Responsibilities of Operating System

Process Management:

• Scheduling processes and threads on the CPUs

• Creating and deleting both user and system processes

• Suspending and resuming processes

• Providing mechanisms for process synchronization

• Providing mechanisms for process communication

Memory Management:

• Keeping track of which parts of memory are currently being used and who is using them.

• Deciding which processes (or parts of processes) and data to move into and out of memory.

• Allocating and deallocating memory space as needed.

Storage Management:

• Creating and deleting files

• Creating and deleting directories to organize files

• Supporting primitives for manipulating files and directories

• Mapping files onto secondary storage

• Backing up files on stable (nonvolatile) storage media

Disk Management:

• Free-space management

• Storage allocation

• Disk scheduling

What fork() does and what exec(0) does

Message Passing vs Shared memoey

dtrace

lsmod

**How a process is loaded in memory?**

A typical life for a process is..

> **Created:** the fork system call has been used and memory area is allocated for it.

> **Running:**

> Execve is called by shell which creates a stack and push argc, argv[] which are passed to it and calls \_start method.

\_start function loads data in registers, sets up instruction pointer to first instruction in \_libc\_start\_main

int \_\_libc\_start\_main( int (\*main) (int, char \* \*, char \* \*),

int argc, char \* \* ubp\_av,

void (\*init) (void), // \_\_libc\_csu\_init, Constructor of this program.

called by \_\_libc\_start\_main before main.

void (\*fini) (void), // \_\_libc\_csu\_fini - Destructor of this program.

// registered by \_\_libc\_start\_main with \_\_cxat\_exit().

void (\*rtld\_fini) (void), // Destructor of dynamic linker \_\_cxat\_exit()

void (\* stack\_end));

So we expect \_start to push those arguments on the stack in reverse order before the call to \_\_libc\_start\_main.

Ok, the loader handed control to \_start, who called \_\_libc\_start\_main who called \_\_libc\_csu\_init who now calls \_init.

After this main is called. Now the control switches between user and kernel mode, probably several times..)

> **Sleeping**, waking up, sleeping, waking up, ...

> **Exiting** (final switch to kernel mode, zombie state, disappearance).

**Process Control Block**

Each process in operating system is represented by a process control block which is stored in protected mode in kernel stack area.

It contains the following information:

* **Process State**: New, Ready, Running, Waiting, Terminated
* **Program Counter**: Stores the address of next instruction.
* **CPU registers**: Accumulators, General purpose, Index registers and stack pointers
* **CPU scheduling information**: Process priority, pointers to scheduling queues
* **Memory management information**: Value of base/limit registers, page/segment tables
* **Accounting information**: CPU time used and time limits etc
* **I/O status information:** It includes I/O devices allocated to the process.

Whenever a process is interrupted its state is saved in PCB which is reloaded when process is resumed.

In a process which contains multiple threads, information for each thread is stored in PCB.

PCB is generally of size 1.7Kb/process

Within kernel, processes are represented by doubly linked list of *task\_struct.*

**Context Switch**

When CPU encounters and interrupt, it stops the process under execution and based on interrupt executes interrupt service routine by checking out entry in interrupt vector table.

After pid = fork(), the pid of child is 0 while that of parent is >0, while <0 means an error.

The parent process can wait for the child process to terminate by invoking the wait() system call. The PCB of child is not released unless and until parent calls wait() system call.

**Interrupts:**

These are of 3 types:

Software interrupts, Hardware interrupts and Processor exceptions.

Interrupt descriptor table consists of 256 entries, of which 32 are reserved for process exceptions.

The IVT always resides at the same location in memory, ranging from 0x0000 to 0x03ff, and consists of 256 four-byte real mode far pointers (256 × 4 = 1024 bytes of memory).

Interrupt vs Exception

> Interrupt includes hardware interrupt and software interrupt; Exception includes trap, fault, and abort.

> Hardware interrupt is asynchronous, and others (exceptions and software interrupt) are synchronous to instructions.

> Software interrupt are used to implement system call. In intel processor, we use INT instruction. In MIPS we use syscall.

