

THREADS

Synchronization:

- When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes.....
- threads to synchronize their use of shared resources, so that one thread doesn't try to access a shared variable at the same time as another thread is modifying it.

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as mutual exclusion. When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques[1] are not applied, it may cause a race condition where, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Thread:

PTHREAD(POSIX THREADS)

- A Thread is a concurrent unit of execution. It has its own call stack for methods being invoked, their arguments and local variables. Each application has at least one thread running when it is started, the main thread, in the main ThreadGroup . The runtime keeps its own threads in the system thread group.
- A thread is an independent path of execution within a program. Many threads can run concurrently within a program
- The functions in the Pthread API do things differently. All Pthread functions return 0 on success and positive value on failure.

Thread creation:

- When a program is started, the resulting process consists of a single thread called the **initial** or **main thread**.
`#include <pthread.h>`
`int pthread_create(pthread_t *thread id, pthread_attr_t *attr, void*(*function name)(void *), void *arg)`
- pthread_create() creates a new thread, much as fork creates a new process with attributes specified by attr(stack size, detach stat) within a process. The new thread commences execution by calling the function identified by start_routine with arguments in arg.
- The thread that calls pthread_create() continues execution with next statements that follow the call.
- – Upon successful completion, function stores the ID of the created thread in the location referenced by thread(first argument).
- – If you need to pass more than one argument to the start_rtn function, then you need to store them in a structure and pass the address of the structure in arg.
–When a thread is created there is no guarantee which runs first

Thread termination:

- The execution of a thread terminates in one of the following ways without stopping the entire process.
- – The thread's start_routine function performs a return specifying a
- return value. The return value is the thread's exit code.

- – The thread can be cancelled by another thread in same process using `pthread_cancel()`.
`int pthread_cancel(pthread_t tid);`
- – Any of the threads call `exit()`, or the main thread performs a return in the main function, which causes all threads in the process to terminate.
– The thread can call `pthread_exit()`
`#include <pthread.h>`
`void pthread_exit(void *value_ptr);`

• The `pthread_exit()` function terminates the calling thread and makes the value `value_ptr` available to other threads in the process by calling the `pthread_join` function.

• If the main thread calls `pthread_exit()` instead of calling `exit()` or performing a return, then the other threads continue to execute.

Wait for thread termination(join):

`Include <pthread.h>`

`int pthread_join(pthread_t thread, void **value_ptr);`

–The `pthread_join()` function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

–On return from a successful `pthread_join()` call with a non-NULL `value_ptr` argument, the value passed to `pthread_exit()` by the terminating thread is made available in the location referenced by `value_ptr`.

–If successful, the `pthread_join()` function returns zero.

Synchronization between threads

When multiple threads of control trying to increment the same variable at the same time, the increment operations are usually broken down into three steps.

- Read the memory location into a register.
- Increment the value in the register.
- Write the new value back to the memory location.

ThreadS Synchronization :

•Threads use two tools to synchronize their actions:

– **Mutexes:** allows threads to synchronize their use of shared resources, so that one thread doesn't try to access a shared variable at the same time as another thread is modifying it, ensure that only one thread at a time can access the variable.

A mutex has two states: locked and unlocked.

- At any moment, at most one thread may hold the lock on a mutex. Attempting to lock a mutex that is already locked either blocks or fails with an error, depending on the method used to place the lock.
- **When a thread locks a mutex, it becomes the owner of that mutex. Only the mutex owner can unlock the mutex.** Acquire and release are also used synonymously for lock and unlock.
Each thread employs the following protocol for accessing a resource:
– Lock the mutex for the shared resource.
– Access the shared resource, and

- Unlock the mutex.

- If multiple threads try to execute this block of code(**a critical section**) the fact that only one thread can hold the mutex(the others remain blocked) means that only one thread at a time can enter the block.
- A mutex is a variable of type **pthread_mutex_t**. Before it can be used a mutex must always be initialized. For statically allocated mutex, we can do this by assigning it the value **PTHREAD_MUTEX_INITIALIZER** as below.

pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

- To lock and unlock we use the below functions.

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

- To lock a mutex we must specify the mutex in a call to

pthread_mutex_lock(). If the mutex is currently unlocked, this call locks the mutex and returns immediately.

- If the mutex is currently locked by another thread, then **pthread_mutex_lock()** blocks until the mutex is unlocked, at which point it locks the mutex and returns.

- If the calling thread itself has already locked the mutex given to **pthread_mutex_lock()**, then for the default type of mutex, one of the two implementations defined possibilities may result:

– The thread **deadlocks**, blocked trying to lock a variable it already owns

– or the call fails, returns the error EDEADLK.

•The pthread API provides two variants of **pthread_mutex_lock()**.

pthread_mutex_trylock() and **pthread_mutex_timedlock()**.

– The **pthread_mutex_trylock()** function is the same as **pthread_mutex_lock()**, except that if the mutex is currently locked, **pthread_mutex_trylock()** fails, returning the error EBUSY.

Synchronization with mutexes

–Mutexes, which act as a mutual exclusion device to protect critical sections of code.

–Mutexes act by allowing the programmer to “lock” an object so that only one thread can access it.

–To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.

The basic functions required to use mutexes are:

#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

As usual, 0 is returned for success, and on failure an error code is returned, but errno is not set.

A thread that becomes the owner of a mutex is said to have acquired the mutex and the mutex is said to have become locked.

When a thread gives up ownership of a mutex it is said to have released the mutex and the mutex is said to have become unlocked.

pthread_mutex_init()

initialises the mutex referenced by mutex with attributes specified by attr.

#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

- Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

- On success, the **pthread_mutex_init()** and **pthread_mutex_destroy()** functions return zero, else an error number is returned to indicate the error.-

pthread_mutex_lock()

The mutex object referenced by mutex is locked by calling pthread_mutex_lock().

- **#include <pthread.h>**

int pthread_mutex_lock(pthread_mutex_t *mutex);

- If the mutex is already locked, the calling thread blocks until the mutex becomes available.
- Attempting to relock the mutex causes **deadlock**.
- If successful, the pthread_mutex_lock() return zero.
- Otherwise, an error number is returned to indicate the error.

pthread_mutex_unlock()

The pthread_mutex_unlock() function releases the mutex object referenced by mutex.

- **#include <pthread.h>**

int pthread_mutex_unlock(pthread_mutex_t *mutex);

- The manner in which a mutex is released is dependent upon the mutex's type attribute (Scheduling policy).
- If successful, the pthread_mutex_lock() return zero.
- Otherwise, an error number is returned to indicate the error.-

pthread_mutex_destroy()

The pthread_mutex_destroy() function destroys the mutex object referenced by mutex;

- **#include <pthread.h>**

int pthread_mutex_destroy(pthread_mutex_t *mutex);

- A destroyed mutex object can be re-initialised using **pthread_mutex_init()**;

- It is safe to destroy an initialised mutex that is unlocked.
- On success it returns zero, otherwise an error number is returned to indicate the error.

- **Condition variables:** perform a complementary task: they allow threads to inform each shared variable or shared resource has changed state.

or

Condition variables are synchronization primitives that enable threads to wait until a particular **condition** occurs. **Condition variables** are user-mode objects that cannot be shared across processes. **Condition variables** enable threads to atomically release a lock and enter the sleeping state.

• **Critical Section:** Section of code that accesses a shared resource and whose execution should be atomic. That is its execution should be interrupted by another thread that simultaneously access the same shared resource.

Or

Here, the important point is that when one process is executing shared modifiable data in its **critical section**, no other process is to be allowed to execute in its **critical section**. Thus, the execution of **critical sections** by the processes is mutually exclusive in time

atomic operation: In concurrent programming, an **operation** (or set of **operations**) is **atomic**, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes.

Deadlock:

If the mutex is already locked, the calling thread blocks until the mutex becomes available.

- Attempting to relock the mutex causes deadlock.

Multi threading :

- Like processes threads are a mechanism that permits an application to perform multiple task concurrently. A single process can contain multiple threads.
- All of these threads are independently executing the same program, and they all share the same global memory, including Text, Data, BSS and Heap segments with other threads.
- But each thread will have a separate stack area within the virtual address space.

The programs with multi thread execution are called multi-threaded application.

- In a multithreaded application, all threads must be running the same program(but different functions).
- The programs with multi thread execution are called multi-threaded application.
- The threads in a process can execute concurrently. On a multi-processor machine multiple threads can execute parallel. If one thread is blocked on I/O, other threads are still eligible to execute.
- A unix process has single thread of control i.e. each process does only one thing at a time. With multiple threads of control, we can design our program to do more than one thing at a time.

Process Vs Threads:

The key differences

Thread and a Process

The processes and threads are independent sequences of execution, the typical difference is that threads run in a shared memory space, while processes run in separate memory spaces.

A process has a self contained execution environment that means it has a complete, private set of basic run time resources particularly each process has its own memory space. Threads exist within a process and every process has at least one thread.

Each process provides the resources needed to execute a program. Each process is started with a single thread, known as the primary thread. A process can have multiple threads in addition to the primary thread.

On a multiprocessor system, multiple processes can be executed in parallel. Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.

Threads have direct access to the data segment of its process but a processes have their own copy of the data segment of the parent process.

Changes to the main thread may affect the behavior of the other threads of the process while changes to the parent process does not affect child processes.

Processes are heavily dependent on system resources available while threads require minimal amounts of resource, so a process is considered as heavyweight while a thread is termed as a lightweight process

Process:

- An executing instance of a program is called a process.

- Using fork() we can create a process and copy all the data from parent process and share the information and shared resources b/w processes is complicated we have to use ipc for that. each process has its own memory.
- context-switch time may be lower for threads than for processes.
- In a multiprocess application different processes can run different programs. Aside from data, threads also share other
- Some operating systems use the term 'task' to refer to a program that is being executed.
- A process is always stored in the main memory also termed as the primary memory or random access memory. .
- Several processes may be associated with a same program.
- On a multiprocessor system, multiple processes can be executed in parallel.
- On a uni-processor system a process scheduling algorithm is applied and the processor is scheduled to execute each process one at a time yielding an illusion of concurrency.

Thread:

- A thread is a subset of the process.
- Using clone() we can create a thread and share the information or shared (global data, text, bss, heap) resources b/w threads easily. memory saving, all threads share same memory.
- In a multithreaded application, all threads must be running the same program (but different functions)
- It is termed as a 'lightweight process', since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel.
- Usually, a process has only one thread of control – one set of machine instructions executing at a time.
- A process may also be made up of multiple threads of execution that execute instructions concurrently.
- Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.
- On a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time.
- All the threads running within a process share the same address space, file descriptors, stack and other process related attributes.
- Since the threads of a process share the same memory, synchronizing the access to the shared data within the process gains unprecedented importance.

INTER PROCESS COMMUNICATION(IPC)

Agenda

- Pipes
- FIFO (Named Pipes)
- Message Queues
- Semaphores
- Shared Memory

Inter Process Communication:

- Processes communicate with each other and with the kernel to coordinate their activities.

- - Linux supports a number of IPC mechanisms.
- - Signals and pipes are two of them but Linux also supports other IPC mechanisms such as

FIFOs (Named Pipes)

- Message Queues

- Semaphores

- Shared Memory

Pipes :

- -A pipe is a method of connecting the standard output of one process to the standard input of another.
- -They provide a method of one-way communications between processes.(Full-duplex pipes are also there).
- -When a process creates a pipe, the kernel sets up **two file descriptors** for use by the pipe.
- -One descriptor is used to allow a path of **input into the pipe (write)**, the other to obtain data from the pipe(**read**).
- -pipes are actually represented internally with a valid inode.
- The size of pipe in linux is 65536byte(64kb)

Creating Pipes in C:

To create a simple pipe , we make use of the *pipe()* system call.

- It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline.

used b/w only related process(parent &child)

- After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).

- When one end of the pipe has been closed the following two rules apply.

• If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read. Normally there will be a single reader and single writer for a pipe.

. Read is a blocking call

• If we write to a pipe whose read end has been close, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns -1.

#include <unistd.h>

int pipe(int fd[2]);

-Returns 0 on success and -1 on error:

-**fd[0] is set up for reading, fd[1] is set up for writing.**

-The first integer in the array (fd[0]) is set up and opened for reading, while the second integer (fd[1]) is set up and opened for writing.

-Visually speaking, the output of fd1 becomes the input for fd0.

Parent process -----> PIPE -----> child
fd[1] fd[0]

popen calle -----> PIPE -----> popen caller
fd[1] fd[0]

Creating Pipes in C:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
    int fd[2], nbytes;
    pid_t childpid;
    char string[] = "Hello, world!\n";
    char readbuffer[80];
    pipe(fd);
    if((childpid = fork()) == -1)
    {
        perror("fork");
        exit(1);
    }
    if(childpid == 0)
    {
        /* Child process closes up input side of pipe */
        close(fd[0]);
        /* Send "string" through the output side of pipe */
        write(fd[1], string, (strlen(string)+1));
        exit(0);
    }
    else
    {
        /* Parent process closes up output side of pipe */
        close(fd[1]);
        /* Read in a string from the pipe */
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Received string: %s", readbuffer);
    }
    return(0);
}
```

dis advantages of pipes:

- 1. uni direction
- 2.used b/w only related proccess(parent &child)
- 3.once you terminate the c program pipe objects are distroyed.data in pipe object lost
- The size of pipe in linux is 65539byte(65kb)
- to overcome few dis advantages we are using named fipes(fifos)

FIFOs (Named Pipes):

- A named pipe works much like a regular pipe.
 - Named pipes exist as a device special file in the file system.
- Fifos used transfer data b/w two unrealed proccess,

- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.
- There are several ways of creating a named pipe.
- `mkfifo()` - make a FIFO special file (a named pipe).
- `mknod()` - create a special or ordinary file.
- `#include <sys/types.h>`
- `#include <sys/stat.h>`
- `int mkfifo(const char *filename, mode_t mode);`
- `int mknod(const char *filename, mode_t mode|S_IFIFO,`
- `(dev_t) 0);`

- Once we have used `mkfifo` to create FIFO we open it using `open`. Normal file IO functions work with FIFO's.
- When we open a FIFO, the non blocking `O_NONBLOCK` affects what happened.
- If not specified: An open for read only blocks until some other process opens the FIFO for writing.
- Similarly an open for write only block until some other process opens the FIFO for reading.
- If specified: An open for read only returns immediately. But an open for write-only returns 1 if no process has the FIFO open for reading.

Creating FIFOs :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
int res = mkfifo("/tmp/my_fifo", 0777);
if (res == 0)
printf("FIFO created\n");
exit(EXIT_SUCCESS);
}
```

dis advantages of fifos

- *BI directional*
 - *only two process can share data*
 - *complexity associated with blocking call `open()` and `read()`*
 - *if not specify 0-rdonly is open is called non block*
- > *these over come by message queues*

difference b/w pipes and named pipes

pipe:

- 1) These are created by the shell automatically.
- 2) They exists in the kernel.

- 3) They can not be accessed by any process, including the process that creates it.
- 4) They are opened at the time of creation only.
- 5) They are unidirectional.

Named Pipe: (also called FIFO, First In First Out)

- 1) They are created programatically using the command `mkfifo`.
- 2) They exist in the file system with a given file name.
- 3) They can be viewed and accessed by any two un-related processes. `ls` cmd shows "p" in the permission bits for a named pipe.
- 4) They are not opened while creation.
- 5) They are Bi-directional.
- 6) A process writing a named pipe blocks until there is a process that reads that data.
- 7) Broken pipe error occurs when the writing process closes the named pipe while another reading process reads it.

Message Queues:

Message queues provide a reasonably easy and efficient way of passing data between two unrelated processes.

- Message queues provide a way of sending a block of data from one process to another.
- There's a maximum size limit imposed on each block of data and also a limit on the maximum total size of all blocks on all queues throughout the system.

msgid_ds structure

```
struct msqid_ds
{
    struct ipc_perm msg_perm;
    msgqnum_t msg_qnum; //Num of messages on queue
    msglen_t msg_qbytes; //max num of bytes on queue
    pid_t msg_lspid;
    //pid of last msgsnd()
    pid_t msg_lrpid;
    //pid of last msgrcv()
    time_t msg_stime;
    //last-msgsnd() time
    time_t msg_rtime;
    //last-msgrcv() time
    time_t msg_ctime; //last-change time
    .
    .
    .
}
```

This structure defines the current status of the queue.

System limits that affects messages queues:

- Size in bytes of largest message we can send 8,192 Bytes
- The maximum size in bytes of a particular queue(i.e., the sum of all the messages on the queue – 819200
- The maximum number of messages queues, systemwide – 100.

– ipc_perm(permission structure) is associated with each message queue. It includes owner user id, group id, creator's user id and group id and access mode.

– When a new queue is created, the following members of the **msqid_ds** structure are **initialized**.

- The **ipc_perm** structure is **initialized**.

- **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are **all set to 0**.

- **msg_ctime** is set to the current time.

- **msg_qbytes** is set to the system limit.

- to create a msg queue object by **msgget()**
- msg queue object created in kernel and maintained in a table
- every msg queue object is identified by unique value is called **key**
- client want to send the msg to server the server is receive the msg
- **msgget()** is used to create a msg object also access already existed objects
- **msgsnd()** is used to add msg to msg queue object

-The message queue function definitions are

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
```

```
int msgctl(int msqid, int cmd[diff operations], struct msqid_ds *buf);
```

```
int msgget(key_t key, int msgflg);
```

```
int msgrcv(int msqid, void *msg_ptr[rx msg], size_t msg_sz, long int msgtype, int msgflg);
```

```
int msgsnd(int msqid, const void *msg_ptr[tx msg], size_t msg_sz, int msgflg);
```

int msgget(key_t key, int msgflg)– need to provide a key value with which name is assigned to msg queue.

- **msgget()** is used to create a msg object also access already existed objects
- **msgsnd()** is used to add msg to msg queue object

– **msgflag == IPC_PRIVATE** -> private queue to that process

– we can pass **IPC_CREAT** with permission bits

eg: **msgflag = 666 | IPC_CREAT**

msgrcv:

The type argument lets us specify which message we want.

Type == 0 :The first message on the queue is returned.

Type > 0: The first message on the queue whose message type equals type is returned.

Type < 0: The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

msgctl: The msgctl function performs various operations on the queue.

