The lifecycle of a computer involves 5 steps.

Booting, Kernel Initialization, Device Management Initialization, full operation and shutdown.

For booting the operating system has a loader which is a program that moves bits from disk to memory and then transfers CPU control to those newly loaded bits. Then there is the bootloader which is a program that loads the “first program” (kernel)

The bootloader is kept in the Boot PROM, which is persistent code that is already loaded on power up. The boot manager is a program that lets you chose the first program to load.

The bootloader is the program that lets you load the first program. It is usually staged in a primary and secondary stage and it requires firmware support or “a hardware bootstrap” that is kept in firmware.

The BIOS differ from a PROM monitor in that the BIOS is only limited to be accessed only during the booting of the system. The monitor instead, can be continuously accessed by the command interpreter even if the system has already been loaded.

Boot Manager allow the user to pick the operating system he wants to run, and it basically copies itself in the (MBR) Master boot record which specifies the program that will run first during the startup of the operating system.

When booting a PC the first step is to do the POST check, then an interrupt 19h is generated which calls the bootstrap loader that allows you to select the device you want to boot from, it will then load the boot sector of whichever device you pick and in turn the device will allow you to load the bootloader of the operating system. The bootloader is kept in the MBR of your harddisk or in the first boot sector of your floppy disk. After the boot sector is loaded the operating system performs a magic number check that determines whether what has been loaded is a valid executable or not, and then allows the PC to execute the boot sector or (primary bootloader). A way to accelerate this process is to have the kernel compressed, in which then will consist of two pieces a head and a compressed tail, the head will be the code that will allow the tail to be decompressed.

Setup.S performs the real-mode hardware initialization, and lets you load more information from the kernel and execute as it is retrieved from harddisk.

Relations between processes are recorded via references that include the original parent, the parent, the youngest child, the oldest and youngest siblings.w/ this refs u can traverse all levels of related tasks.

The task structure is one of the main data structures that are kept in the operating system. It includes information about the state of the process, the flags and signals of the process. It also has information such as the address\_limit which determines the address space that is possible to access by your process. Exec\_domain is also kept as it provides a description of another platform that could be being emulated to execute the process.

Information for scheduling is also kept. Such as the priority which may be used for scheduling algorithms that are based on the priority of the task. Additionally a counter can be kept to count the number of clock cycles the process can run before scheduling it again. Also information of policy is kept to deal with multi-level queuing which allows multiple scheduling algorithms to be used in the operating system.

Information for memory management is also kept in the task structure, the struct for associated to this is the mem\_manage (struct \*) which describes the positions of all parts of the process in memory. This struct includes start\_code, end\_code, start\_data, and end\_data positions of the program. It also has a start\_stack for the stack ,and a start\_mmap for the heap, and an arg\_start, arg\_end, and env\_start and env\_end.

Information about files is also kept because the process can use them as resources. A files\_info pointer to the files informs structure is kept that contains a count of the number of processes that are reading/writing from a given file.

The task structure also keeps track of the per\_cpu\_utime and per\_cpu\_stime, which is the time that a process has taken to execute in user mode and kernel mode in each processor respectively.

Finally semaphore information is also kept, a semsleep reference is kept in order to determine what resources the process is sleeping on. A semhold reference is kept in order to determine what semaphores need to be released after the process finishes using the resource.

The Process table is the data structure that keeps tasks within the system organized in a doubly-linked list. That way, the tasks are moved between different lists that have different meanings/states. The tasks can be accessed via the next\_task and prev\_task in the task structure. A reference init\_task is kept in order to access the first task in the list. One can also access the current task using the get\_current macro, and the current task can be in any state.

The maximum number of tasks in the system is restricted to a global value: max\_threads. This helps maintain some aspects of security in the system.

For files two data structures are kept, one is the inode data structure and the other is the file data structure. The inode data structure stores basic information about a regular file, directory, or other file system object. It is basically the view of the “system” on the file. This means there is exactly one inode per file used. The information kept in this structure is primarily the physical location, meaning which blocks are comprised in the file, and what type of file it is. The file data structure on the other hand is the persepective of a “process” on the file. This structure keeps information which includes the usage mode, the current position of the file; etc… The file to the process is considered to be a continuous sequence of data bytes. However when looked at from the inode perspective it is not contiguous but the blocks for a given file maybe scattered all over the disk.

Memory is managed on a page basis, where each page is of size 2^n bytes.

To request a free page the alloc\_pages or get\_free\_pages is used.