First entry of IVT is divide-by-0

**Process Synchronization**

A situation when multiple processes access the same data concurrently and the outcome of the operation depends upon the sequence in which statements were executed is called race condition.

The block where concurrent data modifications are being made is called critical section. To solve this issue, a solution must satisfy the following 3 conditions:

* Mutual Exclusion – If one process is executing its critical section, then no other process is allowed to execute the same.
* Progress – If there is no process executing in critical section then only those processes which are not executing in remainder section are allowed to enter critical section and this decision cannot be postponed.
* Bounded Waiting - There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections.

**Deadlock**:

When 2 processes are waiting indefinitely for an event that can only be caused by one of those 2 processes, the processes are said to be in deadlocked state.

A deadlocked system satisfies below 4 criteria:

Mutual Exclusion – At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.

Hold and Wait – The process is holding one resource and waiting for another to acquire.

Circular Wait – The dependency of processes w.r.t resources can be depicted in the form of circle.

No Preemption: A resource cannot be taken from a process unless the other process releases the resource.

**Prevention** – Eliminate one of the criteria mentioned above.

Prevent Mutual Exclusion: Make the resource shareable if possible such as read only files.

Prevent hold and wait: If a process is asking for resource, deallocate all resources allocated to it.

Prevent circular wait: Banker’s algorithm.

No Pre-emption: Pre-empt the resources if some high priority process needs resource.

Other ways:

Don’t start a process until it is allocated all resources or

Banker’s Algorithm

1. Max need of resources by each process.

2. Currently allocated resources by each process.

3. Max free available resources in the system.

Request will only be granted under below condition.

1. If request made by process is less than equal to max need to that process.

2. If request made by process is less than equal to freely available resource in the system.

Ostrich Algorithm:

Just ignore the problem. It is used when it is more cost-effective to allow the problem to occur than to attempt its prevention. For example, if each PC deadlocks once per 10 years, the one reboot may be less painful than the restrictions needed to prevent it

TLS

Thread local storage refers to the variables which are declared static/global but each thread maintains its own copy of that variable. A good example is **errno**. If a thread executes a system call and an error occurs, in that case it might happen that another thread can override that variable so in that case it makes sense to keep a local copy for the thread. In C++11 this type of TLS (thread local storage) is now a storage class now which can be declared by prepended the variable type with thread\_local

Synchronization

Semaphores should be used in conditions like producer/consumer while condition variables in conditions where one thread is waiting for some condition to become true.

Check if semaphore also allows broadcasting just like condition variable

Synchronization allows you to control program flow and access to shared data for concurrently executing threads.

The four synchronization models are mutex locks, read/write locks, condition variables, and semaphores.

•Mutex locks allow only one thread at a time to execute a specific section of code, or to access specific data.

•Read/write locks permit concurrent reads and exclusive write to a protected shared resource. To modify a resource, a thread must first acquire the exclusive write lock. An exclusive write lock is not permitted until all read locks have been released. (Check in present code). One implementation could be to maintain a boolean variable that will maintain state and tell whether the resource access is locked or not.

•Condition variables block threads until a particular condition is true.

•Counting semaphores typically coordinate access to resources. The count is the limit on how many threads can have access to a semaphore. When the count is reached, the semaphore blocks.

\* Thread can create a process

Like process if one thread is blocked another one can run

Thread do not need IPC

Context switching are fast when working with thread

If a thread makes an exec() call, the new process overlays the current memory and current process is suspended.

A thread can explicitly call pthread\_exit() while another thread can kill it using pthread\_cancel()

Thread IDs are guaranteed to be unique only within a process.