```
#include <sys/msg.h>
```

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

The **cmd** argument specifies the command to be performed on the queue.

command

Description

IPC_STAT

sets the data in msqid_ds structure to reflect values associated with the message queue

IPC_SET

If the process has permissions to do so, this sets the values associated with the msg queue to those provided in the msqid_ds structure

IPC_RMID

Deletes the msg queue

difference b/w named pipes and message queues

named pipes are stream based(half duplex)

msg queues are full duplex packet based

Shared memory:

Shared memory allows two unrelated processes to access the same logical memory.

- Shared memory is a very efficient way of transferring data between two running processes.

Fastest of all ipc mechanism is shared memory here no system calls used in this . Switching from user space to kernel space consumes same cpu time.

- Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process.

- All processes can access the memory locations just as if the memory had been allocated by malloc().

-The functions for shared memory resemble those for semaphores:

```
#include<sys/types.h>
```

```
#include<sys/ipc.h>
```

```
#include <sys/shm.h>
```

```
#define key 199220
```

```
char *shm;
```

```
int shmget(key_t key, size_t size, int shmflg);
```

```
void *shmat(int shm_id, const void *shm_addr[null offset], int shmflg);
```

```
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

```
int shmdt(const void *shm_addr);
```

- shmget() is used to create a shm id
- shmat() used to create a page table entry we get valid virtual address range

- client has to read the block of memory until server writes data into it. Also known as synchronization.
- >synchronization is in built in pipes named pipes and message queues
- in pipes
- parent writes()-----child read() is blocking call
- in named pipes sever writes()-----client read() blocking call
- in message queues client msgsnd()----- server msgrsv() blocking call

in shared memory we dnt have any synchronization mechanis in built so we have to use semaphores here

Debugging

The ipcs command provides **information on interprocess communication** facilities, including shared segments. Use the -m flag to obtain information about shared memory.

For example, this code illustrates that one **shared memory** segment, numbered 1627649, is in use:

– ipcs -m

If this memory segment is left behind by the program, you can use the ipcrm command to **remove** it.

– ipcrm shm 1627649

SYSTEM V SEMAPHORES:

Semaphores:

- >synchronization is in built in pipes named pipes and message queues
- in pipes
- parent writes()-----child read() is blocking call
- in named pipes sever writes()-----client read() blocking call
- in message queues client msgsnd()----- server msgrsv() blocking call
- server has to waituntil the client has to write data to 1st half,
- and client has to wait until server copy the data to 2nd half .
- To wait client and server need synchronization.

in shared memory we dnt have any synchronization mechanis in built so we have to use semaphores here

semphores used b/w process synchronization.

1.binary semaphore()the semaphore value /counter valueis either 0 or 1)

2.general semaphore(any possitive value)

System V semaphore isn't a form of IPC similar to the others that we've described(pipes, FIFOs, and message queues). System V semaphors are not used to transfer data between processes. Instead they allow processes to synchronize their actions.

•One common use of semaphore is to synchronize access to a block of shared memory, in order to prevent one process from accessing the shared memory at the same time as another process is updating it.

•The simplest semaphore is a variable that can take only the values 0 and 1, a binary semaphore.

•Semaphores that can take many positive values are called general semaphores.

A semaphore is a kernel-maintained counter(integer) whose value is restricted to being greater than or equal to 0.

Various operations can be performed on a semaphore, including the following.

- Setting a semaphore to an absolute value using **semctl()**
- Adding a number to the current value of the semaphore (increment)
- Subtracting a number from the current value of the semaphore (decrement) using **semop()**

- Waiting for the semaphore value to be equal to 0. (blocking)
block until semaphore counter is made zero,
as soon as counter zero the semop() call comes out of blocking state.

--.>Server process are terminated the semaphore object not removed, destroy semaphore object by using **ipcrm**

•The last of two operations may can use the calling processes to block.

- When lowering a semaphore value, the kernel blocks any attempt to decrease value below 0.

- Similarly waiting for semaphore to equal 0 blocks the calling process if the semaphore value is not currently 0.

•In both cases, the calling process remains blocked until some other process alters the semaphore to a value that allows the operation to proceed, at which point the kernel wakes the blocked process.

Using a Semaphore to synchronize two processes

PROCESS A

Initialize semaphore to 0

|
|
Add 1 to semaphore
|
Subtract 1 from semaphore
blocks

| < -----
resumes

PROCESS B

Subtract 1 from semaphore
blocks

|
> resumes

|
Add 1 to semaphore

General Steps for using a System V Semaphore :

- Create or open a semaphore set using `semget()`.
- Initialize the semaphores in the set using `semctl()` SETVAL or SETALL operation. (Only one process should do this).
- Perform operations on semaphore values using `semop()`. The processes using the semaphore typically use these operations to indicate acquisition and release of a shared resource.
- When all processes have finished using the semaphore set, remove the set using the `semctl()` IPC_RMID operation. (Only one process should do this).

System V Semaphore :

- System V semaphores are rendered unusually complex as they are allocated in groups called semaphore sets. The number of semaphores in set is specified when the set is created using the `semget()`.
- While it is common to operate on a single semaphore at a time, the `semop()` system call allows us to atomically perform a group of operations on multiple semaphores in the same set.
- Each semaphore set has an associated `semid_ds` data structure of the following form.

```
struct semid_ds{  
    struct ipc_perm sem_perm;  
    unsigned short sem_nsems; /*no of semaphores in set*/  
    time_t sem_otime; /*last semop() time*/  
    time_t sem_ctime; /*last-change time*/.....}
```

Creating or opening a Semaphores Set :

- The `semget()` system call creates a new semaphore set or obtains the identifier of an existing set.

```
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems[no of semaphores], int semflg);
```

- The `key` argument is a key generated using one of the methods (usually the value `IPC_PRIVATE` or a key returned by `ftok()`).
- If we are using `semget()` to create a new semaphore set, then **`nsems` specifies the number of semaphores in the set**, and must be **greater than 0**. if we are using the `semget()` to obtain the identifier of an existing set, then `nsems` must be less than or equal to the size of the set (or the error `EINVAL` results). It is not possible to change the number of semaphores in an

existing set.

- The **semflag** argument is a bit mask specifying the permissions to be placed on a new semaphore set or checked against an existing set.
- **IPC_CREATE**: If no semaphore set with the specified key exists, new one is created.
- **IPC_EXCL**: A semaphore set with the specified key already exists, fail with the error **EEXIST**.

Semaphore control operations :

- The **semctl()** system call performs a variety of control operations on a semaphore set or on an individual semaphore within a set.

#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd[set val initialize the value of semaphore], .../*union semun arg[counter value]*/);

- The **semid** argument is the identifier of the semaphore set on which the operation is to be performed.

- For those operations performed on a single semaphore, the **semnum** argument identifies a particular semaphore within a set. For other operations, this argument is ignored, and we can specify **0**.

- The **cmd** argument specifies the operation to be performed.

- Certain operations require a **forth argument** to **semctl()**, which we refer to by the **name arg**. This argument is **optional**.

- **Union semun{ int val;**

struct semid_ds *buf;

unsigned short * array;

#if defined(__linux__)

struct seminfo * _buf;

Generic control operations:

- Below are the **list of cmds used in semctl()**. In each of the below case, the **semnum** argument is ignored.

- **IPC_RMID**: Immediately remove the semaphore set and its associated **semid_ds** data structure. Any process blocked in **semop()** calls waiting on semaphores in this set are immediately awakened, with **semop()** reporting the error **EIDRM**. The **arg** argument is not required.

- **IPC_STAT**: Place a copy of the **semid_ds** data structure associated with this semaphore set in the buffer pointed to by **arg.buf**.

- **IPC_SET**: Update the selected fields of the **semid_ds** data structure associated with this semaphore set using values in the buffer pointed to by **arg.buf**.

Retrieving and initializing semaphore values :

- The following operations retrieve or initialize the value(s) of an individual semaphore or of all semaphores in a set. Retrieving a semaphore value requires read permission on the semaphore, while initializing the value requires write permissions.
- **GETVAL**: `semctl()` returns the value of the `semnum`-th semaphore in the semaphore set specified by `semid`. The `arg` argument is not required.
- **SETVAL**: The value of the `semnum`-th semaphore in the set referred to by `semid` is initialized to the value specified in `arg.val`.
- **GETALL**: Retrieve the values of all of the semaphore in the set referred to by `semid`, placing them in the array pointed to by the `arg.array`. The programmer must ensure that this array is of sufficient size. The `semnum` argument is ignored.
- **SETALL**: Initialize all semaphores in the set referred to by `semid`, using the values supplied in the array pointed to by `arg.array`. `Semnum` is ignored.

Retrieving per-semaphore information :

- The following operations return (via the function result value) information about the `semnum`-th semaphore of the set referred to by `semid`. For all these operations, read permission is required on the semaphore set, and the `arg` argument is not required.
- **GETPID**: Return the process ID of the last process to perform a `semop()` on this semaphore; this is referred to as the `semid` value. If no process has yet performed a `semop()` on this semaphore, 0 is returned.
- **GETNCNT**: Return the number of processes currently waiting for the value of this semaphore to increase; this is referred to as the `semncnt` value.
- Return the number of processes currently waiting for the value of this semaphore to become 0. This is referred to as the `semzcnt` value.

Semaphore Operations :

- The **`semop()`** system call performs one or more operations on the semaphores in the semaphore set identified by `semid`.

`Int semop(int semid, struct sembuf *sops, unsigned int nsops);`

- The **`sops`** argument is a pointer to an array that contains the operations to be performed, and **`nsops`** gives the size of this array(which must contain atleast one element).The elements of the `sops` array are structures of the following form.

`Struct sembuf{`

`unsigned short sem_num; /*semaphore number*/`

```
short sem_op;  
/*Operation to be performed*/  
short sem_flg;  
/*Operation flags  
};
```

```
(IPC_NOWAIT and SEM_UNDO)*/
```

- The sem_num field identifies the semaphore within the set upon which the operation is to be performed.

Semaphore Operations

The sem_op field specifies the operation to be performed.

- If **sem_op > 0**, the value of sem_op is added to the semaphore value. As a result, other process waiting to decrease the semaphore value may be awakened and perform their operations. The calling process must have alter permissions on the semaphore.

- If **sem_op == 0**, the value of the semaphore is checked to see whether it currently equals 0. If it does, the operation completes immediately; other semop() blocks until the semaphore value becomes 0. the calling process must have read permissions.

- If **sem_op < 0**, decrease the value of the semaphore by the amount specified in sem_op. If the current value of the semaphore is greater than or equal to the absolute value of sem_op, the operation completes immediately. Otherwise, semop() blocks until the semaphore value has been increased to a level that permits the operation to be performed without resulting in a negative value. The calling process must have Semaphore Operations

Semantically, increasing the value of a semaphore corresponds to making a resource available so that others can use it, while decreasing the value of a semaphore corresponds to reserving a resource for(exclusive) use by the process.

- When decreasing the value of a semaphore, the operating is blocked if the semaphore value is too low-that is, if some other process has already reserved the resource.

- When a semop() call blocks, the process remains blocks until one of the following occurs:

- Another process modifies the value of the semaphore such that the requested operation can proceed.

- A signal interrupts the semop() call. In this case, the error EINTR results.

- Another process deletes the semaphore referred to by semid. In this case semop() fails with the error EIDRM.

Debugging semaphores:

- Use the command **ipcs -s** to display information about existing

semaphore sets.

- Use the **ipcrm sem** command to remove a semaphore set from the command line.

- For example, to remove the semaphore set with identifier 5790517 use the below line.

```
Ipcrm sem 5790517
```

POSIX SEMAPHORES

Named Semaphores

To work with a named semaphore, we employ the following functions:

The **sem_open()** function opens or creates a semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.

The **sem_post(sem)** and **sem_wait(sem)** functions respectively increment and dec-rement a semaphore's value.

The **sem_getvalue()** function retrieves a semaphore's current value.

The **sem_close()** function removes the calling process's association with a semaphore that it previously opened.

The **sem_unlink()** function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

POSIX SHARED MEMORY

Shared Memory

The **shm_open()** function creates and opens a new shared memory object or opens an existing object.

```
int shm_open(const char *name, int oflag, mode_t mode);
```

Shared memory objects are created as files in a special location of a standard file system. Linux uses a dedicated **tmpfs** file system mounted under the directory **/dev/shm**.

Use the below command to see the list of shared memory objects created **\$ ls -l /dev/shm**

When a new shared memory object is created, it initially has zero length. You have to use **ftruncate()** to set the size of the object

before calling **mmap()**. The created memory is filled with zero.

Pass the file descriptor obtained in the previous step in a call to **mmap()** that specifies **MAP_SHARED** in the **flags** argument. This maps the shared memory into the process's virtual address space. Once we have mapped the object, we can close the file descriptor without affecting the mapping.

MEMORY MAPPINGS

mmap()

•The **mmap()** system call creates a new memory mapping in the calling process's virtual address space. A mapping can be of two types.

– **File mapping:** A file mapping maps a region of a file directly into the calling process's virtual memory. Once a file is mapped, its contents can be accessed by operations on the bytes in the corresponding memory region. The pages of the mapping are(automatically loaded) from the file as required. This type of mapping is also known as a file-based mapping or memory-mapped file.

– **Anonymous mapping:** An anonymous mapping doesn't have a corresponding file. Instead the pages of the mapping are initialized to 0.

mmap() :

•The memory in one process's mapping may be shared with mapping in other processes.(i.e., the page-table entries of each process point to the same pages of RAM). This can occur in two ways.

- – When two processes map the same region of a file, they share the same pages of physical memory.
- – A child process created by **fork()** inherits copies of its parent's mapping, and these mappings refer to the same pages of physical memory as the corresponding mappings in the parent.

When two or more process share the same pages, each process can potentially see the changes to the page contents made by other processes, depending on whether the mapping is private or shared.

Private Mapping & Shared Mapping

• **Private mapping(MAP_PRIVATE):** Modifications to the contents of the mapping are not visible to other processes and, for a file mapping, are not carried through to the underlying file. Although the pages of a private mapping are initially shared in the circumstances describe above, changes to the contents of the mapping are nevertheless private to each process. The kernel accomplishes this using copy-on-write technique. This means that when ever a process attempts to modify the contents of a page, the kernel first creates a new, separate copy of that page for the process(and adjusts the process's page tables).

• **Shared mapping(MAP_SHARED):** Modifications to the contents of the mapping are visible to other processes that share the same mapping and, for a file mapping, are carried through to the underlying layer.

Mmap() function :

#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

Addr argument indicates the virtual address at which the mapping is to be located. If we specify **addr** as **NULL**, the kernel chooses a suitable address for the mapping. This is the preferred way of creating a mapping. We can specify a non-**NULL** value as **addr**, which the kernel takes as a hint about the address at which the mapping should be placed. Kernel chooses an address that doesn't conflict with any existing mapping.

- On success **mmap()** returns the starting address of the new mapping.
- The **length** argument specifies the size of the mapping in bytes.
- The **prot** argument is a bit mask specifying the protection to be placed on the mapping. It can be either **PROT_NONE** or combination of another three flags **PROT_READ**, **PROT_WRITE**, **PROT_EXEC**.
- The **flag** argument is a bit mask of options controlling various aspects of the mapping operation. **MAP_PRIVATE**, **MAP_SHARED**.
- **Offset's** are used with file mappings, they are ignored for anonymous mapping.

Linux Boot Process

From Power ON to Loading Kernel:

- **Step 1:** When we power on PC, **BIOS**(Which is stored on Motherboard) loads into RAM. The purpose of BIOS is load to **load OS or Kernel** into RAM.
- **Step 2:** BIOS search for **Bootable Device**. When bootable device found goes to next step.
- **Step 3:** When bootable device found it loads 1 stage **Boot loader i.e. MBR in RAM**. Size of MBR is just 512 bytes. Just first sector of hardisk.
- **Step 4:** First stage boot loader loads Second stage boot loader i.e. **GRUB or LILO**.
- **Step 5:** When second stage boot loader gets executed in RAM, **Splash screen gets displayed**. Job of second stage boot loader is to **load kernel in RAM**.Linux Boot Process
- **Step 6:** Stage 2 boot loader **loads kernel and optional initial Root File system into RAM**. It passes control to kernel and kernel get **decompressed into RAM and get initialized**. At this second stage boot loader checks hardware and mount root device also loads necessary kernel modules. When it completes first Userspace program gets executed i.e. **init**. **Init** is father of all processes.

From init to Login prompt:

When the kernel is loaded, it immediately initializes and configures the computers memory and configures various hardware attached to the system, including all processors, i/o subsystems, and storage devices. It then looks for the compressed initrd in a predetermined location in memory. Decompresses it, mounts it and loads all necessary drivers.

After this the kernel locates & Starts the first user-space application

/sbin/init.Linux Boot Process

Init is the father of all processes. Its PID is 1.

Before /sbin/init loads into RAM, it reads /etc/inittab file

/etc/inittab is an ASCII text file. Where we can configure multiple parameters for init daemon.

strace

- *Strace* captures and displays information for every kernel system call executed by a Linux application program. It can be run on programs for which no source code is available.

strace ./a.out

- Starting the application from command line simply returns control to console. It produces some systems logs and strace quickly identifies the problems. The output can be some times hundreds of lines. Each line represents a discrete kernel system call. We dont need to understand each and every line of the trace. We are needed to look for any anomalies which might help pinpoint why the program won't run.

Strace variations(1)

- Strace utility has many command line options. One of the more useful is the ability to select a subset of system calls for tracing. For example if you want to see only the network related activity of a given process, issue the command as follows:

strace -e trace=network a.out

- This produces a trace of all the network related system calls, such as socket(), connect(), recvfrom(), send(). This is a powerful way to view the network activity of the given program. Several other subsets are available. For example you can view only a program's file related activity by using "**trace=file**" with open(), close(), read(), write() and so on. Additional subset include process related system calls, signal related system calls, and IPC related system calls.

