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

**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.

**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.

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 writes 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 )

•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

Default attributes of a thread :

• It is unbounded

• It is nondetached

• It has a 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

Multiple threads cannot wait for a thread to terminate

you can detach a thread by calling pthread\_detach(thread\_id)

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

A thread can set its cancel state as

int pthread\_setcancelstate(int state, int \*oldstate); ( PTHREAD\_CANCEL\_DISABLE/ENABLE, NULL )

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

( PTHREAD\_CANCEL\_DEFERRED/ASYNCHRONOUS, NULL )

pthread\_join(pid, &status) is the only way to know whether thread was cancelled successfully or not

pthread\_cancel() is implemented using signals and can result into memory leaks

So to prevent that 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).)

On Linux, cancellation is implemented using signals. Signals are fired up in the process queue.

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

you should also destroy the attr structure like

pthread\_attr\_destroy(&attr);

Synchronization Objects

* Mutex
* Semaphore
* Condition Variables

TLS or thread local storage is an old mechanism of pthreads but new mechanism on C++11 (where it is a storage class, auto and register has been deprecated ) Here a global/static object is defined but each thread maintains a copy of each of that variable

Following are the associated functions

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 depenedent on the thread object...

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);

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

PA 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 server the client opens a new port on its side, it opens a socket on its side on its IP and waits for the connection

As soon as it gets the connection it creates a new socket combining the address of client and its own address

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.

Advanages 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. 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);