Default attributes of a thread:

• It is unbounded

• It is non-detached

• It has a default stack and stack size

• It inherits the parent's priority

• A thread can be set whether it is cancellable or not and whether its cancellation is synchronized or instantaneous

Following is simple example of thread creation and invocation:

pthread\_t thread\_id;

pthread\_create(&thread\_id, NULL, myThreadFun, NULL);

pthread\_join(thread\_id, NULL);

Thread can get its id by calling pthread\_self();

To compare 2 thread id's int pthread\_equal(pthread\_t tid1, pthread\_t tid2);

pthread\_setschedparam() to modify the priority of an existing thread

int pthread\_setschedparam(pthread\_t tid, int policy,

const struct sched\_param \*param);

Sample code to set priority of thread :

#include <pthread.h>

pthread\_t tid;

int ret;

struct sched\_param param;

int priority;

/\* sched\_priority will be the priority of the thread \*/

sched\_param.sched\_priority = priority;

/\* only supported policy, others will result in ENOTSUP \*/

policy = SCHED\_OTHER;

/\* scheduling parameters of target thread \*/

ret = pthread\_setschedparam(tid, policy, &param);

pthread\_once\_t once\_control = PTHREAD\_ONCE\_INIT;

pthread\_once(once\_control, init\_routine)

pthread\_mutex\_trylock() can be employed to prevent deadlocks in the program

int pthread\_getschedparam(pthread\_t tid, int policy, struct schedparam \*param) gets the priority of the existing thread.

sched\_yield(); is called by thread to yield its execution in favor of another thread

**Thread Cancelation:**

A thread can set its cancel state as

int pthread\_setcancelstate(int state, int \*oldstate);

( type can be either of (PTHREAD\_CANCEL\_DISABLE/ENABLE, NULL )

int pthread\_setcanceltype(int type, int \*oldtype);

type can be either of : (PTHREAD\_CANCEL\_DEFERRED/ASYNCHRONOUS, NULL )

Finally, pthread can be canceled using pthread\_cancel() API which is implemented using signals (which are fired in process queue) and can result into memory leaks. So, to prevent that, please create routine to flush the threads by pushing cleanup routines which are popped in the reverse order.

Major steps took place when a pthread\_cancel(thread\_id) call is made:

1. Cancellation clean-up handlers are popped (in the reverse of the order in which they were pushed) and called. (See pthread\_cleanup\_push(3).)

A cleanup may be required if we want to release resources held up by thread like mutexes and other handles. These are not called when a thread makes a normal return even without calling pthread\_exit()

2. Thread-specific data destructors are called, in an unspecified order. (See pthread\_key\_create(3).)

3. The thread is terminated. (See pthread\_exit(3).)

pthread\_attr\_\* (&attr, <>); specifies a large number of API's o set various attributes of the thread

pthread\_attr.setscope(&attr, PTHREAD\_SCOPE\_SYSTEM); //bounded

pthread\_attr.setschedpolicy(&attr, SCHED\_OTHER);

pthread\_attr.setstacksize(&attr, stack\_size\_int);

pthread\_attr.setdetachstate(&attr, PTHRED\_CREATE\_DETACHED ); or PTHREAD\_CREATE\_JOINABLE

// attr structure can be destroyed using following API

pthread\_attr\_destroy(&attr);

Notes:

1. Multiple threads cannot wait for a thread to terminate
2. you can detach a thread by calling pthread\_detach(thread\_id)
3. pthread\_join(pid, &status) is the only way to know whether thread was cancelled successfully or not
4. Disadvantages of multithreading:
5. Deadlocks
6. Race Conditions
7. Difficulty in debugging

**Concurrency**

Mutex and Semaphores, both, are synchronization primitives. While mutex is based on locking-unlocking mechanism, semaphore is based on signalling mechanism.

Differences:

>Semaphores can provide sync. Services access to multiple resources while mutex only one

>Mutex is unlocked by the process that locked it while semaphore can be signaled by any other thread.

> Mutex implementation underneath is provided by hardware atomic instructions compare-and-swap idiom. The thread first checks the global mutex value ( a variable created for a given mutex ) if that is ‘0’ it makes it ‘1’ and thread can proceed else thread will be blocked, put in scheduling queue and wait for mutex flag to be turned ‘0’ again.