It is worth nothing that **strace** can deal with tracing programs that spawn additional processes. Invoking strace with the -f option instructs strace to follow child process that are created using the fork() system call.

Strace variations(2)

- One very useful way to employ strace is to use the -c option. This

option produces a high level profiling for your application. Using -c option accumulates statistics on each system call,

- how many times it was encountered,
- how many time errors were returned
- time spent by each system call

strace -c ./a.out

• Some errors might be normal part of your applications operations, but others might be consuming time in ways that you did not anticipate.

ltrace

• The **ltrace** and **strace** utilities are closely related. The ltrace utility does for **library calls** what strace does for **system calls**.

ltrace ./a.out

• Reproduces the output of ltrace on a small program that executes a handful of standard C library calls.

• For each library call, the name of the call is displayed, along with varying portions of the parameters to the call. Similar to strace, the return value of the library call is then displayed. As with strace, this tool can be used on programs for which source code is unavailable.

• Similar to strace, a variety of switches affect the behaviour of ltrace. You can display the value of the program counter at each library call, which can be helpful in understanding your applications program flow. **ltrace -c ./a.out**

• The ltrace tool is available only for programs that have been compiled to use dynamically linked shared library objects.

mtrace(1)

• Mtrace is a simple utility that analyses and reports on calls to malloc(), realloc(), and free() in your applications. It is easy to use and can potentially help spot trouble in your application.

• Mtrace must be configured and compiled in your target architecture.

Mtrace is a malloc replacement library that is installed on your target. So your embedded Linux distribution should contain the mtrace package.

• For you to use mtrace, three conditions must be satisfied.

- A header file **mcheck.h**, must be included in your source file.
- The application must call mtrace() to install the handlers.
- The environment variable **MALLOC_TRACE** must specify the name of a writeable file to which the trace data is written.

• When these conditions are satisfied, each call to one of the traced functions generates a line in the raw trace file defined by **MALLCO_TRACE**.

mtrace(2)

• The trace data look like this in the file mentioned in environment.

@ ./a.out:[0x804841e] + 0x9472390 0x10

- The @ sign signals that the trace line contains an address or function name. The program is executing at the address in square brackets, 0x8048ec. The + sign that this is a call to allocate memory. A call to free would be indicated by minus sign. The next field indicates the virtual address of the memory location being allocated or freed. The last field is the size.

- The data format is not very user friendly. For this reason the mtrace utility includes a utility that analyses the raw data and report any inconsistencies. The analysis utility is a perl script supplied with mtrace package. In the simplest the Perl script prints a single line with the message No Memory Leaks.

```
[vamsi@localhost day37]$ mtrace ./a.out output.txt
```

Memory not freed:

Address Size Caller

0x088e3378 0x10 at /media/500GB/Classes/day37/mtrace.c:15

mtrace(3)

- This simple tool can help you spot trouble before it happens as well as find trouble when it occurs. Notice that the Perl script displays the filename and line number of each call to malloc() that does not have a corresponding call to free() for the given memory location.

- This requires debugging information in the executable file generated by passing the -g flag on the compiler. If no debugging information is found the script simply reports the address of the function malloc().

dmalloc(3)

- Dmalloc picks up where mtrace leaves off. The mtrace package is simple, relative non intrusive package most useful for simple detection of malloc/free unbalance conditions. The dmalloc package lets you detect much wider range of dynamic memory-management errors. Compared to mtrace, dmalloc is highly intrusive.

- Depending on the configuration, dmalloc can slow your application to crawl. It is definitely not the right tool if you suspect memory errors due to race conditions or other timing issues. dmalloc will definitely change the timing of your application.

- Dmalloc is a debug malloc library replacement. These conditions must be satisfied.

- Application code must include the dmalloc.h header file.

- The application must be linked against the dmalloc library.

- The dmalloc library and utility must be installed on your embedded target.

- Certain environment variables that the dmalloc library references must

GDB: The GNU Project Debugger

What is GDB?

GDB, the GNU Project debugger, allows you to see what is going on 'inside' another program while it executes -- or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

The program being debugged can be written in Ada, C, C++, Objective-C, Pascal (and many other languages). Those programs might be executing on the same machine as GDB (native) or on

another machine (remote). GDB can run on most popular UNIX and Microsoft Windows variants.

A mistake has been detected in the release tar files for all GDB releases from version 6.0 to version 7.3 (included). The mistake has been corrected, and the FSF issued the following announcements:

The difference between fork(), vfork(), exec() and clone()

Fork : The fork call basically makes a duplicate of the current process, identical in almost every way (not everything is copied over, for example, resource limits in some implementations but the idea is to create as close a copy as possible).

The new process (child) gets a different process ID (PID) and has the the PID of the old process (parent) as its parent PID (PPID). Because the two processes are now running exactly the same code, they can tell which is which by the return code of fork - the child gets 0, the parent gets the PID of the child. This is all, of course, assuming the fork call works - if not, no child is created and the parent gets an error code.

Vfork : The basic difference between vfork and fork is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues.

This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use _exit()); actually, this is also true for the child of a normal fork()).

Exec : The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point. exec() replaces the current process with a the executable pointed by the function. Control never returns to the original program unless there is an exec() error.

Clone : Clone, as fork, creates a new process. Unlike fork, these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.

When the child process is created with clone, it executes the function application `fn(arg)`. (This differs from `for`, where execution continues in the child from the point of the fork call.) The `fn` argument is a pointer to a function that is called by the child process at the beginning of its execution. The `arg` argument is passed to the `fn` function.

When the `fn(arg)` function application returns, the child process terminates. The integer returned by `fn` is the exit code for the child process. The child process may also terminate explicitly by calling `exit(2)` or after receiving a fatal signal.

Fork() and vfork difference:

The **fork()** syscall generates two identical processes with separate memory.

The **vmfork()** syscall generates two processes that share the same memory.

With **vmfork()** the parent will wait for the child terminates.

The parent inherits from the variables that the program is sharing.

So after the child was called, all variables modified inside the child will still be modified inside the parent.

Interrupt handler

What are bottom halves?

Interrupt handling is divided into two parts viz. Top Half and Bottom Half. In top half we do only those works that require immediate attention. In bottom half we do rest of the work. Some bottom halves do run in interrupt context but all interrupt lines are enabled while executing the bottom halves.

Why do we need it?

- B'coz we want our Interrupt handler to do only the time critical stuffs and return back to the process context .
- B'coz we don't want to keep the interrupt lines busy.

What are the bottom halves being used in the present linux kernels?

1. SoftIrqs
2. Tasklets
3. WorkQueues

Tasklets Vs SoftIrqs

Tasklets and SoftIrqs are two ways to implement bottom halves (and definitely not the only two ways). Tasklets are dynamically created and are simpler to use. So while deciding upon which one to use , go for tasklets unless the work is time critical. Networking and Block devices, whose work is time critical use SoftIrqs.

SoftIrqs Vs Tasklets Vs WorkQueues

.Deferred work runs in Interrupt context in case of SoftIrqs and Tasklets while it runs in process context in case of workqueues.

- SoftIrqs(same or different) can run simultaneously on different processor cores ; same Tasklets can't run simultaneously on different CPU cores but different Tasklets definitely can; workqueues can run on different CPU cores simultaneously.
- As SoftIrqs and Tasklets run in interrupt context they can't sleep while workqueues can sleep as they run in the process context.
- Both SoftIrqs and Tasklets can't be preempted or scheduled

while Workqueues can be.

- SoftIrqs are not easy to use while Tasklets and WorkQueues are easy to use.
- If the code in question is highly threaded(too many subroutines) for ex. Networking applications then SoftIrq is the best bet as they are the fastest.
- If the code is not highly threaded then the device driver developer must go for Tasklets as they have the simplest interface.
- If the deferred work has to run in the process context then WorkQueues is the only option.
- So, in general if your bottom halve can sleep then use WorkQueues else use Tasklets.
- When it comes to ease of use WorkQueues are the best then comes tasklets and in the end comes softirqs as they have to be statically created and require proper thinking before implementing.

The key difference between the two is that tasklets execute quickly, for a short period of time, and in atomic mode, while workqueue functions may have higher latency but need not be atomic.

Tasklets run in software interrupt context with the result that all tasklet code must be atomic. Instead, workqueue functions run in the context of a special kernel process; as a result they have more flexibility. In particular, workqueue functions can sleep.

TASKLET:

A tasklet exists as a data structure that has to be initialized before use. Initialization can be performed by calling a specific function or by declaring the structure using certain macro.

```

#include <linux/interrupt.h>

struct tasklet_struct{
/*...*/

void (*func)(unsigned long);
unsigned long data;
}

void tasklet_init(struct tasklet_struct t,void (*func)(unsigned
long),unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name,func,data);

```

How to Pass Command Line Arguments to a Kernel Module?

Generally the word command line arguments make you strike to argc/argv in C, here coming to Linux kernel modules approach is bit different and even easy to....!! lets go for a walk on this concept.

To allow arguments to be passed to your module, declare the variables that will take the values of the command line arguments as global and then use the `module_param()` macro, defined in `linux/moduleparam.h` to set the mechanism up.

Value is assigned to this variable at runtime by a command line arguments that are given like `$ insmod mymodule.ko myvariable=5` while inserting/loading the module into kernel.

The variable declarations and macros should be placed at the beginning of the module for clarity.

In this post [Loadable Kernel Module](#) i have explained the basic concepts of kernel modules.

The `module_param()` macro takes 3 arguments:

- arg1 : The name of the variable.
- arg2 : Its type
- arg3 : Permissions for the corresponding file in sysfs.

Explain Callback functions.

A callback function is one which is passed as an argument to another function and is invoked after the completion of the parent function.

In other words **callback** is a piece of executable code that is passed as an argument to other code, which is expected to *call back* (execute) the argument at some convenient time. The invocation may be immediate as in a **synchronous callback** or it might happen at later time, as in an **asynchronous callback**.

Eg.

```
#include <stdio.h>

int a(int (*callback)(void))
{
    printf("inside parent function");
    callback();
}

int test()
{
    printf("inside callback function");
}

int main (void)
{
    a(&test);
    printf("main gotta end");
    return 0;
}
```

In this example what we want to do is to call our callback function test when the function a is finished with its execution. What we do is that we pass the address of our callback function test to the function a . After we are done with the code of function a we call the callback function by calling callback(), internally this function will call our required callback function test as we have passed it as an argument.

What is the difference between a Mutex and a Semaphore?

Strictly speaking, mutex is a locking mechanism whereas semaphore is a signaling mechanism.

Mutex is the lock to a toilet and semaphore is the number of identical keys to a shared resource. Imagine a buffer being split up into 4 parts and then the semaphore count can become 4, i.e there are now 4 instance to the same resource. Mind it , it is not the same resource. A semaphore with a value of 0 is similar to mutex but is has a difference, in mutex ,the thread within the same process can unlock the mutex whereas in semaphore an external thread can also free the semaphore by giving wakeup signal.

What are the Synchronization techniques used in Linux Kernel?

For simple counter variables or for bitwise ----->atomic operations are best methods.

```
atomic_t count=ATOMIC_INIT(0); or atomic_set(&count,0);
atomic_read(&count);
atomic_inc(&count);
atomic_dec(&count);
atomic_add(&count,10);
atomic_sub(&count,10);
```

Spinlocks are used to hold critical section for short time and can use from interrupt context and locks can not sleep,also called busy wait loops. fully spinlocks and reader/writer spin locks are available.

```
spinlock_t my_spinlock;
spin_lock_init( &my_spinlock );
spin_lock( &my_spinlock );
// critical section
spin_unlock( &my_spinlock );
Spinlock variant with local CPU interrupt disable
spin_lock_irqsave( &my_spinlock, flags );
// critical section
spin_unlock_irqrestore( &my_spinlock, flags );
if your kernel thread shares data with a bottom half,
spin_lock_bh( &my_spinlock );
// critical section
spin_unlock_bh( &my_spinlock );
```

If we have more readers than writers for our shared resource
Reader/writer spinlock can be used

```
rwlock_t my_rwlock;
rwlock_init( &my_rwlock );
write_lock( &my_rwlock );
// critical section -- can read and write
write_unlock( &my_rwlock );
```

```
read_lock( &my_rwlock );  
// critical section -- can read only  
read_unlock( &my_rwlock );
```

Mutexs are used when we hold lock for longer time and if we use from process context.

```
DEFINE_MUTEX( my_mutex );  
mutex_lock( &my_mutex );  
mutex_unlock( &my_mutex );
```

When should one use Polling and when should one use Interrupts?

Both the mechanisms have their own pluses and minuses.

We should use interrupts when we know that our event of interest is-

1. Asynchronous
2. Important(urgent).
3. Less frequent

We should use polling when we know that our event of interest is-

1. Synchronous
2. Not so important
3. Frequent(our polling routine should encounter events more than not)

How function pointers are shared across different processes? using which IPCs?

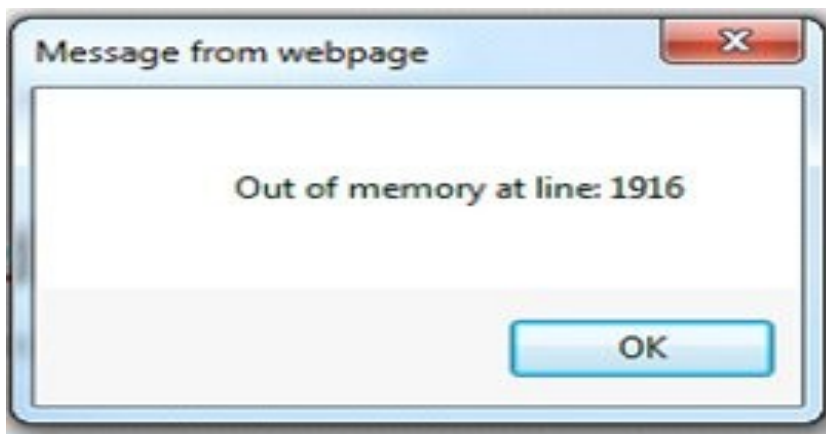
Two processes can not share function pointers. If we want to use functions in two processes we will have to make library for those functions and we use that library in our processes.

What is memory leak?

When Dynamically allocated memory is not freed after its use, it leads to memory leak.

It is a very sensitive issue in embedded programming as lots of memory leaks can cause system to stall.

How to avoid memory leak in Linux?



There are three ways to avoid them -

1. don't create memory leak (avoid while writing code), use free function when done with the memory.
2. use some static tool like covery to detect memory leak and fix it.
3. use some dynamic tool (run time) to detect memory leak eg. valgrind and fix it.

What is a Linux Device Driver Model ?

The Linux Device model is built around the concept of busses, devices and drivers. All devices in the system are connected to a bus of some kind. The bus does not have to be a real one; busses primarily exist to gather similar devices together and coordinate initialization, shutdown and power management.

When a device in the system is found to match a driver, they are bound together. The specifics about how to match devices and drivers are bus-specific. The PCI bus, for example, compares the PCI Device ID of each device against a table of supported PCI IDs provided by the driver. The platform bus, on the other hand, simply compares the name of each device against the name of each driver; if they are the same, the device matches the driver.

Binding a device to a driver involves calling the driver's `probe()` function passing a pointer to the device as a parameter. From this point on, it's the responsibility of the driver to get the device properly initialized and register it with any appropriate subsystems.

Devices that can be hot-plugged must be un-bound from the driver when they are removed from the system. This involves calling the driver's `remove()` function passing a pointer to the device as a parameter. This also happens if the driver is a dynamically loadable module and the module is unloaded. All device driver callbacks, including `probe()` and `remove()`, must follow the return value.

What are monolithic and micro kernels and what are the differences between them?

Monolithic kernel is a single large processes running entirely in a single address space. It is a single static binary file. All kernel services exist and execute in kernel address space. The kernel can invoke functions directly. The examples of monolithic kernel based OSs are Linux, Unix.

In Microkernels, the kernel is broken down into separate processes, known as servers. Some of the servers run in kernel space and some run in user-space. All servers are kept separate and run in different address spaces. The communication in microkernels is done via message passing. The servers communicate through IPC (Interprocess Communication). Servers invoke "services" from each other by sending messages. The separation has advantage that if one server fails other server can still work efficiently. The example of microkernel based OS are Mac OS X and Windows NT.

Differences--

- 1) **Monolithic kernel is much older than Microkernel.** It's used in Unix . While Idea of microkernel appeared at the end of the 1980's.
- 2) the example of os having the **Monolithic kernels are UNIX , LINUX** .While the os having **Microkernel are QNX , L4 , HURD , initially Mach** (not mac os x) later it will converted into hybrid kernel , even MINIX is not pure kernel because device driver are compiled as part of the kernel .
- 3) **Monolithic kernel are faster than microkernel** . While The first microkernel Mach is 50% slower than Monolithic kernel while later version like L4 only 2% or 4% slower than the Monolithic kernel .
- 4) **Monolithic kernel generally are bulky** . While Pure monolithic kernel has to be small in size even fit in s into processor first level cache (first generation microkernel).
- 5) **In the Monolithic kernel device driver reside in the kernel space** . While In the Microkernel device driver reside in the user space.
- 6) **Since the device driver reside in the kernel space it make monolithic kernel less secure than microkernel** . (Failure in the driver may lead to crash) While Microkernels are more secure than the monolithic kernel hence used in some military devices.
- 7) **Monolithic kernels use signals and sockets to ensure IPC while microkernel approach uses message queues** . 1 gen of microkernel poorly implemented IPC so were slow on context switches.
- 8) **Adding new feature to a monolithic system means recompiling the whole kernel While You can add new feature or patches without recompiling.**

What is the difference between kill-6 and kill -9?

SIGKILL and SIGABRT are two type of signals that are sent to process to terminate it.

