping.

1. Softirqs, Tasklets, and Bottom Halves

We mentioned earlier in Section 4.6 that several tasks among those executed by the kernel are not critical: they can be deferred for a long period of time.

No softirq can be interrupted to run another softirq on the same CPU; the same rule holds for tasklets and bottom halves built on top of softirqs.

On a multiprocessor system, however, several deferrable functions can run concurrently on different CPUs.

Softirqs and bottom halves are statically allocated (i.e., defined at compile time), while tasklets can also be allocated and initialized at runtime (for instance, when loading a kernel module)

1) Many softirqs can always be executed concurrently on several CPUs, even if they are of the same type. Generally speaking, softirqs are re-entrant functions and must explicitly protect their data structures with spin locks.

2) Tasklets differ from softirqs because a tasklet is always serialized with respect to itself; in other words, a tasklet cannot be executed by two CPUs at the same time. However, different tasklets can be executed concurrently on several CPUs.

3) Finally, bottom halves are globally serialized. When one bottom half is in execution on some CPU, the other CPUs cannot execute any bottom half, even if it is of different type.

1. How would you handle sleeping or blocking instructions in an Interrupt Service Routine (if unavoidable) or basically if the length of ISR is long? How would sleep in a kernel? (hits:2)

SoftIRQs and tasklets cannot use any blocking instructions as they run in an interrupt context. Workqueues can only be used from a process context. So, I guess if the ISR is long, it can wake up a kernel thread, and let the kernel thread do the dirty work i.e. use blocking instructions.

A process can sleep in two different modes, interruptible and uninterruptible. In an interruptible sleep, the process could be woken up for processing of signals. In an uninterruptible sleep, the process could not be woken up other than by issuing an explicit wake\_up. wait\_event\_interruptible puts the process to an interruptible sleep. wait\_event puts the process into an uninterruptible sleep.

wake\_up - wakes exactly one exclusive sleeping process in TASK\_INTERRUPTIBLE or TASK\_UNINTERRUPTIBLE state from the wait queue

wake\_up\_interruptible - wakes only one exclusive sleeping process in TASK\_INTERRUPTIBLE from the wait queue.

The interruptible\_sleep\_on( ) function is identical to sleep\_on( ) , except that it sets the state of the current process P to TASK\_INTERRUPTIBLE instead of TASK\_UNINTERRUPTIBLE so that P can also be awakened by receiving a signal.

The schedule() Function: (lost wake up)

In Linux, the ready-to-run processes are maintained on a run queue. A ready-to-run process has the state TASK\_RUNNING. Once the time-slice of a running process is over, the Linux scheduler picks up another appropriate process from the run queue and allocates CPU power to that process.

A process also voluntarily can relinquish the CPU. The schedule() function could be used by a process to indicate voluntarily to the scheduler that it can schedule some other process on the processor.

At times, processes want to wait until a certain event occurs, such as a device to initialize, I/O to complete or a timer to expire. In such a case, the process is said to sleep on that event.

Wait Queues: (Thundering Herd Problem)

They are needed when more than one process wants to sleep on the occurrence of one or more than one event.

A wait queue for an event is a list of nodes. Each node points to a process waiting for that event. An individual node in this list is called a wait queue entry. Processes that want to sleep while the event occurs add themselves to this list before going to sleep. On the occurrence of the event, one or more processes on the list are woken up. Upon waking up, the processes remove themselves from the list.

Time-Bound Sleep:

You frequently may want to delay the execution of your process for a given amount of time. It may be required to allow the hardware to catch up or to carry out an activity after specified time intervals, such as polling a device, flushing data to disk or retransmitting a network request. This can be achieved by the function schedule\_timeout(timeout), a variant of schedule(). This function puts the process to sleep until timeout jiffies have elapsed. jiffies is a kernel variable that is incremented for every timer interrupt.

1. What happens after a user presses a key on the keyboard?

1) Keyboard controller interprets the san code from keyboard and send a hardware interrupt to the processor.

2) The processor invokes the interrupt handler

3) ISR reads the scan code and takes proper actions.

Scheduling:

1) What is the difference between process and thread context switch what all needs to be saved for process and thread context switch like do we need to save heap, basically she wanted me to list everything when I said registers she asked me to tell the name of the registers then she asked if process variables needs to be saved and so on?

i) Change virtual memory spaces

ii) processor's Translation Look-aside Buffer (TLB) or equivalent gets flushed making memory accesses much more expensive for a while. This does not happen during a thread switch.

Memory:

1) Draw 2-level paging diagram.

2) What is DMA? Can user level buffer / pointer used by kernel or drivers?