> Cond-Wait mechanism works this way only. threads are made to wait in a kernel scheduling queue for a given mutex (which is assigned a value)

pthread\_key\_t key;

pthread\_key\_init(&key)

pthread\_setspecific(key, const void \*);

void \*buffer\_ptr = pthread\_getspecific(&key);

Synchronization Objects

**Mutexes**

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;//combines bottom 2

pthread\_mutexattr\_t mattr;

pthread\_mutex\_init ( &mutex, &mattr );

pthread\_mutex\_lock(&mutex);

pthread\_mutex\_unlock(&mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mp); //non-blocking but if successful it locks the mutex

Mutex Attributes:

int pthread\_mutexattr\_gettype(const pthread\_mutexattr\_t \*attr, int \*type);

int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int type);

type is PTHREAD\_MUTEX\_DEFAULT/NORMAL/RECURSIVE

pthread\_mutexattr\_init(&mattr);

pthread\_mutexattr\_destroy();

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*mattr, int pshared); // mattr scope is either PTHREAD\_PROCESS\_PRIVATE or PTHREAD\_PROCESS\_SYSTEM

pthread\_mutexattr\_getpshared(pthread\_mutexattr\_t \*mattr, int \*pshared);

pthread\_mutex\_destroy(&mutex);

pthread\_kill() will not kill a thread, it just passes a signal to it. If SIGTERM is passed process is killed

Semaphore:

sem init() function for creating and initializing an unnamed semaphore:

#include <semaphore.h>

sem t sem;

/\* Create the semaphore and initialize it to 1 \*/

sem\_init(&sem, 0, 1);

The sem init() function is passed three parameters:

1. A pointer to the semaphore

2. A flag indicating the level of sharing

3. The semaphore’s initial value

Short Notes :

> Processes can run on different machines while threads cannot.

> Each thread corresponds to a function like main is a function, rest all other threads are dependent on the thread object...

> To prevent a deadlock acquire a mutex in same given order like if 1 starts and Ask for mx1 -> mx2 if 2nd process starts it will try to acquire 2nd which is also claimed by 1st process. That will be a rude idea

If you intend to join a thread and there is a piece of code that can throw exception...then you should put t.join() call in that catch block too.

**Inter-process Communication**

Two modes:

* Shared memory
* Message Passing

Shared memory is fast because message passing involves more system calls. In case of shared memory, system calls are required just to setup shared memory regions, after that there are just normal memory accesses.

The producer-consumer problem can be solved using either of them. In this, producer will first call shm\_open(), ftruncate() and mmap() to map the file descriptor returned by shm\_open() to its address space. mmap() then returns the pointer to that memory area.

Consumer also do shm\_open() in read-only mode and call mmap() to map the shared segment in its address space and finally calls shm\_unlink() to remove the shared object.

Producer:

shm fd=shm open("SHARED\_OBJECT\_NAME",O CREAT|O RDRW,0666);

ftruncate(shm fd,SIZE);

ptr=mmap(0,SIZE,PROT WRITE,MAP SHARED,shm fd,0);

sprintf(ptr,"%s","Hello World");

Consumer:

shm fd=shm open("SHARED\_OBJECT\_NAME",O RDONLY,0666);

ptr = mmap(0,SIZE,PROT READ,MAP SHARED,shm fd,0);

printf("%s",(char\*)ptr);

**Sockets**

Processes across systems can employ Sockets to implements Client-Server architecture.

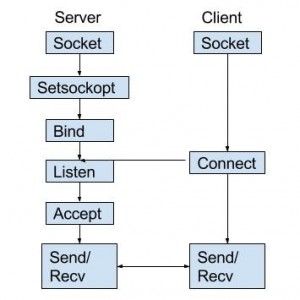
A socket is identified by the combination of IP address and port. When a client requests to initiate a connection, it is assigned a port by its host computer. The packets travelling between the machines are delivered to the appropriate process based on the port number.

Sockets allow unstructured stream of data to be shared between the machines.

A socket is identified by:

**Client IP : Client Port** and **Server IP : Server Port**

To communicate with the client the server opens a new port on its side and waits for the connection. As soon as it gets the connection (via client’s connect() API), it creates a new socket combining the address of client and its own address.