SIGKILL is equivalent of "kill -9" and is used to kill zombie processes, processes that are already dead and waiting for their parent processes to reap them.

SIGABRT is equivalent of "kill -6" and is used to terminate/abort running processes.

SIGKILL signal cannot be caught or ignored and the receiving process cannot perform any clean-up upon receiving this signal.

SIGABRT signal can be caught, but it cannot be blocked.

Given a pid, how will you distinguish if it is a process or a thread ?

Do `ps -AL | grep pid`

1st column is parent id and the second column is thread (LWP) id. if both are same then its a process id otherwise thread.

Why do we need two bootloaders viz. primary and secondary?

When the system starts the BootROM has no idea about the external RAM. It can only access the Internal RAM of the CPU. So the BootROM loads the primary bootloader from the boot media (flash memory) into the internal RAM. The main job of the primary bootloader is to detect the external RAM and load the secondary bootloader into it. After this, the secondary bootloader starts its execution.

Why Kernel Code running in interrupt context cannot sleep?

This is because the kernel design architecture wants it to be like this. Why so?

This is because interrupt context is not considered to be a process. So, it can't sleep. The interrupt code is doing some work on behalf of process and if slept then it will not only lead to the blocking of interrupt code but also the process that has called it.

On x86-32 Linux, at which address the code segment of the program and stack starts?

code segment of the program starts from 0×08048000 (though it can be changed but this is the default one). We don't start from 0 because that is reserved for NULL pointer exception.

Stack grows downwards starting at address $0xbfff0000$.

How will the User Space mapping look like when our RAM is less than 896 MB?

User space mapping is not linear. The entire user space (3GB) is mapped to whatever RAM is there, 896 MB in this case. The physical page can be mapped to several virtual addresses simultaneously.

Explain the module loading in Linux.

A module can be loaded to Linux Kernel in two ways

1. Statically
2. Dynamically

Static loading means that the module is loaded in the memory with the kernel loading itself.

Dynamic loading means that the module is loaded into the kernel at the run time.

The command that is used to achieve it is `insmod`.

The user must have the root permission to do so.

e. `sudo insmod test.ko`

How does Linux Manage memory?

1> Bring pages in only when needed (demand paging of text sections)

- 2> Keep pages brought into memory as long as possible to avoid reading from slower devices (page cache etc.)
- 3> Under memory pressure, get rid of redundant copies of data (text pages being freed during memory pressure)
- 4> If required memory can not be freed as there is only one copy of it (data sections), move the data to slower medium(swapping onto disk) and book keep the information so that they can be brought back into memory when needed.
- 5> Use processor MMUs for isolation and privileges. These MMU components (TLBs and caches) act as a cache of page tables.
- 6> Use optimized algorithms and data structures.

What is the difference between IRQ and FIQ in case of ARM?

ARM treats FIQ(Fast interrupt request) at a higher priority as compared to IRQ(interrupt request). When an IRQ is executing an FIQ can interrupt it while vice versa is not true.

ARM uses a dedicated banked out register for FIQ mode ; register numbers R8-R12.

<http://learnlinuxconcepts.blogspot.in/2014/06/arm-architecture.html>

So when an FIQ comes these registers are directly swapped with the normal working register and the CPU need not take the pain of storing these registers in the stack. So it makes it faster.

One more point worth noting is that the FIQ is located at the end of exception vector table(0X1c) which means that the code can run directly from 0X1C and this saves few cycles on entry to the ISR.

Where are macros stored in the memory?

```
#define func() func1(){...}
```

Macros aren't stored anywhere separately. They get replaced by the code even before compilation. The compiler is unaware of the presence of any macro. If the code that replaces macro is large then the program size will increase considerably due to repetition.

What is a kernel Panic?

It is an action taken by linux kernel when it experiences a situation from where it can't recover safely. In many cases the system may keep on running but due to security risk by fearing security breach the kernel reboots or instructs to be rebooted manually.

It can be caused due to various reasons-

1. Hardware failure
2. Software bug in the OS.
3. During booting the kernel panic can happen due to one of the reasons-
 - a. Kernel not correctly configured, compiled or installed.
 - b. Incompatibility of OS, hardware failure including RAM.
 - c. Missing device driver

- d. Kernel unable to locate root file system
- e. After booting if init process dies.

In windows operating systems the equivalent term is stop error (or, colloquially Blue screen of death).

What is bus error? what are the common causes of bus errors?

The first thing that needs to be addressed is: What is a bus? A bus is a communication unit that allows the CPU to interact with peripherals, there are different type of buses such as PCI, I2C, MDIO, Memory Buses, etc. Normally each bus would have its own protocol for transmitting data across devices, for example in the case of PCI we can have timeout errors or windows errors (data is directed to unknown addresses/devices). In memory, bus errors would refer to alignment but other errors could be attributed to physical HW problems such as faulty connections. Other type of bus errors could be single and multiple bit errors, this could be addressed by using ECC memory.

How a system call is executed in X86 architecture?

- A vector in the interrupt descriptor table (IDT) is used to invoke the system call.
- Only one vector is allocated for the system calls.
- Immediately a question comes into our mind that since there is only one vector so how so many system calls can be serviced?
- This is done by having a generic function (we can say an ISR) which will multiplex all other system calls.
- That means when an interrupt (software interrupt using INT instruction) is raised on this vector, the generic function will be called and a system call number is passed as an argument to this function.
- This generic function uses this system call number as an index into the `sys_call_table` array and gets the address (function pointer) of the system call and invokes that system call.
- `arch/x86/kernel/syscall_64.c#L25`
- Interrupt vector used for the system call is 0x80(128), i.e interrupt descriptor table's 128th entry.(80 in hex is 128 in decimal).
- 128th entry in the IDT table contains the address of the `system_call()` function. `system_call()` is defined in `arch/x86/kernel/entry_32.S`.

Compare I2C and SPI protocols.

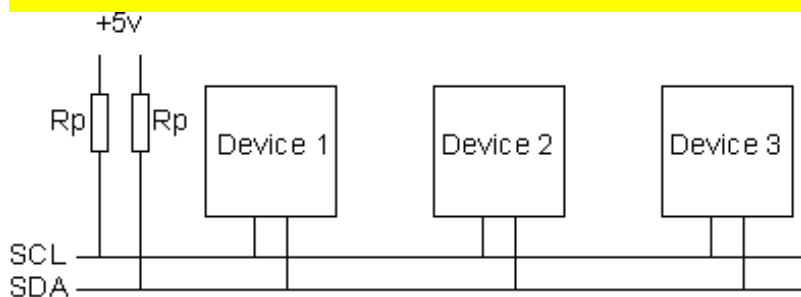
- SPI protocol requires more **hardware**(I2C needs 2 lines and that's it, while SPI formally defines at least 4 Signals).
- SPI protocol is **faster** than I2C, it works in full duplex mode, can transmit upto 10Mbps whereas I2C is limited to 1Mbps in normal mode and 3.4Mbps in fast mode.
- SPI can have **only one master** whereas I2C supports more than one masters.
- In SPI there is no limitation to the number of bits transmitted in one frame(I2c 8bits).
- SPI is **non standard** whereas I2C is standard protocol.

I2C Protocol

I²C (Inter-Integrated Circuit) is a multi-master, multi-slave, single-ended (data flow in only one direction at a time), serial computer bus invented by Philips Semiconductor, known today as NXP Semiconductors, used for attaching low-speed peripherals to computer's motherboards and embedded systems.

The Physical I2C Bus

- Physically I2C consists of just 2 buses.
- One bus is SDA for data transfer and the other one is SCL for the clock.
- All the devices on the I2C bus are connected on these two lines.



- Both these lines are open drain lines, i.e. the chip can drive these lines low but can't drive it high.
- To make it high we need to connect a pull up register to both the lines connected to 5V supply.
- We don't need separate pull up register for each device.

Masters and Slaves

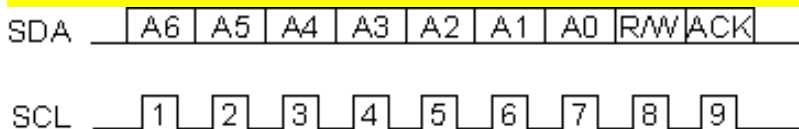
- The devices on I2C bus are either master or slave.
- Characteristics of a master-

1. Master is the one who drives the clock.

2. Master is the one who initiates the transfer.
3. Though both master and slave send data over the bus but the transferred is controlled by master.

I2C Device Addressing

- I2C device addresses are either 7 bit or 10 bit.
- Considering 7 bit addressing we can connect upto 128 devices to the I2C bus.



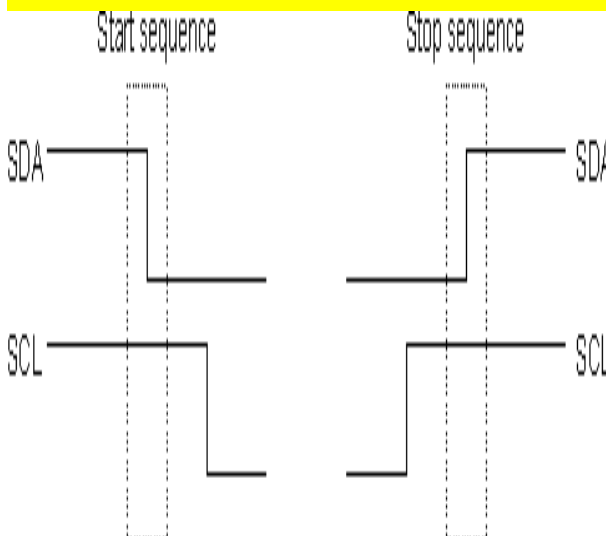
- While sending we send the address as a 8 bit sequence with 8th bit telling the slave if master is **reading**(1) or **writing**(0) from /to the device.

The I2C Protocol

- SDA is allowed to change only when SCL is low , but it is not followed in 2 cases.

1. Start Sequence

2. Stop Sequence



- The start and stop sequence is sent by master and it marks the beginning and end of the transfer with the slave device.
- Steps for **writing to** the slave-

1. Send a start sequence(all the I2C devices start to listen)
2. Send the I2C address of the slave with the R/W bit low (even address/writing)(all the slaves listen to it and matches it with their own address).
The slave sends the acknowledgement bit in return if the address matches.
3. Send the internal register number you want to write to
4. Send the data byte
5. [Optionally, send any further data bytes]
6. Send the stop sequence.

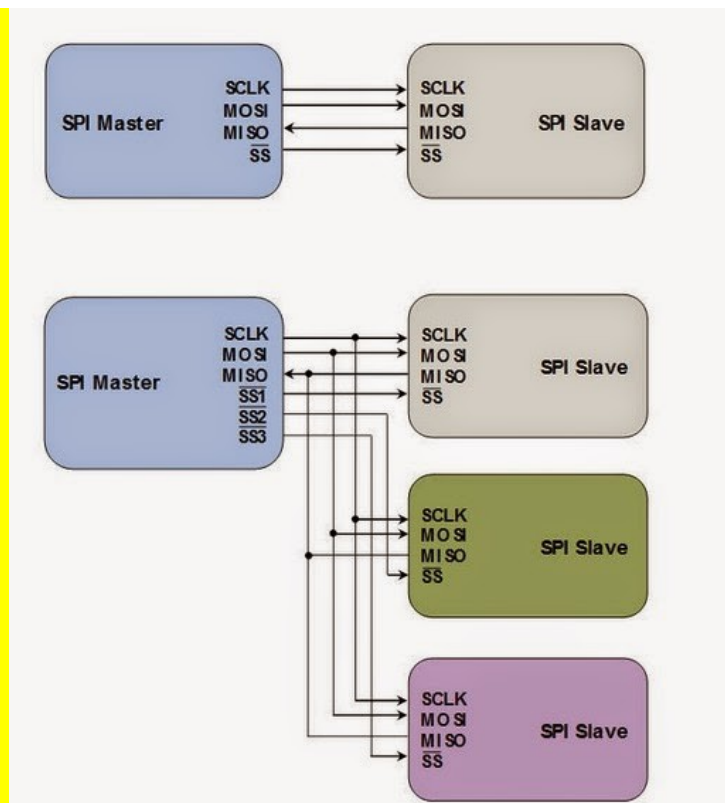
- Steps for **reading from** the slave-

1. Send a start sequence(all the I2C devices on the bus start to listen)
2. Send I2C address of the slave with the R/W bit low (even address/write)(all the slaves listen to it and matches it with their own address).
The slave sends the acknowledgement bit in return if the address matches.
3. Send Internal address of the register to which we need to write.
4. Send a start sequence again (repeated start)
5. Send I2C address of the slave with the R/W bit high (odd address)(slave knows it has to be read from)
6. Read data byte from slave
7. Send the stop sequence.

The peculiar thing in this operation is that even in read operation we do one write operation to know the register from which we need to read from.

SPI Protocol

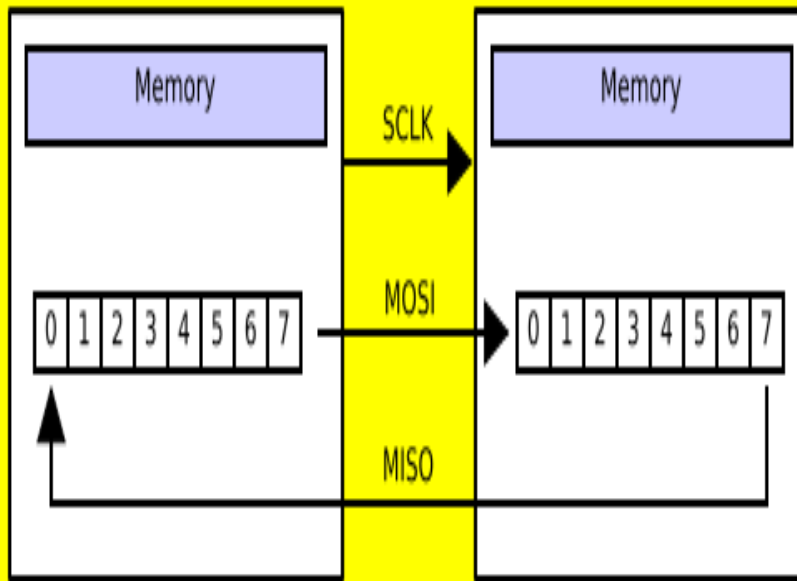
SPI(Serial Peripheral Interface) is a **single master multiple slave serial** transmission protocol that works in **full duplex** mode.



- There can be only one master but there can be multiple slaves.
- MOSI- Master out Slave In
- MISO- Master In Slave Out
- SCLK- Clock Signal
- SS or CS- Slave Select or Chip Select
- The data is transmitted using shift register.
- With the number of clocks equal to the word size(generally 8) the master and slave would have exchanged the value in their shift register.
- They can now process the data they have received or/and can load new data in the shift register.

Master

Slave



Lets Talk Business and Discuss The actual Protocol

- To start off , the the master configures the clock rate of transmission(frequency should be less than the maximum supportable frequency).
- The master then sends a chip select signal 0 on the chip select line which it wants to choose.
- Since the lines are active high so by default they are high(initially).
- Sometimes a waiting time is involved, so the master must wait for that duration before it starts the clock cycle.
- During each SPI clock cycle a full transmission happens with this operation-
 1. The master sends one bit on MOSI line and the slave reads on the same line.
 2. The slave sends one bit on the MISO line and the master reads on the same line.
- We may not need all these transmission for our purpose but as per the protocol this has to happen

GDB

- GDB or GNU debugger is a standard debugger for GNU based OS like Linux.

How to start gdb?

- We compile our program like this

```
cc -g -o program program.c  
gdb program
```

The **g** option helps us to know where the program is failing.

- Now we enter `gdb program`
- How our `gdb` program is running over our program named "program".
- To run a program now with `gdb` debugger we enter the command **run**.
- Whenever the program faults the `gdb` gives the location and reason for failure.

Stacktrace and Examining Variables

- To back trace the call flow of program to get to know from where the problem would have originated in program execution we enter **backtrace** command while running `gdb`.
- We can also know the values of variables in current state when the program had faulted by using **print** command, eg. `print i`; this will give us the value of variable `i` when the program had faulted.
- `gdb` keeps these results in a **pseudo variable** starting with `$1`, `$2` and so.
- To print a number of consecutive items we use `print array[0]@n` , where `n` is the number of items that we want to print and `array` is the array name.

Listing the program and Setting Breakpoints

- When we are within `gdb` and want to view the source code of the program then we use **list** command, this will print the few lines before and few lines after the program faulted while running.
- To display more lines we use `list` command again.
- To set a breakpoint at a particular line we enter these set of commands

```
gdb program  
break lineno  
run
```

- The program stops where we have set the breakpoint and prints the value of variable at that point belonging to that line.
- To continue the program we use **cont** command.

- We can use **display** command to set gdb to display the values of variables whenever the program stops at a breakpoint eg. display k; display array[0]@4.
- To come out of gdb anytime we use **quit** command.
- We can see the breakpoints and displays we have enabled using these commands

info break

info display

- To disable breakpoint and display we use

disable break # (# is the breakpoint number we see using info break command).

disable display # (# is the display we have set using info display command).

Kernel Space and User Space

Understanding of Kernel space and User space in detail is very important if you wish to have a strong base of Linux Kernel.

- Here Kernel Space and User Space corresponds to their Virtual address space.
- Every process in linux utilizes its own separate virtual space.
- In a linux system based on 32 bit Architecture, user space address space corresponds to lower 3GB of virtual space and kernel space the upper 1GB.(general way)
- The kernel space virtual address space is shared between all the processes.
- When a process is active, it can either be running in "user mode" or "kernel mode".
- In a process is running in User mode it means that the CPU is running the user space side of code.
- A process running in the user mode has limited capability and is controlled by a flag in the CPU.
- Even though the kernel memory is present in the process's memory map the user space code is not allowed to access the kernel space code.(can do in some special way).
- When a process wants to do something other than move data around in its own (userspace) virtual memory, like opening a file for example, it must make a syscall to communicate with the kernel space.
- Each CPU architecture has its unique way of making a system call but the basic remains the same i.e.
- A magic instruction is executed, the CPU turns on the "privileged mode" flag, and jumps to a special address in kernel space, the "syscall entry point".(read another post to understand what syscall is)
- Now when the syscall has reached the kernel space then the process is running in kernel mode and executing instructions from the kernel space memory.
- Taking the same example of open system call, to find the requested file, the kernel may consult with filesystem drivers (to figure out where the file is) and block device drivers (to load the necessary blocks from disk) or network device drivers and protocols (to load the file from a remote source).
- These drivers can be either built in or can be loaded as module but the key point that remains it

that they are the part of kernel space.