To free acquired pages one should use the free\_pages call. It is also recommended to acquire a page of memory using the get\_zeroed\_page as it prevents unintended user data to be viewed by other processes when the process is finished.

All the wait queues in linux are cyclic and so they utilizae the same underlying queue structure which includes a

Struct list head

{

Struct list\_head \*next, \*prev

}

Processes enter queue blocked on an event when they are busy. There are different queues for every event in which a process may require waiting. Examples of events could be:

Sleep\_on – in which the process waits indefinitely.

Sleep\_on\_timeout in which the process will wait up to a given time, or it will just give up on waiting on a resource.

Interruptible\_sleep\_on will make the process wait indefinitely but the process may be interrupted when the resource is available.

Uninterruptable\_sleep\_on\_timeout – the process still waits up to a certain period, but otherwise it may not be interrupted.

Wait queues change the processes state and then enter the queue, then processes can exit the queue via functions such as:

Wake\_up – which removes the process from the queue, uninterruptable until completed.

Wake\_up\_interruptible, which makes the rmoving process from the queue interruptible as soon as possible.

Kernel level semaphores, and not user level semaphores. They are just like the regular semaphores except they work within the kernel.

Signals are basically used to inform processes about certain events, such as synchronization, abort or simply a change of state. They are sent typically through a function called send\_sig\_info. They contain a signal number, information about the signal which may include the sender, and the task to which the signal is being sent to. The kernel controls what process can send signals to what processes. If the signal is valid, then it is passed the task through the task structure (pending & sigpending).

Though the task could block the signal through the task structure blocked.

Signals are not processed right away. They are dealt with when the process is moved to the run state. When the scheduler moves the process into the run state and before returning it to user mode, the routine return from sys call checks for signals and if so calls do\_signal() to perform the action. The signal handler may also be changed to be a user defined function to change what to do when given a given signal.

There are two types of interrupts, there are interrupts in which the system halts all aspects and the interrupt is serviced right away. While the system services the interrupt it is uninterruptible.

The second technique involves splitting the interrupt into two stages:

Firs there is a hardware interrupt in which any immediate actions necessary are service right away without interruption, these involve receiving the interrupt and queuing it up for processing. Second stage is the software interrupts or stage 2 of the interrupt in which the processing of data of the interrupt is performed whenever possible, this process is also interruptible.

Why are there two techniques, well the first technique is good for small kernels but the second one is better for hard real-time system which allows better prioritizing when handling interrupts.

Timer interrupts. Internally we have the kernel that keeps track of time in two formats:

Ticks (long jiffies). Which are used for task scheduling?

Wall clock time (struct time) which is the time that is given to the user when he/she asks for the time.

Do\_timer is the interrupt routing for time.

In the first stage it updates the jiffies necessary for scheduling, and in stage two it updates xtime which is for the user in case he asks for the time. This interrupt routine do\_timer will update the jiffies first and then xtime, whenever a timer interrupt is generated.

The schedule routine performs several key operations that try to use the kernel time effectively. During this time the following operations are carried out:

1. Information for the current process and profiling information (if profiling) is kept.
2. pending stage 2 interrupts are checked to be processed.
3. The next process to be scheduled is determined.
4. and the next process’ context is loaded making it the current one.

Spin locks are basically only used when more than one CPU need to access the same part of the kernel. Because you can't disable interrupts on all CPUs fast enough. You have to use a spin lock if another processor is already accessing that kernel structure. It is because you are in the kernel, so you can't use a waiting queue (because the kernel can't be put to sleep) and because you can't disable interrupts on all CPUs fast enough.

Spin locks are used whenever more than one processor needs to access the same kernel data structure. Because you can’t disable interrupts on all CPUs at the same time, and because you can’t use a waiting queue because that would put the kernel to sleep, and that can’t be done, then the only thing left is to use a spinlock in which you will have a processor spinning (testing and setting) until it can access the data structure.

All tasks are created from other tasks. Maybe from the system’s idle task which in turn is the parent of all tasks. A new task is created when a parent process invokes either the fork() or clone() system calls.

Clone() inherits more from the parent task than does fork. The system calls eventuall call do\_fork function which creates the new task.

Do fork():

Allocates a new task\_sctruct data structure for the new process , then it links the task\_struct to the process table, and creates a new kernel space stack for execution when inside the kernel.

Whenever a process is created a task\_struct data structure is created for it, then it is linked to the process table. A kernel space stack is also allocated for it, where it can do execution within the kernel. Fields from the parent’s task\_struct are then copied into the child’s task\_struct. Some fields from the child’s task\_struct are also modified, b/c they are specific to itself.

f/e a new process identifier, links to the tasks parent and siblings, the process specific timers such as the creation time, and time quantum.