**Stages for server**

**Socket creation:**

int sockfd = socket(domain, type, protocol)

*sockfd*: socket descriptor, an integer (like a file-handle)

*domain*: integer, communication domain e.g., AF\_INET (IPv4 protocol) , AF\_INET6 (IPv6 protocol)

*type*: communication type

SOCK\_STREAM: TCP(reliable, connection oriented)

SOCK\_DGRAM: UDP(unreliable, connectionless)

*protocol*: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

**Setsockopt:**

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

**Bind:**

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR\_ANY to specify the IP address. While using UDP, this function call can be dropped.

**Listen:**

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

**Accept:**

int new\_socket= accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

**Stages for Client**

Socket connection: Exactly same as that of server’s socket creation

Connect:

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server’s address and port is specified in addr.

Top handle multiple clients, it’s better to use select() API

Select command allows to monitor multiple file descriptors, waiting until one of the file descriptors become active. For example, if there is some data to be read on one of the sockets select will provide that information. Select works like an interrupt handler, which gets activated as soon as any file descriptor sends any data.

Data structure used for select: fd\_set

It contains the list of file descriptors to monitor for some activity.

There are four functions associated with fd\_set:

fd\_set readfds;

// Clear an fd\_set

FD\_ZERO(&readfds);

// Add a descriptor to an fd\_set

FD\_SET(master\_sock, &readfds);

// Remove a descriptor from an fd\_set

FD\_CLR(master\_sock, &readfds);

//If something happened on the master socket , then its an incoming connection

FD\_ISSET(master\_sock, &readfds);

**Activating select:** Please read the man page for select to check all the arguments for select command.

activity = select( max\_fd + 1 , &readfds , NULL , NULL , NULL);

TCP 3 way handshake takes place in the listen function - Here handshake packets are sent in terms of Sync/Sync-Ack/Ack

In these handshakes there is a sequence number that tells whether the packets being sent are in sync or not.

Steps are :

Handshake : Automated process of negotiation that takes place between 2 nodes It dynamically sets the params required for the transmission

Some params are : tranfer rate, coding parity, alphabets,

3-way

A sends the seq number to B (x) : SYNC

B sends its own seq num. (y) and x+1 : SYNC-ACK

A sends the (y+1) : ACK

which is accepted by B and is not responded back

However HTTP can use unreliable protocols such as

The Simple Service Discovery Protocol (SSDP) is a network protocol based on the Internet Protocol Suite for advertisement and discovery of network services and presence information. the User Datagram Protocol (UDP), for example in Simple Service Discovery Protocol listen on UDP and TCP on the same port

UNIX-domain sockets are generally more flexible than named pipes. Some of their advantages are:

* You can use them for more than two processes communicating (eg. a server process with potentially multiple client processes connecting);
* They are bidirectional;
* They support passing kernel-verified UID / GID credentials between processes;
* They support passing file descriptors between processes;
* They support packet and sequenced packet modes.

To use many of these features, you need to use the send() / recv() family of system calls rather than write() / read().

Pipes:

Pipes act as an conduit between the 2 communicating processes. They typically provides one of the simpler ways to communicate. Ordinary pipes are unidirectional.

Pipes are created using pipe(int fd[]) function call which creates 2 pipes one for reading(fd[0]) purpose and another for writing purpose(fd[1]). These pipes can be accessed using regular read() write() system calls. Pipes are used for communication between parent and child.

There are two types of pipe: Named and Unnamed.

For Unnamed pipes, parent-child relationship is required and pipes are ceased to exist once processes are terminated. While for the named pipes, this condition doesn’t hold since it doesn’t need any relationship and it is bidirectional too. Named pipes are referred as FIFO’s in UNIX.

Thread

A thread is basic unit of CPU utilization. It comprises of a stack pointer, program counter, register set and stack. It shares with other threads executing in same process the code segment, data segment, open files and signals.

Advantages of threads

* Responsiveness
* No special setup required sharing of data as in processes.
* Overhead of creation is less than fork() and too s context switch

Asynchronous threading – Parent and child independent

Synchronous threading – Parent waits for child to complete. It is helpful especially in cases when parent needs to sum up the results of children.

pthread\_t pid;

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_create(&id, &attr, <function>, <data>);

pthread\_join(pid, NULL);

Thread pools is a good strategy to provide multiprogramming. In this, we maintain the active set of threads in a pool. When a task arrives, it is immediately assigned a thread. Advantages of thread pools :

* Servicing a request with already created thread is faster than creating a new thread.
* A thread pool limits the number of threads that can exist.

pthread\_cancel(thread\_id) cancels a thread named by thread\_id. It can be used in situations when multiple threads are querying from the database. When thread returns the result, other should stop performing operation to which we should sent cancel signal.