- Loading a module is done with a syscall that asks the kernel to copy the module's code and data into kernel space and run its initialization code in kernel mode.
- If the kernel can't process the request then the process is made to sleep by the kernel and when the request is complete then the syscall returns back to the user space.
- Returning back to user mode means restoring the CPU registers to what they were before coming to Kernel Mode and changing the CPU privilege level to non-privilege .
- Apart from syscalls there are some other things that take CPU to kernel mode eg.

1. Page faults- If the process tries to access a virtual memory address that doesn't have a physical address assigned to it then the CPU enters the Kernel mode and jumps to page fault handler and the kernel sees whether the virtual address is valid or not and depending upon this it either tries to create a physical page for the given virtual address or if it can't then sends a segmentation fault signal (SIGSEGV).

When the CPU receives some interrupt from the hardware then it jumps to the kernel mode and executes the interrupt handler and when the kernel is finished handling the interrupt the code returns to the user space where it was executing.

STACK AND HEAP

I felt that these two terms are used a lot while discussing linux so a quick look at these two topics would be good.

The stacks and heaps are the part of virtual address space that consists of code;data;heap(growing upwards) and stack portion(growing downwards).(from bottom to top).

STACK:

- This is the memory area used by our thread of execution.
- When a function is called the local variables and return addresses are stored here.
- The space becomes available when the function exits.
- The user doesn't need to take care of the memory allocation and deallocation(happens automatically).

HEAP:

- It is the memory set aside for dynamic allocation.
- Generally, an application is given a separate heap memory and it allocates the memory from the given heap on need basis.
- WE can allocate and free heap anytime.
- User must take care of deallocation of memory when he doesn't need it.
- To get heap memory we use malloc and calloc if we are in user space and if we are in kernel space we use kmalloc, vmalloc.

- We must free the memory allocated by using free, kfree and vfree respectively.

System Calls

In this post we will discuss mainly about what are system calls, why do we need it and how to implement it.

What is a system call ?

To understand this first we would ask ourselves what are the stuffs the OS(read kernel) needs to do ?

- Process Management (starting, running, stopping processes)
- File Management(creating, opening, closing, reading, writing, renaming files)
- Memory Management (allocating, deallocating memory)
- Other stuff (timing, scheduling, network management).

So, system call is an interface through which user space applications request the Kernel to perform the operations listed above.

An example would be , the user space requests to open a device(hardware).

In short we can say that the System call is an interface between user space processes and hardware.

Why do we need system call?

1. It provides an abstraction to the user space process. Eg. open call for user means just open the device, the user doesn't need to care about intricacy of the call.
2. It maintains the system security and stability as the kernel first checks the authenticity of the call before requesting it a service.
3. It helps in virtualization of various processes i.e various processes can use it independently.

System call interface and C library.

The system call interface in Linux, as with most Unix systems, is provided in part by the C library.

We will see How System call works using a example of printf() call in userspace.

Syscalls

- System calls (*syscalls* in Linux) are accessed via function calls. System calls need inputs and also provide a return value (*long*) signifies success or error.(0 generally means success).
- System calls have a defined behavior.

For example, the system call `getpid()` is defined to return an integer that is the current process's PID.

The implementation of this syscall in the kernel is very simple:

```
asm linkage long sys_getpid(void)
{
    return current->tgid;
}
```

Some important observations from this-

A convention in which a system call is appended with sys in kernel space.
asm linkage modifier -tells the compiler that the function should not expect to find any of its arguments in registers (a common optimization), but only on the CPU's stack.

- In Linux, each system call is assigned a *syscall number*. This is a unique number that is used to reference a specific system call.
- When the syscall number is assigned, it cannot be changed or be recycled.
- System calls in Linux are faster than in many other operating systems. (such as fast context switch times).
- The kernel keeps track of all the registered system calls in table `sys_call_table` which is defined in `entry.S` (assembler file) in `arch/arch-name/kernel/`

System Call Handler:-

- Since the system call code lies in kernel side, so to execute it we must switch the processor to kernel mode when system call is executed.
- This is done by issuing a software interrupt.
- In this mechanism an exception is raised and the Kernel switches to kernel mode and execute the system call handler.

- The defined software interrupt on x86 is the **int \$0x80** instruction.
- It triggers a switch to kernel mode and the execution of exception vector 128, which is the system call handler.
- The system call handler function is `system_call()`.
- It is architecture dependent and typically implemented in assembly in **entry.S**
- User space first enters the system call number in `eax` register(X86) and causes the trap.
- The kernel reads the value of the `eax` register and calls the appropriate system call handler.
- The `system_call()` function checks the validity of the given system call number by comparing it to `NR_syscalls`.
- If it is larger than or equal to `NR_syscalls`, the function returns `-ENOSYS`. Otherwise, the specified system call is invoked:

```
call *sys_call_table(%eax,4)
```

- Because each element in the system call table is 32 bits (four bytes), the kernel multiplies the given system call number by four to arrive at its location in the system call table



- Now, the system call is called with some parameters, generally upto 5 parameters, we store the parameters values in registers **ebx**, **ecx**, **edx**, **esi**, and **edi**.
- In some unique cases when 6 or more parameters are passed then a single register is used which stores the pointer to the user space where all the parameters are stored.
- Not only this, even the return value is stored in the the register(`eax` in case of X86).

How to implement system calls?

Adding a system call is an easy task. But it is the implementation that has to be done carefully.

Now we will see what are the steps used to implement a system call.

First we must define its purpose. What is the use of this system call? The syscall should have exactly one purpose.

Next, we must define system call's arguments, return value, and error codes.

The system call should have a clean and simple interface with the smallest number of arguments possible.

- Final Steps in Binding a System Call
 1. First, add an entry to the end of the system call table.
 2. For each architecture supported, the syscall number needs to be defined in `<asm/unistd.h>`.
 3. The syscall needs to be compiled into the kernel image

How system call verifies parameters(arguments)?

- System calls must make sure all of their parameters are valid and legal. Such as *access permission*.
- System calls must carefully verify all their parameters to ensure that they are valid and legal.
- The system call runs in kernel-space, and if the user is able to pass invalid input into the kernel without restraint, the system's security and stability can suffer, in short the kernel can be hacked!!
- For example, for file I/O syscalls, the syscall must check whether the file descriptor is valid. Process-related functions must check whether the provided PID is valid. Every parameter must be checked to ensure it is not just valid and legal, but correct.
- One of the most important checks is the validity of any pointers that the user provides. Imagine if a process could

pass any pointer into the kernel, unchecked, with warts and all, even passing a pointer for which it did not have read access! Processes could then trick the kernel into copying data for which they did not have access permission, such as data belonging to another process. Before following a pointer into user-space, the system must ensure that

1. The pointer points to a region of memory in user-space. Processes must not be able to trick the kernel into reading data in kernel-space on their behalf.
 2. The pointer points to a region of memory in the process's address space. The process must not be able to trick the kernel into reading someone else's data.
 3. If reading, the memory is marked readable. If writing, the memory is marked writable. The process must not be able to bypass memory access restrictions
- Two methods for performing the requisite checks and the desired copy to and from user-space:
 1. For writing into user-space, the method **copy_to_user**(*destination memory address , source pointer , size of the data to copy*) is provided.
 2. For reading from user-space, the method **copy_from_user**(*destination memory address , source pointer, the number from the second parameter reading into the first parameter*) is used.
 - Both of these functions return the number of bytes they failed to copy on error. On success, they return zero. It is standard for the syscall to return -EFAULT in the case of such an error.
 - check is for valid permission. A call to `capable()` with a valid capabilities flag returns nonzero if the caller holds the specified capability and zero otherwise. For example, `capable(CAP_SYS_NICE)` checks whether the caller has the ability to modify nice values of other processes.

Paging and Segmentation

I hope that you know about fragmentation . If not please go through my other post in the same blog about it.

Paging and segmentation are both used by the linux kernel to deal with the problem of external fragmentation.

PAGING:

- Physical memory is broken into fixed sized blocks called frames.
- Logical memory is also broken into same sized blocks called pages.
- Every logical address generated by CPU is divided into two parts : Page number(p) and pageoffset(d).
- The system maintains a page table which has a mapping of logical and physical addresses eg. page 3 in logical memory corresponds to frame number 8 in physical memory.
- When a pointer tries to access a page that is currently not mapped to the physical memory **page fault** occurs.(it's a normal occurrence and not observed by user).

SEGMENTATION:

- Segmentation differs from paging in the sense that in segmentation the blocks of memory(both logical and physical) are of different size.
- The segment table (analogous to the page table in case of paging) keeps two values for each segment 1. Segment base 2. Segment Limit.
- The segment table uses these two values to generate physical addresses.
- If a program tries to access the data beyond the limit specified by segment base and segment limit then the kernel receives a trap called the **segmentation fault** and it can abort the program.

Linux Addressing

- There are two kinds of address used in Linux.
- 1. Virtual address(logical address) and
- 2. Physical address.
- Physical addresses are the addresses of the contents of the RAM.

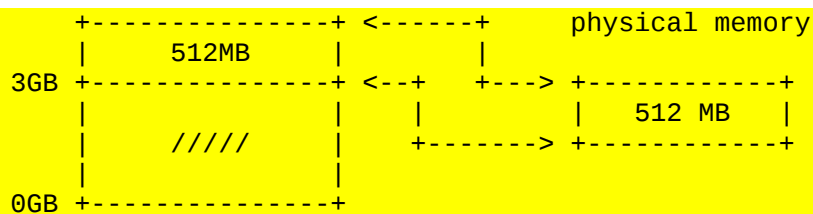
- Virtual addresses as the name says are virtual, i.e. they do not point to any address in RAM directly but need to be converted into physical address by MMU at run time.
- For a 32 bit system there can be $2^{32}=4\text{GB}$ virtual addresses possible.
- Each process is given a separate virtual address space so that they feel as if they have the access to the full RAM and each process remain independent of each other.
- Virtual address is divided into two parts
 1. User space virtual addresses
 2. Kernel space virtual addresses.



- The user space can be given upper 3 GB (0xc0000000 .. 0xffffffff) and kernel space the lower 1 GB (0x0... 0xbfffffff) also but the diagram shows the general way of doing it.
- Virtual memory is not really used (committed) until actually used. This means when you allocate memory, you get an "IOU" or "promise" for memory, but the memory only gets consumed when you actually use the memory, as in write some value to it.
- A process running in user space may need to request some functions from kernel(ex. syscalls), so kernel image is fully mapped to the Kernel Virtual Space.
- Suppose a user space application makes use of libc, then libc should also be mapped to the virtual address space of the process
- This mapping is kept permanent till the process is terminated.
- Since the kernel needs to access each and every physical memory so the entire physical memory must be mapped to the kernel virtual space (1 GB).
- The mapping is one to one.
- For the mapping to be truly one to one the RAM has to be of 1GB, but this is not the case always.
- We will consider two cases 1. RAM is less than 1 GB, say 512MB and 2. when the RAM is more than 1 GB say 3GB.

process address space(virtual address space)

4GB +-----+
 | 512MB |



When the RAM is 512MB then whole of RAM is linearly mapped to the 512MB of the lower 512 MB of the kernel virtual space.

The virtual address space of Kernel is divided into two parts 1. Low memory 2. High Memory.

The first 896MB constitute the low memory region or LOWMEM.

The top 128 MB is called high memory region or HIGHMEM.

The first 16MB of LOWMEM is reserved for DMA usage.

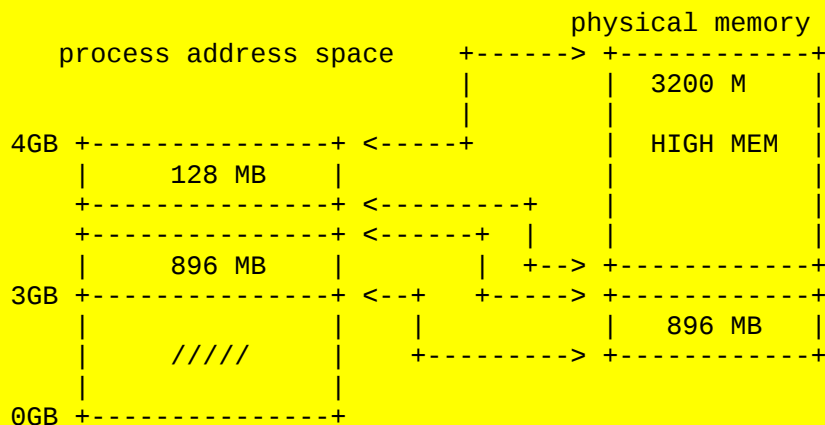
There are some zones of physical memory:

ZONE_DMA - Contains page frames of memory below 16 MB

ZONE_NORMAL - Contains page frames of memory at and above 16 MB and below 896 MB

ZONE_HIGHMEM - Contains page frames of memory at and above 896 MB

So, if you have 512 MB, your ZONE_HIGHMEM will be empty, and ZONE_NORMAL will have 496 MB of physical memory mapped.



As we can see from the diagram the physical memory above 896MB is mapped to the 128MB of the HIGHMEM region.

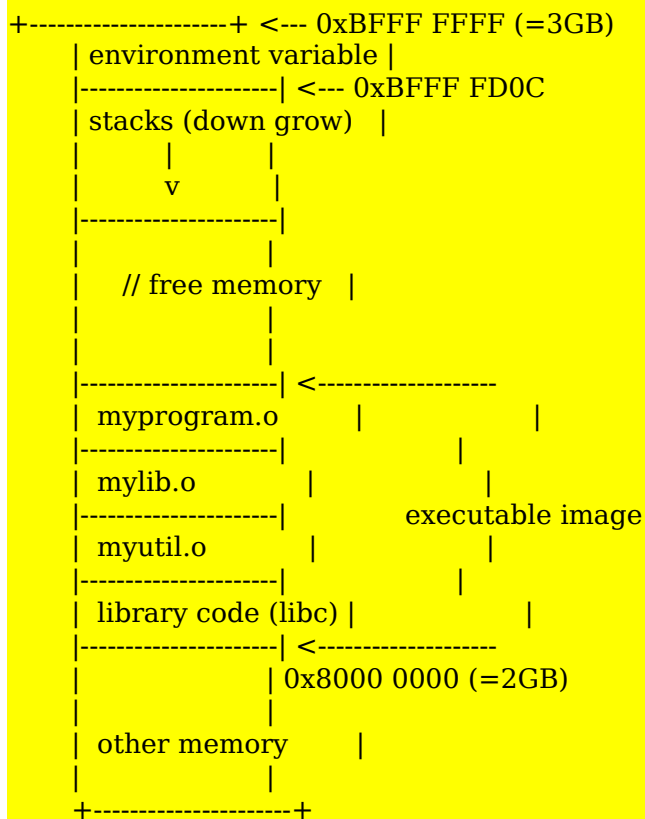
When we request a large chunk of memory from physical space using `vmalloc()` then we get it from HIGHMEM also ,that's why the physical pages that we get are not contiguous.

For a user-space application, if you print out the pointer address you defined, it should be one of the virtual addresses out of the (0-3GB) range.

What about kernel? what if print a pointer address in kernel? is it always from kernel address space? The answer is no ... since kernel can access user space address, depending on the pointer, it can be from either.

To distinguish between kernel space and user space address is easy, if it fall into 0-3GB, then it is from user-space, otherwise, it is from kernel. The programmer sees virtual address only.

To make things clear from user point of view lets see the user space addresses after compiling the program.



PAGE_OFFSET = 0xC8000000 = 3GB

For a 512MB system, the virtual address for kernel will be from 3GB ~ PAGE_OFFSET + 512MB

Basics of Linux Device Drivers

We can define a device driver as a "Specific code that will define the behavior of a device connected to our system" for eg. the driver for touch screen will have code that will define the working of touchscreen and a driver for camera will have all the functioning of camera, similarly for other devices too. [A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.](#)

So the very next question that would come into our mind is-"how does the driver communicate with the device?"

The communication is through the computer bus communication subsystem(say I2C protocol) to which the hardware is connected.

Two more point worth mentioning is that -

1. The drivers are hardware dependent , for eg. the driver for an accelerometer sensor would be drastically different from the drivers for camera as both the devices are used for different purposes.
2. The drivers are operating system specific (OS), for eg. the same camera driver for Linux OS would be considerably different from that for Windows OS as different OS exposes different APIs for middlewares.

Device drivers help high level application programmers to not worry about the specifications of the hardware, they just need to know the API that can be used to communicate to the driver.

For eg. a high-level application for interacting with a serial port may simply have two functions for sending data-func1 and for receiving data func2.. At a lower level, a device driver implementing these functions would communicate to the particular serial port controller installed on a user's computer.

Some drivers get loaded automatically when the system starts and some drivers load when we actually insert the device into the system. Almost all of us have seen the "installing the device software" notification in windows when we insert any new device into our system during runtime.

Linux allows the drivers to be loaded/unloaded on runtime also , thats why the drivers are also known as loadable modules or just modules. Due to this very useful feature Linux OS is used in servers also so that required modules can be loaded and unloaded at runtime and we don't need to restart the system every time.

Drivers can run in both user mode as well as in kernel mode. The main advantage of running in user mode is that if the driver crashes it won't crash the kernel. On the other hand, user/kernel-mode transitions usually impose a considerable performance overhead, thereby prohibiting user-mode drivers for low latency and high throughput requirements.