Do fork also copies other data structures that are in parent and should be replicated for the child task. For example.

The file table and new file descriptor for each open file.

It creates a new user data segment and copies the data to it.

It copies signal and signal handling information from the parent.

It also copies virtual memory tables.

The child’s state is also changed to running.

Tasks can also be terminated, and there are several ways to do this. The first is by making the exit() system call. The second way is for it to be delivered a signal with signal handler disposition to die. If the signal handler is changed so that it does not terminate when receiving the die signal, then the kernel can force it to terminate.

Eventually termination work is done by do\_exit() which grabes a global kernel lock, and sets the task state to zombie so that it gets no CPU time. It notifies all the children for termination as well using a the signal SIGCHILD. It releases any resources allocated by the do\_fork() function such as open files, and then calls schedule().

The tasks are scheduled using the schedule function. The schedule functions behavior depends on the scheduling algorithm that is being used. In some cases it may use more than one algorithm if using multi-level queues.

Schedule does the following:

1. Information for the current process and profiling information (if profiling) is kept.
2. It checks for pending stage 2 interrupts that need to be processed, and processes them.
3. It chooses the next process to be scheduled based on the scheduling policty.
4. It loads the next process’s context and dispatches the task to run on the CPU until another interrupt occurs, and it is called again.

Schedule ()

It releases the global kernel lock,

Then it handles software interrupts that are in stage two

Then it grabs the current process and current CPU. Current process’s state is changed to ready or to what is appropriate for it in case it has to wait on a resource.

It grabs the next process to schedule then the switch\_to macro is used to perform the transfer (which saves the state of the old task, and loads the state of the new task.

System\_call ISR is the global system call handler.

It basically saves the registers, and restricts memory accesing to kernel space. It grabs the system call id and verifies that it is legit, then it checks whether the task is being traced or profiled in order to keep record of time. And invokes the system call with the parameters.

A global kernel lock would only let one thing happen to the kernel at once.

It could be a single process doing a system call.

Remember, anything that happens to the kernel is always done in kernel mode by kernel code.

System calls just describe the interface that exists between nuser progs and the kernel.

This interface hast to be flexible and powerful, and allow for easy development of applications. The interface also has to be clear and controlled. Necessary for \*any\* security support the kernel wants to provide.

System calls must define a transition from user mode to system mode; this transition can vary dramatically depending on the kernel design, in which a kernel thread may offer all services, or processor architecture – in which you have processor modes and addressing support.

There is a simple design of system calls in which the simplest approach is to provide interrupt “wrapping”, that is, each of your system calls starts by disabling interrupts and ends with enabling interrupts. This results in the system call taking control of the system w/o any possibility of preemption. With this design there is no support for kernel developer, as one must be careful to ensure all system calls deal with interrupts properly.

This approach works but it makes some vital assumptions, that disable interrupts will not cause app problems and that the kernel trusts the user.

The alternative approach consists in invoking all system calls through a common interface. The interface will perform the transition btwn system and user modes. A common interface is through an interrupt. In linux the interrupt 0x80 is used.

With this approach each system call has a registered number. When the system call is made, the user mode process will put the system call id into a register and push parameters into registers or onto the stack, and then generate the interrupt. The kernel ISR routine then takes over.

The ISR, or global system call handler will save these registers, and restrict memory accesing to kernel space. Then it’ll grab system call id and verify it is legit, chk whether the task is being traced or profiled, and then invoke the sys call with its params.

After performing a specific system call function the handler will place the result of system call in a register and perform system maintenance (scheduling of tasks, check for pending signals, etc).

Why not save time since we are in the kernel anyway. Finally it will perform the return from interrupt.

Sys call development.

Writing a system call is simple for the kernel developer. First one must write the system call function, and then one can place a system call id number and register the system call, and then add the function pointer and system call id to the system call table. Some limitations are that there’s a finite number of system calls, and there is a fixed number of parameters the user can have in the sys call. No access to shared libraries from the system call.

System call stubs are generated to aid the user application developer. They have to load registers w/ params & sys call id then generate the sys call interrupt.

The stub can be generated by kernel during its compilation.

IPC is used to coordinate and share information btwn tasks, resource sharing, and synchronization. The kernel manages system resources and access to these resources must be synchronized in many cases.