Pthreads names these operations sem wait() and sem post(),

respectively. The following code sample illustrates protecting a critical section

using the semaphore created above:

/\* acquire the semaphore \*/

sem wait(&sem);

/\* critical section \*/

/\* release the semaphore \*/

sem post(&sem);

**OS Concepts**

**Nice Value**;

CPU nice values means scheduling priority. Processes with +ve nice values have less priority (it ranges from -19 to +19 )

**About System calls**

Once the wrapper has done its initial work it’s time to jump into hyperspace the kernel. The mechanics of this transition vary by processor architecture. In Intel processors, arguments and the syscall number are loaded into registers, then an instruction is executed to put the CPU in privileged mode and immediately transfer control to a global syscall entry point within the kernel. The kernel then uses the syscall number as an index into sys\_call\_table, an array of function pointers to each syscall implementation like this

[0] = sys\_read,

[1] = sys\_write,

Following function in entry.S is called

call \*sys\_call\_table(,%rax,8) # XXX: rip relative

where system call number is stored in %rax register

And this gives us all we need to join the dots from user space to the kernel code. The standard ABI for how x86\_64 user programs invoke a system call is to put the system call number (0 for read) into the RAX register, and the other parameters into specific registers (RDI, RSI, RDX for the first 3 parameters), then issue the SYSCALL instruction. This instruction causes the processor to transition to ring 0 and invoke the code referenced by the MSR\_LSTAR model-specific register — namely system\_call. The system\_call code pushes the registers onto the kernel stack, and calls the function pointer at entry RAX in the sys\_call\_table table — namely sys\_read(), which is a thin, asmlinkage wrapper for the real implementation in SYSC\_read().

> All regular file I/O takes place through page cache, kernel loads files in the form of 4Kb chunks, even if you read 1 byte , 4 KB will be loaded

Virtual Memory

It is a large address space that is available to the processes running on computer which consists of both physical memory and secondary memory.

Page table

It is basically mappping between virtual addresses as seen by process to real memory which is in the form of pages it also contains a bit that tell if a page is in memory or needs to be fetched from memory.When paging and page stealing are used, a problem called "thrashing" can occur, in which the computer spends an unsuitably large amount of time transferring pages to and from a backing store, hence slowing down useful work. Thrashing occurs when there is insufficient memory available to store the working sets of all active programs

Each process has its page table

The virtual address generated by a process has offset + virtual page frame number

VPFN is translated into the virtual address and offset is added to it to go to that instruction

The set of pages that a process is currently using is called the working set

The Linux kernel is linked to run in physical address space.

Block devices are only ever accessed via the buffer cache

Similar to software cache there is hardware cache which is called TLB that contain frequently accessed Page table entries. A corrupted cache may bring down the whole system

free\_area is a vector array with each entry represnting the queue of free blocks of size 2^entry. It means linked list at entry 2 will have nodes denoting blocks which are free and are of size 2^2

Each free store also has a map that contains the block numbers of the allocated array of that given size

When an executable is introduced it is mapped to virtual address space and not directly taken into the physical memory and the former process is called as memory mapping

A complete process address space is denoted by memory descriptor which contains all the information related to the process address space. The memory descriptor is represented by struct mm\_struct {}

A mm\_struct{} is basically structure that denotes a memory segment it further contains list of vm\_area\_struct {}

rss ( resident set size , number of allocated pages )

total\_vm ( Total pages )

locked\_vm ( Locked pages ), memory area semaphore

mm\_users ( Number of processes using this adress space ),

map\_count ( Number of memory areas )

first and last addresses of stack , code , data and heap.

mm\_count ( Primary reference count of usage )

struct vm\_area\_struct \*mmap; /\* list of memory areas \*/

When fork() is called copy\_mm() is executed that copies parent's memory descriptor

Linux kernel doesn't differentiates between the processes and threads

kernel threads do not have any pages in user-space

When an executable is introduced in the virtual memory a vm\_area\_struct {} structure is filled for that particular area, hence treating that area as a memory object. This structure holds attributes that are applicable to the whole memory area.