There are broadly three kinds of device drivers in Linux-

1. **Character Device Drivers**- eg. Keyboard drivers, camera drivers, sensor drivers- these drivers work on devices which transmit data per byte.

2. **Block Device Drivers**- eg. USB, Hard Drive drivers- these drivers work on devices that transmit block of data.

3. **Network Device Drivers**- These are the hardware used by networking hardware, these are similar to block device drivers in a sense that they work on packets but they have some differences too which we will see later.

Since the basics of device drivers is clear so, let's begin by writing a simple loadable module. As we write we will understand the stuffs required to write a device driver.

It is said that to begin any programming chapter we must write a hello world program otherwise something bad might happen to us, so let's start with it.
Hello world program-

```
-----  
#include<linux/init.h>  
#include<linux/module.h>  
  
MODULE_LICENSE("GPL");  
  
static int hello_init(void)  
{  
    printk(KERN_ALERT "Hello world");  
    return 0;  
}  
static void hello_exit(void)  
{  
    printk(KERN_ALERT "Goodbye");  
}  
  
module_init(hello_init);  
module_exit(hello_exit);  
-----
```

Let's write this program in vim editor and save as hello.c.

Now, let's see how can we compile and run this program.

To compile a kernel module we need to create a makefile for the same, and we keep the make file in the same directory as the file else we will have to give the full path of makefile.

Makefile for the code shown above would be

```
-----  
obj-m +=hello.o
```

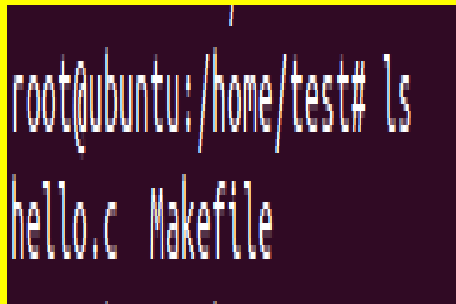
all:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

clean:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

obj-m shows that we want to build this program as a module.



```
root@ubuntu: /home/test# ls
hello.c  Makefile
```

Makefiles are a unique species in its own, in a sense that we have to follow some strict guidelines otherwise it won't work. The two very important considerations that we need to take care of are- 1. while saving the Makefile its name should have **capital M** and not small m. 2. make word after all: and clean: should be shifted by one **tab** from the left end.

The module is compiled and built by giving make command in the same directory (test directory here) as the hello.c and Makefile we can generate hello.ko file. (ko means kernel object).

```
xfwqBxOvKQ5NzDkXO7e//a3kA/rJYuqrHX59p2uaait6/vH9nmGfKqgo8QCwhfgiEvWCtOZ5
AMmw52IoARCC99XwQwf6pNtFhSOdtAZ/qGrO/LCoE6vJP985vLwldPPFuANuOM0lpzS
mScXXjSH5F6Bmcy+C6RoFP7Ua7gRpgF5RxoWD99eziPRfs/So8Ek5Pw+45bqSbxcxad+
3rtS+mezPPXDly5zC67/2bYT/F/VI79hEF4+6+JgDB8/Zp6F6zXyOtOBKMIYC8+EAIBXkK
vLS/sb6Jj72CMQRJBRIRLCQiD6/c+BWOAVpm68aSC0urWjGymrN379FsiT2QJNjA84eH
Cys5bJSWJGbg+NXuu2XDra5lf4PEVHKaiZ5U5Z2gvO+emsDLjX5SXj+8YQk78L0QElav7
4Xst0/3X5CdqJpezjWFGW6P2Dx3dRzv9znFpx3J9dPn7MVa9dNrCMLLqM9cvnPF7zLwH
AKdXJ5w79ji0y/4WZ3xY0ID7w0KFhxWuRK9gXTLedueF3y9IgMmh5EebzZb/MPNR6Tvn5
z8rLzACAuv/sol/GLh61dW9TfvO3tm5+Nsa93/x8zON+
+OkOAPjwCcnWiLa6J1A3wVrZXwvBABIQJCBIQI/FgIMHdTY/Taiu3H6oFQWiECQgSECA
gRECIGRODnQgDLfEWCr9ckfq5+C3sjRECIGBABIQJCBIQI/FsRYG4myu+WJ/
+GPmAIR5vb2+8Z02ezIX+DcLxkwEqN3X7ooO3Yru8X82ogvC5EQliAEAEhAkIEfkUE/gc0
J1jzoLTvXwAAAABJR5ErkJggg==">
```

we can see that hello.ko file got generated.

```
root@ubuntu:/home/test# ls
hello.c  hello.mod.c  hello.o  modules.order
hello.ko  hello.mod.o  Makefile  Module.symvers
```

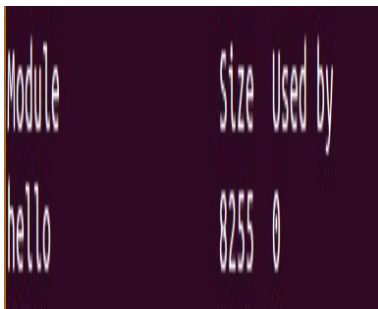
Lets try to understand the structure of this program-

1. Like all other programs that we have written, here we need to include some headers. These header files live in **/include/linux** folder. `init.h` header file is used to start the `init` process while `module.h` file is used to add module capability to this program.
2. The next line is used to specify the license that we are using for this program- it's GPL (GNU Public License).
3. The given program is written in user space. To insert this program into kernel space as a module we give `insmod` command.
4. The command would look like `insmod hello.ko` (`hello.ko` is the kernel object file gerenrated when we compile this program as a module).

```
root@ubuntu:/home/test# insmod hello.ko
root@ubuntu:/home/test# lsmod
Module                  Size  Used by
hello                   1636  0

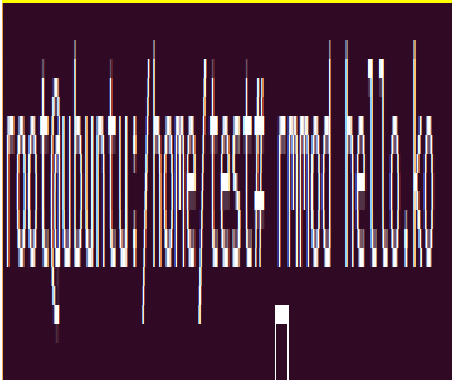
```

After inserting the module in Kernel space we can see if it is present or not by giving `lsmod` command. It is showing in my system-



| Module | Size | Used by |
|--------|------|---------|
| hello | 8255 | 0 |

5. Similarly `rmmod hello.ko` command is used to remove the module from kernel space.



If I perform `lsmod` command I don't see hello modules listed there.

6. When we give `insmod hello.ko` command the initialization function `hello_init` gets called and our module gets loaded into kernel space and when we give `rmmod hello.ko` command the exit function `hello_exit` gets called and the module is removed from the kernel space.

7. `printk` is the kernel space equivalent of user space function `printf`.

8. `KERN_ALERT` is a type of kernel log levels(higher the log level more logs will appear while the program runs). We will see other log levels in advanced topic.

You must be wondering where did those logs go that were supposed to get printed when we insert and remove modules. As I told you those are kernel logs that's why they get stored in Kernel log buffer which you can see by giving `dmesg` command.



```
[ 9995.705614] Hello world
```

It is advisable for you to perform all these steps on your linux system.

Initialization and Shutdown Function

Lets delve more into these initialization and shutdown functions.

The module initialization function registers any facility offered by the module.

The format of initialization function is:

```
static int __init initialization_function(void)
{
    /* Initialization code here */
}
module_init(initialization_function);
```

There are some points to be noted about the initialization function:-

1. Initialization functions should be declared static, since they are not meant to be visible outside the specific file.
2. The `__init` token is a hint to the kernel that the given function is used only at initialization time. The module loader drops the initialization function after the module is loaded, making its memory available for other uses
3. There is a similar tag (`__initdata`), this is for data used only during initialization.
4. Use of `__init` and `__initdata` is optional, but it is worth the trouble.

Modules can register many different types of facilities, including different kinds of devices, filesystems, cryptographic transforms, and more. For each facility, there is a specific kernel function that accomplishes this registration. The arguments passed to the kernel registration functions are usually pointers to data structures describing the new facility and the name of the facility being registered. The data structure usually contains pointers to module functions, which is how functions in the module body get called.

Lets understand this using one opensource file for a camera driver.

<https://android.googlesource.com/kernel/samsung/+529a92f923d124b02c1b2011313b11e2f82a6960/drivers/media/video/s5k4ecgx.c>

Lets trace the order in which the functions will be called.

```
1. module_init(v4l2_i2c_drv_init);
```

This function is called when the system boots. The kernel calls this function for all the

modules present. We can manually call this by giving insmod command in userspace(which we have previously understood).

```
1. static int __init v4l2_i2c_drv_init(void)

2. {

3. return i2c_add_driver(&v4l2_i2c_driver);

4. }
```

Here we can see that inside the initialization function we are adding our camera sensor as an I2C client(I2C is a communication protocol). Below we can see the structure v4l2_i2c_driver.

```
1. static struct i2c_driver v4l2_i2c_driver = {

2. .driver.name = S5K4ECGX_DRIVER_NAME,

3. .probe = s5k4ecgx_probe,

4. .remove = s5k4ecgx_remove,

5. .id_table = s5k4ecgx_id,

6. };
```

These facilities get registered in the kernel.

After all these steps for module_init is completed then we can say that our driver has been registered with Kernel.

There is special file in the Linux kernel called Device Tree file. This file contains the list of all the devices present in the system in hierarchical manner. We will understand the device tree file and I2C protocol in detail in next chapters.

During the kernel bootup after the modules are registered the kernel traverses the device tree file and if it sees the presence of a device with the same name as driver which we have already registered then the probe function gets called.

In the probe function (from the opensource code provided above) you can see that we are basically doing 2 things 1. getting platform data 2. Registering our camera driver as a V4L2 subdevice(V4L2 is a video capture and output API and driver framework for Linux Kernel).

Similarly we have exit/cleanup function which does the reverse of what we have done in initialization function.

If our module doesn't define cleanup function then the kernel won't allow it to be unloaded from the system.

This function also has a fixed format and looks like

```
static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}
```

```
module_exit(cleanup_function);
```

Some important points to be noted for cleanup function

1. Its return value is 0.
2. __exit modifier means that the code is used only for exit purpose of modules.

Advanced topic:

Virtual Device Drivers

Virtual device drivers are used where there is no physical hardware present to perform that operation but we need to give the OS a feel that the hardware is there. The best example would be IPC driver (inter processor communication). When we run an OS in virtualised environment then also we use virtual device drivers. For example a virtual network adapter is used with a virtual private network, while a virtual disk device is used with iSCSI. One more good example for virtual device drivers can be Daemon Tools.

8 logs level for printk function

KERN_EMERG : This log level gets used only for emergency purpose.Ex: process stops.

KERN_ALERT: To attract the attention.

KERN_ERR :To display the error.

KERN_CRIT : To told about Critical serious hardware or software problem.

KERN_WARNING :warning message of danger.

KERN_NOTICE : Simple notice message.

KERN_INFO : Provide Information

KERN_DEBUG :To see log messages during debugging.

Passing Parameters to Linux Device Drivers

- In this section we will see how we can pass an argument to our module.,
- Parameters are declared with the `module_param` macro, which is defined in `moduleparam.h`.
- `module_param` takes three parameters: the name of the variable, its type, and a permission given to that parameter.
- The macro should be placed outside of any function and is typically found near the head of the source file.

This will be more clear if we consider a program-

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/moduleparam.h>

MODULE_LICENSE("GPL");

int param;
module_param(param, int,S_IRUSR|S_IWUSR); //line 6

static int param_init(void)
{
    printk( "displaying the parameter");
    printk("the value of parameter is
%d",param);
    return 0;
}
static void param_exit(void)
{
    printk("exiting");
}
module_init(param_init);
module_exit(param_exit);
```


A closer look at the program tells that it is similar to the previous program that we have discussed except for the lines in red. param is the parameter that we want to pass to this kernel module. In line number 6 we can see that we are passing the parameter param, its type is int and it has been given permission as Read and Write both. To understand the permissions in detail please read the advanced topic at the end of this section.

```
9QfmWjMajmL0RDziCThogPQfaCKVJErhOxecMilJeASqOXLDuZ3nKqTbDL30g8XWY3FVVv
7FcmFIKLGM2bbXJw96OqVgfKpn/teJzE27KSWmSggRoFKZWY/wiEdEQb6BQM7c+eWj0w/v
halztPy9F+47+PXXKcpTqF5eUdaiZ4v0V13MVRLEJgCQxHhLPliYL4gAAIdFuBxjRQc8t9fENZp/
HWLoLUZSfmIFajv+6vmJzJuyCgGZhbaCFPk3LryFUJ2YjVI9YDaUmtDtv8IXIqCsoQxqDB6tT4
GpEHS5zNIuvfSr3oGFLVlHbqmPHZhJXpKGrB+XTxtY22hpJboSqfOZ3eOjJ15zz3t2v/2b53z905r
rHlZyMqK8hc4yqhhIFqRQ8ShGs2nqEoa4idGPi7cvk54jF5M/U9r8oEUqYlLcEp6IyaoIVk3c8j+2
cMGpCzIrd/d3Dkc55hjmr+W6T9NEBvv5XwfLXVafW9q6Gad03X+9QQKbyeZ7/h0bQYpQ8wszS
jpl2482+N2CsYCCOBVSAAArIVEPsID5535eAJ16M/HdvB8Y/OoHyy6HsX44Kz06+S9W4i78qh
YNejXqcCqf6RsXmsPqP7lkhj7Yj7X/LpVGLPGm/vqrCbBRytUVrkQyOtQ8arXjwuQRwcli0uv1v
Vux/1wR9ncupfXznwi8sJj7Bg9cMR0U9L2AwTY92KC2Gxr0RmksbucAmtUNUvfA469//7h5FhW
fnIFjeL8yF7Nlyetuly0wEcr8y+m494OK10Kr1Z1EuX9uBcRDZ/COzXj5l1LhO3etjsjvuPqm2o+uh
cOJN3kcIpiA9LxfYttX5wJSxbKLCRK9w9lBKZHM5qz2GVsYsuMYrmbH+TclEzJe4L0gNS8mt
QzU+1+7IC1Xv1kr876KroZoiURpuM75Pccy1V42/Df5y+hDn6FOOhkjJmLkzVj+SeL0nfkiwBgR
AQHYC4svUREWyl52nX5bx2n0HzuxZMuR58Nx5frEVvPM3ojJ53Vy37UlqC338L4QcOrlE7cX
9+5k1DafInLzgVWv2PeznvDvgcujhkMW9c1IeZnw2Onuue7Jr0+/Jfex+CTpwar2NmT7v5WOi6s
GPc5ZvuFL31cptp08cPhfw/QRzAWT1RfJHbCtUeYzH9pUqN77z/iufPNXnFp3ftveO0rQDq0eoNh
eBWM/9fwi+TbXyP3QgdP23YxsIpfLdAAABv0lEQVTCaPgQpbF+K8Of6i3yiTp5KGzd9JE65IV
Lw4f7JnL/5RIT1wOzdYXfLcOVTT1990XtXzWh7LzrAu+L//GGXP8i3H5VaNqgJSd/
+/Wfc6F/rDYuzkhl1rRTJHm3VvwYRf6zq6EagqDqW3vbamT8GvLY8K4MpzI3m7dEWVv4gktk9
LAQBECgcwXQEUqDO3cPPap3Sh/rn89EzawIXLpsE+/xU96H2n9hwk2vUrcJ025I9TJaj4JpMxi
UYbw6NHyn9rkp0/3i27w9RlGf4BUSPSfd1vKnGw1nFfABARD4oAJii0IfNKqmnaN0rf8bqMkQ
vmTB6wtf5Lzp+ld5uaVXvb/z0gzZG3K4wMox4IVg1fvDmXUrKEzVZFXg0Z3DMjbbBiYI5gCK
1tTfL/vOQBK3OPnGQA74cD8Y2DMItBL03mmAbuxw6fwCXREzVhXk+41XquhH+EVsLrtFRB
0z6Ds75nSDtNzukgPIwXUrKEzZwLRX8ga7LYeftU3U3NJ7vm527qnMonq4DpDdjxJ6AoH3EoC
i0HvxQWMQAAEQkHcB4XqLvI8I4gcBEAABEJBCANKAFFiwKQiAAAj0PIH/ByfL0HCiWTaI
AAAAAEIFTkSuQmCC">
```

Like previously done we will write one Makefile in the same directory in which directory this file resides and build this file as a kernel object. Now we will pass the parameter while inserting the module itself.

```
sudo insmod test.ko param=10
```

If we use `dmesg` then the output will be displaying the parameter the value of parameter is 10

Lets see a self explanatory program on passing an array to the module

```

#include<linux/init.h>
#include<linux/module.h>
#include<linux/moduleparam.h>

MODULE_LICENSE("GPL");

int paramArray[3];
module_param_array(paramArray, int, NULL, S_IWUSR|S_IRUSR); /*Counter records
how many parameter is passed.In our program we have not used that feature so
setting its value as 0.*/

static int array_init(void)
{
    printk("Parameter array");
    printk("Array elements are
    :%d\t%d\t%d",paramArray[0],paramArray[1], paramArray[2]);
    return 0;
}
static void array_exit(void)
{
    printk("Exiting ");
}
module_init(array_init);

module_exit(array_exit);

sudo insmod testarray.ko paramArray=1,2,3

When we do dmesg we can see the output on screen as 1 2 3

```

poll()

poll() is a userspace function which provides application to monitor I/O events on a set of file descriptors.