To provide synchronization the kernel controls who can run during a system call. Other process may be scheduled only in three different cases:

When the system call invokes the schedule method, when the system call invokes a suspend method that will suspend the process, or if preemptive scheduling is in force, when an interrupt occurs that affects scheduling. So the kernel provides synchronization internally by:

Only calling schedule when it won’t affect synchronization.

Same for suspending a function.

And finally turning off interrupts when performing critical regions. Simple but it slows down the system..

However in multiprocessor systems turning of interrupts does not work, because one cannot do this for all processors at the same time. So.. synchronization in these systems is performed using a spinlock. It is just like a regular mutex when unlocked contains a 1 and locked it contains a 0.

Sys calls try to acquire the lock by decrementing the value. Releasing the lock will increment the val. Spinlocks require processor test&set atomic operations.

Unlike the OS\_mutex/semaphore these locks have a busy loop.

Memory management forces tasks to communicate over fixed interfaces that guarantee security and protection from one another (and the kernel). Processes are tricked and made believe that they all have all the memory to themselves than it is actually available. Each process is provided its own separate task virtual address space where it is basically tricked to believe it has more memory than it is actually available. At the same time this ensures exclusive control of the allocated blocks. All memory management in the kernel is done on the basis of a page, for the i386 a page is 4K. In a virtual addressing space where the address is 32 bits wide there are 4Gb of virtual address space for a task, which ends up being around 2^20 pages per task.

This process virtual address space is divided into a user segment and a kernel segment. The user segment and kernel segment are further divided in code and data segments. The kernel segment contains data structures relevant to the operating system and system calls are executed within it. All processes see the same kernel segment, and this one constantly mapped to physical, so it is guaranteed pretty much that what is in the logical address space in the kernel segment, is also in the physical address space. Where code is the actual program executed, and the data segment contains the stack and heap for data structures that the code manipulates. Virtual addresses in this space are mapped to actual physical memory addresses by the Architecture Independent Memory Model (AIMM). Through the AIMM the kernel and hardware together guarantee that the virtual and physical addresses are correctly corresponding. The kernel does it by ensuring that the adds correspond to a phys addy, the hardware does it by ensuring that the mode is correct when addressing each of the segments. If the kernel is unmapped it means that we’re killing the process. The TASK\_SIZE struct allows u to know how much mem for the user proc.

In the kernel there are several macros defined to allow access to the user segment of a process via a virtual addressing space. The access\_ok() macro allows the kernel to query a virtual addy to see if it is legitimate for the task, and each of the kernel’s user memory access macros have a \_\_ equivalente that performs the action w/o th chk. Why, because it is faster.

Logical addresses are converted from virtual to physical in an AIMM in a three level process, the process is dictated by the hardware’s MMU. Each level of indirection is handled by a structure that points to the next level of indirection until the page table is reached. The page table is the lowest level of the memory model. Here the final mapping is done, so it addresses an actual page in the physical memory. The page table entry has flags which indicate information about the status of the page, such as whether it is associated to a physical memory page, whether it is shared, read-only, accessible only by the kernel, or kernel read-only access, dirty, or executable.

Swap-space vs Ram

The AIMM handles the case when a particular logical address corresponds to a physical address that is at the moment in swap-space. Swap-space can be implemented either as a swap file, or a swap partition. A swap partition is more efficient than a swap file, but a swap file is more flexible, and is easier to allocate after initial setup. To handle such case it uses an AVL tree or linked list to find the vm\_area struct that corresponds it and then using the nopage function with the area the address, and the write\_access, it obtains the physical page, copies the data, and does the necessary corrections so that the logical address points to the corresponding pages.

Because shared libraries and applications are sometimes large, they may fit in the addressable space, but it is wasteful to load. So virtual memory provides a mechanism for sharing memory , which makes it quicker to write to a mem location than to perform a system call.

Read Only data can also be shared btwn tasks

Vm\_opps is a structure of fuction pointers where the it points to different operations depending on whether it is a file or a device.

The swap space may be stored as a swap file or a swap partition. Swap partition is faster. Checks the vmarea struct, it realizes that the memory has been swapped out, and calls nopages, then the address of vm\_area, and gives it the access. It goes and asks for a physical page, and that can put in the AIMM. Once it gets the physical page, it copies the page from the virtual memory area, into the physical memory page, and adds it to the AIMM. It must do changes to the page table.

User dynamic memory management is managed through pages, while kernel’s dynamic memory management in the kernel is done in a slab basis. It is in some ways easier than the user segment, because it has no shared libraries, all the memory is always kept in physical pages.