**Process Scheduling**

Initially there was O(1) scheduler which basically used 2 queues, one active and another expired.

Active queues contains 140 entries (in decreasing priorities) of pointers each pointing to doubly linked list. Scheduler allocates processes entries 1 by 1. After that it switches the queues, that is why it is called O (1) scheduler because it swaps queues.

Completely fair scheduler is the current scheduler implemented inside the kernel which is implemented in the form of RB-tree. Each node of RB tree is a task and each task is associated with **vruntime** Every time scheduler runs it picks the leftmost element from the binary tree.

CFS basically assigns processes a proportion of CPU which is affected by nice value of that process

Processes which requires interactivity are given predence. The two determining factors are timeslice and priority of the process

NI is the nice value, which is a user-space concept. PR is the process's actual priority, as viewed by the Linux kernel.

SCHED\_OTHER is default scheduling behaviour, SCHED\_BATCH indicates to CPU that the given task is CPU-intensive so more penalty will be added to processes in case such process are pre-rmpted so these are favored in case process is CPU bound and we don't want to change its nice value.

There is sched\_entity which keeps the data regarding the scheduling of the process and it is embedded inside the task\_struct

It includes – load weight, timeslice consumed and exec start + total\_exec\_time + vruntime (measured in nanoseconds)

Whenever a task in created or process in fork()'ed it is stored in the rb tree of tasks

When a task is going to sleep it removes itself from RB tree and put itself in wait queue and calls schedule() which then picks the next task from RB tree. The task can relinquish the Cpu by calling sched\_yield(). If that process is the only process in RB tree, that process continues execution.

Task state can be *TASK\_INTERRUPTIBLE, TASK\_UNINTERRUPTIBLE, TASK\_RUNNING*

**Context Switching**

Process context is the mode of operation the kernel is in while it is executing on behalf of a

process—for example, executing a system call or running a kernel thread.

Interrupt context mean that there will not be scheduling taking place untill

and unless that task ends while in the process context every task gets the

CPU time

Interrupt handlers did not receive their own stacks. Instead, they would share the stack of the

process that they interrupted. 1 The kernel stack is two pages in size; typically, that is 8KB

on 32-bit architectures and 16KB on 64-bit architectures

It is handled by context\_switch() function which is called by schedule() function when a new task is about to run

it calls switch\_mm( ) // that swaps the vm\_memory\_area

it calls switch\_to() // that swap the processor state

thread\_info contains a variable called preemp\_count that is incremented whenever a resource is acquired and decremented whenever a resource is released. Task is preemptive only when value of this count is 0

**Pre-emptive Kernel**

It means that process currently running in kernel mode can be pre-empted by another process which is also running in kernel mode. Earlier, it wasn't possible which lead to priority inversion in which the lower priority process continues to make system calls and bars the higher priority process to take control of CPU. Whenever a process/task calls schedule() (voluntarily or when it blocks) means it simply wants to relinquish the control

Real time priority ranges from 0 – 99. By default, this means the –20 to +19 nice

range maps directly onto the priority space from 100 to 139

*Process Affinity* : BitMask to make the process run on 1 or various processors

example of such type of system call is sched\_setaffinity()

**Syscalls**

The only way a user process can communicate with kernel, user application generally calls C library functions which in turns call system calls. The pid is a member of task\_struct .