```

#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);// polls over an array of structures

```

The `<poll.h>` header shall define the **pollfd** structure, which shall include at least the following members:

int fd The following descriptor being polled.

short events The input event flags

short revents The output event flags

nfds means the number of file descriptors on which polling is being done.

Timeout means the poll function will wait for this many milliseconds for the event to occur.

If the value of timeout is 0 then the function return immediately.

If the value of timeout is -1 then poll shall block until a requested event has occurred or until the call is interrupted.

The poll function supports for regular files, terminal and pseudo-terminal devices, FIFOs, pipes, sockets and STREAMS-based files.

The **flags** used for events and revents are-

POLLIN

Data other than high-priority data may be read without blocking.

POLLRDNORM

Normal data may be read without blocking.

POLLRDBAND

Priority data may be read without blocking.

POLLPRI

High priority data may be read without blocking.

POLLOUT

Normal data may be written without blocking.

POLLWRNORM

Equivalent to POLLOUT.

POLLWRBAND

Priority data may be written.

POLLERR

An error has occurred (*revents* only).

POLLHUP

Device has been disconnected (*revents* only).

POLLNVAL

Invalid *fd* member (*revents* only).

Return value of poll function

On successful return the poll functions returns a positive integer value which means the number of fds selected. 0 means no fd selected and the call timed out.

Upon failure, *poll()* shall return -1 and set *errno* to indicate the error.

ERRORS REPORTED:

[EAGAIN]

The allocation of internal data structures failed but a subsequent request may succeed.

[EINTR]

A signal was caught during *poll()*.

[EINVAL]

The *nfds* argument is greater than {OPEN_MAX}, or one of the *fd* members refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

EXAMPLE

Checking for Events on a Stream

The following example opens a pair of STREAMS devices and then waits for either one to become writable. This example proceeds as follows:

1. Sets the *timeout* parameter to 500 milliseconds.
2. Opens the STREAMS devices **/dev/dev0** and **/dev/dev1**, and then polls them, specifying POLLOUT and POLLWRBAND as the events of interest.
The STREAMS device names **/dev/dev0** and **/dev/dev1** are only examples of how STREAMS devices can be named;
3. Uses the *ret* variable to determine whether an event has occurred on either of the two STREAMS. The *poll()* function is given 500 milliseconds to wait for an event to occur (if it has not occurred prior to the *poll()* call).
4. Checks the returned value of *ret*. If a positive value is returned, one of the following can be done:
 - a. Priority data can be written to the open STREAM on priority bands greater than 0, because the POLLWRBAND event occurred on the open STREAM (*fds*[0] or *fds*[1]).
 - b. Data can be written to the open STREAM on priority-band 0, because the POLLOUT event occurred on the open STREAM (*fds*[0] or *fds*[1]).
5. If the returned value is not a positive value, permission to write data to the open STREAM (on any priority band) is denied.
6. If the POLLHUP event occurs on the open STREAM (*fds*[0] or *fds*[1]), the device on the open STREAM has disconnected.

```
#include <stropts.h>
```

```
#include <poll.h>
```

```
...
```

```
struct pollfd fds[2];
```

```
int timeout_msecs = 500;
```

```
int ret;
```

```
    int i;
```

```
/* Open STREAMS device. */
```

```
fds[0].fd = open("/dev/dev0", ...);
```

```
fds[1].fd = open("/dev/dev1", ...);
```

```
fds[0].events = POLLOUT | POLLWRBAND;
```

```
fds[1].events = POLLOUT | POLLWRBAND;
```

```
ret = poll(fds, 2, timeout_msecs);
```

```

if (ret > 0) {
    /* An event on one of the fds has occurred. */
    for (i=0; i<2; i++) {
        if (fds[i].revents & POLLWRBAND) {
            /* Priority data may be written on device number i. */
...
        }
        if (fds[i].revents & POLLOUT) {
            /* Data may be written on device number i. */
...
        }
        if (fds[i].revents & POLLHUP) {
            /* A hangup has occurred on device number i. */
...
        }
    }
}

```

There is one important variant of poll function known as poll_wait

The system uses a **poll_wait** call to indicate that the **poll** system is interested in events. The **poll_wait** call includes a reference to a wait queue that must be triggered by a driver event. Another argument to **poll_wait** function is a poll_table structure.

What exactly happens is that on calling poll_wait() the kernel calls all the fops->poll on all associated fds, passing the the global poll table.

poll_wait() adds the process' wait_queue(events its is waiting for) to this global poll table. Or simply, a process adds itself to all the wait queues it is dependent on, and on poll/poll_wait invocation the kernel checks the global poll table if the event has occurred. If the event is not ready (blocked on I/O, device not ready etc), it puts the process to sleep. Once an event has occurred (say read on FD), it is removed from all wait queues and all the wake handlers associated with the queue are called waking up the process. So from the outside it'll look like the poll_wait() block until an event has occurred, but actually the event triggers the wakeup of the process.

mmap() and munmap()

In computing, **mmap(2)** is a POSIX-compliant Unix system call that maps files or devices into memory.

- It is a method of **memory-mapped file I/O**.
- It naturally implements **demand paging**, because initially file contents are not entirely read from disk and do not use physical RAM at all.
- The actual reads from disk are performed in a "lazy" manner, after a specific location is accessed. After the memory is no longer needed it is important to munmap(2) the pointers to it.
- A **memory-mapped file** is a segment of **virtual memory** which has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource.

- This resource is typically a file that is physically present on-disk, but can also be a device, shared memory object, or other resource that the operating system can reference through a file descriptor.
- Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory.
- Memory mapping is the only way to transfer data between user and kernel spaces that does not involve explicit copying, and is the fastest way to handle large amounts of data.
- There is a major difference between the conventional read(2) and write(2) functions and mmap.
- While data is transferred with mmap no "control" messages are exchanged.
- This means that a user space process can put data into the memory, but that the kernel does not know that new data is available. The same holds for the opposite scenario: The kernel puts its data into the shared memory, but the user space process does not get a notification of this event.
- This characteristic implies that memory mapping has to be used with some other communication means that transfers control messages, or that the shared memory needs to be checked in regular intervals for new content.
- Similar to the read and write function calls mmap can be used with different file systems and with sockets.

From Linux Man page-

```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot, int flags,
```

```
int fd, off_t offset);
```

```
int munmap(void *addr, size_t length);
```

mmap() creates a new mapping in the virtual address space of the

calling process. The starting address for the new mapping is

specified in *addr*. The *length* argument specifies the length of the

mapping.

If *addr* is NULL, then the kernel chooses the address at which to

create the mapping; this is the most portable method of creating a

new mapping. If *addr* is not NULL, then the kernel takes it as a hint

about where to place the mapping; on Linux, the mapping will be

created at a nearby page boundary. The address of the new mapping is

returned as the result of the call.

The contents of a file mapping are initialized using *length* bytes starting

at offset *offset* in the file (or other object) referred to by the

file descriptor *fd*. *offset* must be a multiple of the page size as

returned by *sysconf(_SC_PAGE_SIZE)*.

The *prot* argument describes the desired memory protection of the

mapping (and must not conflict with the open mode of the file). It

is either **PROT_NONE** or the bitwise OR of one or more of the following

flags:

PROT_EXEC Pages may be executed.

PROT_READ Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE Pages may not be accessed.

The *flags* argument determines whether updates to the mapping are

visible to other processes mapping the same region, and whether

updates are carried through to the underlying file. This behavior is

determined by including exactly one of the following values in *flags*:

MAP_SHARED Share this mapping. Updates to the mapping are visible to

other processes that map this file, and are carried

through to the underlying file. The file may not actually

be updated until [msync\(2\)](#) or **munmap()** is called.

MAP_PRIVATE

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap()** call are visible in the mapped region.

In addition, zero or more of the following values can be ORed in

flags:

MAP_32BIT (since Linux 2.4.20, 2.6)

Put the mapping into the first 2 Gigabytes of the process address space. This flag is supported only on x86-64, for 64-bit programs.

MAP_ANON

Synonym for **MAP_ANONYMOUS**. Deprecated.

MAP_ANONYMOUS

The mapping is not backed by any file; its contents are initialized to zero. The *fd* and *offset* arguments are ignored;

MAP_DENYWRITE

This flag is ignored.

MAP_EXECUTABLE

This flag is ignored.

MAP_FILE

Compatibility flag. Ignored.

MAP_FIXED

Because requiring a

fixed address for a mapping is less portable, the use of this option is discouraged.

MAP_GROWSDOWN

Used for stacks. Indicates to the kernel virtual memory system that the mapping should extend downward in memory.

MAP_HUGETLB (since Linux 2.6.32)

Allocate the mapping using "huge pages." See the Linux kernel source file [Documentation/vm/hugetlbpage.txt](#) for further information.

MAP_LOCKED (since Linux 2.5.37)

Lock the pages of the mapped region into memory in the manner

of [mlock\(2\)](#). This flag is ignored in older kernels.

MAP_NONBLOCK (since Linux 2.5.46)

Only meaningful in conjunction with **MAP_POPULATE**. Don't perform read-ahead: create page tables entries only for pages that are already present in RAM.

MAP_NORESERVE

Do not reserve swap space for this mapping. When swap space is reserved, one has the guarantee that it is possible to modify the mapping. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available.

MAP_POPULATE (since Linux 2.5.46)

Populate (prefault) page tables for a mapping. For a file mapping, this causes read-ahead on the file. Later accesses to the mapping will not be blocked by page faults.

MAP_POPULATE is supported for private mappings only since Linux 2.6.23.

MAP_STACK (since Linux 2.6.27)

Allocate the mapping at an address suitable for a process or thread stack. This flag is currently a no-op, but is used in the glibc threading implementation so that if some architectures require special treatment for stack allocations, support can later be transparently implemented for glibc.

MAP_UNINITIALIZED (since Linux 2.6.33)

Don't clear anonymous pages. This flag is intended to improve performance on embedded devices. This flag is honored only if **CONFIG_MMAP_ALLOW_UNINITIALIZED** option. Because of the security implications, that option is normally enabled only on embedded devices (i.e., devices where one has complete control of the contents of user memory).

Of the above flags, only **MAP_FIXED** is specified in POSIX.1-2001.

However, most systems also support **MAP_ANONYMOUS** (or its synonym **MAP_ANON**).

Some systems document the additional flags **MAP_AUTOGROW**, **MAP_AUTORES**, **MAP_COPY**, and **MAP_LOCAL**.

A file is mapped in multiples of the page size. For a file that is

not a multiple of the page size, the remaining memory is zeroed when

mapped, and writes to that region are not written out to the file.

on the pages that correspond to added or removed regions of the file

The effect of changing the size of the underlying file of a mapping

is unspecified.

Munmap()

The **munmap()** system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also hand, closing the file descriptor does not unmap the region. automatically unmapped when the process is terminated. On the other The address *addr* must be a multiple of the page size. All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate **SIGSEGV**. It is not an error if the indicated range does not contain any mapped pages.

ARM Architecture

- It is a spinoff from a UK based company Acorn Computers.
- Its name has changes from Acorn RISC Machine to Advanced RISC Machine and now simply ARM.
- ARM doesnt produce Silicon, it just provides RISC processor core designs(Physical IPs).
- There are two category of ARM processors- Embedded Cortex Processor, Application Cortex Processors.
- Embedded Cortex Processors consist of -Microprocessors and Real Time Processors.
- Application Cortex Processor consists of 32 bit [ARM Cortex-A5](#), [ARM Cortex-A7](#), [ARM Cortex-A8](#), [ARM Cortex-A9](#), [ARM Cortex-A12](#), [ARM Cortex-A15](#), ARM Cortex-A17 MPCore. They need an platform OS(linux), has extended instruction set and a good memory management technique.Newer cortex A series processors supports multicore.
- he **ARM Cortex-M** is a group of 32-bit RISC ARM processor cores licensed by [ARM Holdings](#). The cores are intended for microcontroller use, and consist of the Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 .
- The **ARM Cortex-R** is a group of 32-bit RISC ARM processor cores licensed by [ARM Holdings](#). The cores are intended for robust real-time use, and consists of the Cortex-R4, Cortex-R5, Cortex-R7.

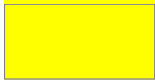
Von Neuman and Harvard Architecture

Before we move ahead we must look at these two architectures as some ARM architecture use one and some the other one.

In Von Neuman Architecture the Instruction(Program) and Data both reside in the same memory segment whereas in Harvard Architecture both reside separately. In Von Neuman Architecture we

can get either instruction or data to the cache(which will ultimately goto the ALU) in one clock cycle whereas in Harvard Architecture we can get both the instruction and data to the respective caches in the same clock cycle itself. This is achieved because in the idle time the address and data bus keeps on filling the caches so that when the processing unit requests for it , it can be given. ARM v7 uses Von Neuman Architecture whereas ARM 9 uses Harvard Architecture.

Snooping



Snooping is a cache coherency protocol used in ARM systems with multiple cores and thus multiple caches. I will first explain the problem that can occur because of this setup.

Suppose the same data , say X from the main memory is there in two caches(of different cores say core 1 and core 4). Suppose the core 1 performs some operation on data X and it gets modified to Y. But the cache of core 4 will still have data X i.e it doesn't have updated data. So, if the core 4 needs to perform some operation then it will do it in the old data and this is undesirable. The diagram below will show that how different caches do snooping on data being requested by other cache from the memory. If the data is being requested from the same memory location from where the other cache has data then cache hit happens and the other cache transfers the data to the cache which is requesting from the memory.

Fast Burst Access

The memory is divided into horizontal and vertical array of cells. For the addresses in same vertical line we have Row access and in different vertical line we have Column Access. If the addresses are in the same vertical line then out of 32 bits of the address only 16 bits has to be changed and the other 16 bit is constant.

This makes access faster and its called fast burst mode.

Static Memory and Dynamic Memory

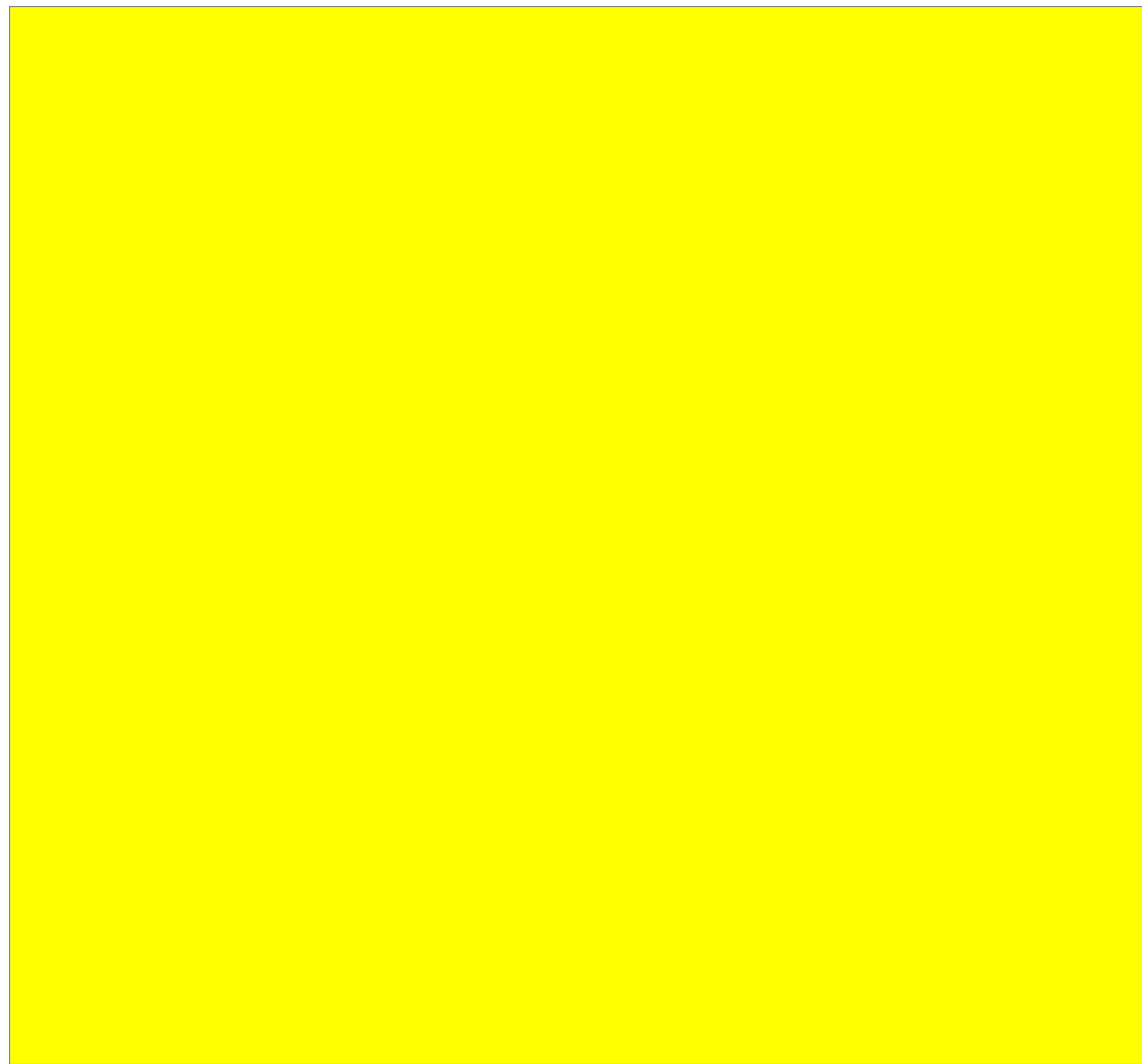
Static memories are those memories which don't need to be refreshed regularly whereas dynamic memories need to be refreshed after every 14ns then only they keep data. To store the same data static memory uses 10 times more transistors as compared to dynamic memories.

It has 37 registers (31 normal registers and 6 status registers). Barrel shifter can multiply the second operand by shifting the bits. It has 32 8 bit multipliers also for multiplication operations.

ARM Register Sets

There are 37 registers in total. We can see that when the core enters abort mode from user mode then the registers of abort mode are swapped in and those of user mode are swapped out. The contents of cpsr is copied into spsr and when the mode is exited then the contents of spsr will be copied into cpsr.

Explanation of Modes



FIQ uses separate registers as it is used to handle very fast interrupts(very high priority) so we want the current registers to not get disturbed so it uses separate set of registers.

Thumb and Thumb2 Instructions

Instruction Format

Saturated Math

Priority Of instruction

1. Reset
2. Data Abort
3. FIQ
4. IRQ
5. Prefetch Abort
6. Undefined

Some Key Points to be noted:-\

Abort happens during instruction prefetch.

PC counts the fetched state out of fetch decode and execute cycle.

Exception can be handled only in ARM mode and not in thumb mode.

Thumb only sees from R0-R7, though there are some provisions to use it beyond that.

Since there is only 1 LR so internal branching is not possible.

Cache is organised as 16 bit so we need to take full 16 bits even for a single change.

for SWI there are 24 bits.

Thumb2 is a superset of ARM and thumb.

Cortex A has MMU.

Cortex R has MPU(subset of MMU).

Cortex M has optional MPU.

P is proportional to frequency.

Cache can use logical or physical add. space.

Timers and Time Management

When do we need absolute time and when do we need relative time ?

Absolute time is used by the kernel to know the exact time of the day whereas relative time is used by kernel when we need schedule some event for the future(after a fixed time in future, say 10 seconds).For absolute time we have an hardware called RTC(real time clock) whereas to manage relative time we have the hardware called system timer.

How does the Kernel synchronize its timing of events?

To help kernel we have our friend System Timer Hardware. The system timer is preprogrammed to work at a certain frequency. In case of ARM and X86 it is 100 HZ. It means that the system timer issues 100 interrupts in one second to the CPU. These interrupts are also called ticks. We can get the value of system up time by knowing the value number of ticks received by the processor till now.The kernel also knows the frequency of system timer, so it also knows the interval in which it is receiving system timer interrupts. This gives the kernel the fair idea of the time and it helps it to sync events.

What are Jiffies?

They are the global variables that hold the number of ticks that have occurred since the system booted. On boot, the system initializes it to zero and is incremented on receipt of each system timer interrupt. Since there are Hz timer interrupt in a second therefore there are Hz jiffies incremented per second of system time.

- The jiffies variable is declared in `<linux/jiffies.h>`
- `extern unsigned long volatile jiffies;`
- Conversion formula--- $\text{jiffies} = \text{seconds} * \text{Hz}$.

How jiffies are stored internally?

- The jiffies variable has always been an unsigned long, and therefore 32 bits in size on 32-bit architectures and 64-bits on 64-bit architectures.
- With a tick rate of 100, a 32-bit jiffies variable would overflow in about 497 days.
- With HZ increased to 1000, however, that overflow now occurs in just 49.7 days!
- If jiffies were stored in a 64-bit variable on all architectures,

then for any reasonable HZ value the jiffies variable would never overflow in anyone's lifetime.

- A second variable is also defined in `<linux/jiffies.h>`:
 `extern u64 jiffies_64;`
- `jiffies = jiffies_64;`
- `jiffies` variable becomes the lower 32 bit of `jiffies_64` variable
- by accessing the lower 32 bits of the `jiffies_64` variable we get the value of our `jiffies`.
- The function `get_jiffies_64()` can be used to read the full 64-bit value though such a need is rare.
- On 64-bit architectures, `jiffies_64` and `jiffies` refer to the same thing. Code can either read `jiffies` or call `get_jiffies_64()` as both actions have the same effect.
- For a 32-bit unsigned integer, the maximum value of jiffy count is $2^{32} - 1$. Thus, a possible 4294967295 timer ticks can occur before the tick count overflows.
- When the tick count is equal to this maximum and it is incremented, it wraps around to zero.

Linux Memory Management

PAGES:

- The physical pages are the basic unit of memory management for the Kernel.
- The MMU(memory management unit) manages the memory in terms of page sizes.
- Generally a 32 bit architecture has 4KB page size and a 64 bit architecture has 8KB page size.
- Kernel stores info about these pages(physical pages) in its structure `struct page`.
- This structure is defined in `<linux/mm.h>`.