When a sys call is made kernel stores the processor / registers state on stack. So that when user mode is just going to return it restores the state. The kernel keeps a list of all registered system calls in the system call table, stored in

sys\_call\_table

Example of getting processId:

asmlinkage long sys\_getpid(void)

**Interrupts**

Whenever any of the following event occurs, control starts working in kernel mode  
> Exception

> Interrupt

> System Call

In the case of system call system call handler is invoked

The defined software interrupt on

x86 is interrupt number 128

Hardware Interrupts are generated by hardware devices which are basically hadled by 8259 IRQ Controller Chip. The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ). Programmable Interrupt Controller (PIC) may be connected between the interrupting device and the processor's interrupt pin to multiplex several sources of interrupt onto the one or two CPU lines typically available. Any interrupt is received by Interrupt Vector Table . Example : **Keyboard** :keyboard Service Routine then generates a Two Byte code that it puts in the Keyboard Buffer area in RAM Memory, from 0041E hex to 0043D hex.

Register %eax is passed the system call number and the registers ebx , ecx , edx , esi , and

edi contain, in order, the first five arguments

In case system call in prepended with asmlinkage it means the kernel needs to find the parameters of the function is CPU stack rather than in the registers.

return value is written back to %eax register

While executing code the kernel reads and writes data to and from user level space for this there are 2 functions   
copy\_from\_user(&buf, src, len) and

copy\_to\_user(dst, &buf, len) //buf is local long type variable

both contains pointer to kernel space, pointer to user space and the size to be copied.

Exceptions ( Trap for a system call and divide-by-zero ) and page faults are synchronous interrupts.

Throughput and latency are reciprocal terms

Function to register interrupt

int request\_irq(unsigned int irq, //interrupt req. number

irq\_handler\_t **handler**, // function pointer to int. handler

unsigned long flags, // example IRQF\_SHARED

const char \*name, // name of device

void \*dev) //cookied used when interrupt handler has to be freed

see /proc/interrupts and /proc/irq for details

// proc refers to process information pseudo-file system

Function to free interrupt handler

void free\_irq(unsigned int irq, void \*dev)

Example

static irqreturn\_t **intr\_handler**(int irq, void \*dev);

//return values IRQ\_NONE or IRQ\_HANDLED

When an interrupt is received, all the handlers are called and whichever returns IRQ\_HANDLED services the interrupt

**Memory Management**

**Memory layout**

Stack -- High address

|

-

^

|

Heap

Uninitialized section

Initialized section (read-only and read-write)

Code section -- Low Address

To learn memory layout remember sorted order from low-high (CIU )

Whole virtual memory is divided into chunk of non-contiguous memory areas which are also called vm\_areas. Each memory area has permission attached to it RWX. If a process access memory area of another process, it results in segm fault. Memory areas are accessible by memory map. Each map entry displays the library/text segment. [ anon ] areas designates the heap memory.

The Kernel represents each process address in the form of memory descriptor which contains pointer to memory areas. ( in form of vm\_area\_struct ) , start and final address of stack, heap. Kernel threads do not occupy process address space so their mm field is NULL.

Each memory area has address to start and last memory

Typical page structure

struct page {

unsigned long flags; // Whether page is dirty

atomic\_t \_count; // number of references

atomic\_t \_mapcount; //

unsigned long private;

struct address\_space \*mapping;

pgoff\_t index;

struct list\_head lru;

void \*virtual;

};

**Little and Big Endian Mystery:**

In big endian the MSB stored first

in little endian MSB (byte) stored last

**To retain - L^3 – LSB stored in lower address for Little Endian**

unsigned int i = 1;

char \*c = (char\*)&i;

if (\*c)

printf("Little endian");

else

printf("Big endian")

**Priority Inversion and Its Solution:**

Consider 3 tasks H, M and L with following priorities: High, Medium and Low respectively. Consider that H and L share a resource and L is currently executing its critical section. Now, M being a higher priority process will try to pre-empt ‘L’ which will result in priority inversion because now ‘H’ has to wait for ‘M’ to complete when both ‘H’ and ‘M’ were in ready state. To counter this problem, the lower priority task’s priority is elevated to the priority of ‘H’ so that ‘M’ cannot preempt it and ‘H’ can resume after ‘L’ is completed. This solution is called Priority Inheritance.

**DHCP**

Client with its mac send the request to router or whosoever assigns the ip address, send DHCP discover packet in the form of UDP server it gets ACK packet and the corr. IP address

DHCP servers use the MAC address to identify devices and give some devices fixed IP addresses