```

struct page {
    page_flags_t    flags;
    atomic_t        _count;
    atomic_t        _mapcount;
    unsigned long    private;
    struct address_space *mapping;
    pgoff_t          index;
    struct list_head lru;
    void             *virtual;
};

```

- Some important fields are-
 1. The **flags** field stores the status of the page. Such flags include whether the page is dirty(it has been modified) or whether it is locked in memory. There are 32 different flags available. The flag values are defined in **<linux/page-flags.h>**.
 2. **_count** field means how many instance virtual pages are there for the given physical page. When the value of _count reaches zero that means noone is using the page at current.
 3. **virtual** this is the address of the page in virtual memory. For highmem(highmemory >896MB of virtual memory) this field is zero.
- The goal of this data structure is to describe the physical pages and not the data contained in that page.

ZONES:

- The kernel divides its 1GB virtual address space into three zones -**ZONE_DMA(<16MB)**, **ZONE_NORMAL(16MB-896MB)** and **ZONE_HIGHMEM(>896MB)**.
- Kernel groups pages with similar properties into separate zones.
- The zones have no physical relevance, it has just logical relevance.
- Each zone is represented by **struct zone**, which is defined in **<linux/mmzone.h>**:
- For more details on ZONES , read my other post on linux addressing.

GETTING PAGES:

- Kernel allows us with some interfaces to allocate and free memory within kernel space.
- All these interfaces allocate memory with page-sized granularity and are declared in **<linux/gfp.h>**.
- We can either allocate physical contiguous memory or only virtual contiguous memory.
- One should never attempt to allocate memory for userspace from the kernel - this is a huge violation of the kernel's abstraction layering.
- Instead have userspace mmap pages owned by your driver directly into its address space or have userspace ask how much space it needs. Userspace allocates, then grabs the memory from the kernel.
- There no way to allocate contiguous physical memory from userspace in linux.
- This is because a user space program has no way of controlling or even knowing if the underlying memory is contiguous or not.
- The core function is **struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)**
- This allocates 2^{order} (that is, $1 \leq \text{order}$) contiguous physical pages and returns a pointer to the first page's **page** structure; on error it returns **NULL**.
- To convert a given page(physical) to its logical address we can use the function-**void * page_address(struct page *page)**.
- This function returns a pointer to the logical address where our allocated physical pages resides.
- If we just need the virtual address of the pages(we don't need page structure) we can use the function -**unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)**.
- The pages thus obtained are contiguous in virtual space.
- This function also uses the core function **alloc_pages**, but it directly gives us the starting address of the first page.
- If we just need a single page(order 0 then we have two functions, one for physical and other for logical-

1. **struct page * alloc_page(unsigned int gfp_mask)**

2. **unsigned long __get_free_page(unsigned int gfp_mask)**

- If we need page filled with zero(for security issues we want to initialize memory with all zeros so that if we need to pass this memory to user space then the user space will get access to the contents written on this memory location previously) we can use this function **unsigned long get_zeroed_page(unsigned int gfp_mask)**
- This function works the same as **__get_free_page()**, except that the allocated page is then zero-filled
- To free the pages we have some functions-

1. **void __free_pages(struct page *page, unsigned int order)**
2. **void free_pages(unsigned long addr, unsigned int order)**
3. **void free_page(unsigned long addr)**

- Allocation of page/s may fail so we must define a handler to handle such situations.

kmalloc()

- The kmalloc() function's operation is very similar to that of user-space's familiar malloc() routine, with the exception of the addition of a flags parameter.
- This is used when we want to allocate a small chunk of memory in bytes size.
- For bigger sized memory, the previous page allocation functions is a good option.
- Mostly in Kernel we use Kmalloc() for memory allocation.
- The function is declared in <linux/slab.h>
- `void * kmalloc(size_t size, int flags)`
- The function returns a pointer to a region of memory that is at least size bytes in length.
- The region of memory allocated is physically contiguous.
- On error, it returns NULL.
- Kernel allocations almost always succeed, unless there is an insufficient amount of memory available.
- Still we must check for NULL after all calls to kmalloc() and handle the error appropriately.
- eg. `struct abc *ptr;`

```
ptr = kmalloc(sizeof(struct abc), GFP_KERNEL);
if (!ptr)
    /* handle error ... */
```

- The GFP_KERNEL flag specifies the behavior of the memory allocator while trying to obtain the memory to return to the caller of kmalloc().

gfp_mask Flags

In this section we will discuss about the flags that we used in kmalloc and other low level page functions.

The flags are broken up into three categories:

1. action modifiers
2. zone modifiers
3. types.

- Action modifiers specify how the kernel is supposed to allocate the requested memory.
- In certain situations, only certain methods can be employed to allocate memory.
- Zone modifiers specify from where to allocate memory.
- As we saw in the article on linux addressing (<http://learnlinuxconcepts.blogspot.in/2014/02/linux-addressing.html>) the kernel divides physical memory into multiple zones, each of which serves a different purpose.
- Zone modifiers specify from which of these zones to allocate.
- Type flags specify a combination of action and zone modifiers as needed by a certain type of memory allocation.
- Type flags simplify specifying numerous modifiers; instead, we generally specify just one type flag.
- All the flags are declared in <linux/gfp.h>.
- The file <linux/slab.h> includes this header, however, so we don't often need not include it directly.

Action modifiers-

- For example, interrupt handlers must instruct the kernel not to sleep (because interrupt handlers cannot reschedule) in the course of allocating memory.

| Flag | Description |
|----------------------------|--|
| <code>__GFP_WAIT</code> | The allocator can sleep. |
| <code>__GFP_HIGH</code> | The allocator can access emergency pools. |
| <code>__GFP_IO</code> | The allocator can start disk I/O. |
| <code>__GFP_FS</code> | The allocator can start filesystem I/O. |
| <code>__GFP_COLD</code> | The allocator should use cache cold pages. |
| <code>__GFP_NOWARN</code> | The allocator will not print failure warnings. |
| <code>__GFP_REPEAT</code> | The allocator will repeat the allocation if it fails, but the allocation can potentially fail. |
| <code>__GFP_NOFAIL</code> | The allocator will indefinitely repeat the allocation. The allocation cannot fail. |
| <code>__GFP_NORETRY</code> | The allocator will never retry if the allocation fails. |
| <code>__GFP_NO_GROW</code> | Used internally by the slab layer. |
| <code>__GFP_COMP</code> | Add compound page metadata. Used internally by the hugetlb code. |

- These allocations can be specified together. For example, **`ptr = kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);`**
- Lets see how this allocation will work--
- It will instruct the page allocator (function finally comes to `alloc_pages()` as we had seen before) that the allocation can-
 1. block
 2. perform I/O
 3. perform filesystem operations, if needed.
- This allows the kernel great freedom in how it can find the free memory to satisfy the allocation.

Zone Modifier-

- Zone modifiers specify from which memory zone the allocation should

originate.

- Normally, allocations can be fulfilled from any zone.
- The kernel prefers `ZONE_NORMAL`, however, to ensure that the other zones have free pages when they are needed.
- There are only two zone modifiers because there are only two zones other than `ZONE_NORMAL` (which is where, by default, allocations originate).

| Flag | Description |
|--------------------------|---|
| <code>GFP_DMA</code> | Allocate only from <code>ZONE_DMA</code> |
| <code>GFP_HIGHMEM</code> | Allocate from <code>ZONE_HIGHMEM</code> or <code>ZONE_NORMAL</code> |

- If none of the flags are specified, the kernel fulfills the allocation from either `ZONE_DMA` or `ZONE_NORMAL`, with a strong preference to satisfy the allocation from `ZONE_NORMAL`.
- We cannot specify `GFP_HIGHMEM` to either `__get_free_pages()` or `kmalloc()` because these both return a logical address, and not a page structure.
- Though it is possible that these functions would allocate memory that is not currently mapped in the kernel's virtual address space and, thus, does not have a logical address.
- Only `alloc_pages()` can allocate high memory.
- For majority of our allocations, however, we don't need to specify a zone modifier because `ZONE_NORMAL` is sufficient.

Type Flags-

- The type flags specify the required action and zone modifiers to fulfill a particular type of transaction.
- Therefore, there is a good news that kernel code tends to use the correct type flag and not specify the various number of flags it would want to define.

| | |
|-------------------------|--|
| <code>GFP_ATOMIC</code> | The allocation is high priority and must not sleep. This is |
|-------------------------|--|

| | |
|--------------|--|
| C | the flag to use in interrupt handlers, in bottom halves, while holding a spinlock, and in other situations where we cannot sleep. |
| GFP_NOIO | This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when we cannot cause more disk I/O, which might lead to some unpleasant recursion. |
| GFP_NOFS | This allocation can block and can initiate disk I/O, if it must, but will not initiate a filesystem operation. This is the flag to use in filesystem code when we cannot start another filesystem operation. |
| GFP_KERNEL | This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to in order to obtain the memory requested by the caller. This flag should be our first choice. |
| GFP_USER | This is a normal allocation and might block. This flag is used to allocate memory for user-space processes. |
| GFP_HIGHUSER | This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes. |
| GFP_DMA | This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the above. |

What all action modifier files are internally involved in Type Flags ?

| | |
|--------------|---|
| GFP_ATOMIC | __GFP_HIGH |
| GFP_NOIO | __GFP_WAIT |
| GFP_NOFS | (__GFP_WAIT __GFP_IO) |
| GFP_KERNEL | (__GFP_WAIT __GFP_IO __GFP_FS) |
| GFP_USER | (__GFP_WAIT __GFP_IO __GFP_FS) |
| GFP_HIGHUSER | (__GFP_WAIT __GFP_IO __GFP_FS __GFP_HIGHMEM) |
| GFP_DMA | __GFP_DMA |

Lets try to understand important Type flags.

GFP_KERNEL flag-

- The vast majority of allocations in the kernel use the GFP_KERNEL flag.
- The resulting allocation can sleep as it is normal priority allocation.
- Because the call can block, this flag can be used only from process context that can safely reschedule (that is, no locks are held and so on).
- Because this flag does not make any stipulations as to how the kernel may obtain the requested memory, the memory allocation has a high probability of succeeding.

GFP_ATOMIC flag-

- The GFP_ATOMIC flag is at the extreme end as compared to GFP_KERNEL flag.
- This flag specifies a memory allocation that cannot sleep, the allocation is very restrictive in the memory it can obtain for the caller.
- If no sufficiently sized contiguous chunk of memory is available, the kernel is not very likely to free memory because it cannot put the caller to sleep.
- Conversely, the GFP_KERNEL allocation can put the caller to sleep to swap inactive pages to disk, flush dirty pages to disk, and so on.
- Because GFP_ATOMIC is unable to perform any of these actions, it has less of a chance of succeeding (at least when memory is low) compared to GFP_KERNEL allocations
- Still the GFP_ATOMIC flag is the only option when the current code is unable to sleep, such as with interrupt handlers, softirqs, and tasklets.

GFP_NOIO and GFP_NOFS flags-

- In between these two flags are GFP_NOIO and GFP_NOFS.
- Allocations initiated with these flags might block, but they refrain from performing certain other operations.
- A GFP_NOIO allocation does not initiate any disk I/O whatsoever to fulfill the request
- On the other hand, GFP_NOFS might initiate disk I/O, but does not initiate filesystem I/O.
- One question that immediately comes to our mind. Why might you need these flags?
- They are needed for certain low-level block I/O or filesystem code, respectively

- Imagine if a common path in the filesystem code allocated memory without the GFP_NOFS flag. The allocation could result in more filesystem operations, which would then beget other allocations and, thus, more filesystem operations! This could continue indefinitely.
- Code such as this that invokes the allocator must ensure that the allocator also does not execute it, or else the allocation can create a deadlock.
- Not surprisingly, the kernel uses these two flags only in few places.

GFP_DMA flag-

- The GFP_DMA flag is used to specify that the allocator must satisfy the request from ZONE_DMA.
- This flag is used by device drivers, which need DMA-able memory for their devices. Normally, we combine this flag with the GFP_ATOMIC or GFP_KERNEL flag

Which flag to use when??

| Situation | Solution |
|------------------------------------|---|
| Process context, can sleep | Use GFP_KERNEL |
| Process context, cannot sleep | Use GFP_ATOMIC, or perform your allocations with GFP_KERNEL at an earlier or later point when you can sleep |
| Interrupt handler | Use GFP_ATOMIC |
| Softirq | Use GFP_ATOMIC |
| Tasklet | Use GFP_ATOMIC |
| Need DMA-able memory, can sleep | Use (GFP_DMA GFP_KERNEL) |
| Need DMA-able memory, cannot sleep | Use (GFP_DMA GFP_ATOMIC), or perform your allocation at an earlier point when you can sleep |

kfree()

- kfree undoes the work done by kmalloc().
- This function is declared in `<linux/slab.h>`.
- `void kfree(const void *ptr).`
- use it only for those blocks of memory that was previously allocated using kmalloc().
- eg. `char *buf;`

```
buffer = kmalloc(BUF_SIZE, GFP_ATOMIC);
if (!buffer)
    /* error allocating memory ! */
```

Later, when we no longer need the memory, we must free it.

```
kfree(buffer);
```

vmalloc()

- This Kernel function is similar to user space function malloc().
- Both vmalloc() and malloc() returns virtually contiguous memory but not necessarily physically contiguous.
- In kernel we normally use kmalloc() and seldom use vmalloc().
- vmalloc is used when the requested memory size is quite big as it may not be possible to allocate a large block of contiguous memory via kmalloc() and it may fail.
- The vmalloc() function is declared in `<linux/vmalloc.h>` and defined in `mm/vmalloc.c`.
- Usage is identical to user-space's malloc(): `void * vmalloc(unsigned long size).`
- Usage of vmalloc() also affects the system performance.
- To free an allocation obtained via vmalloc(), we use **void vfree(void *addr).**

What are processes and threads??

- A process is a program (object code stored on some media) in execution.
- Processes are, however, more than just the executing program code (often called the text section in Unix). They also include a set of resources such as open files and pending signals, internal kernel data, processor state, an address space, one or more threads of execution, and a data section containing global variables. Processes, in effect, are the living result of running program code.
- Threads of execution, often shortened to threads, are the objects of activity within the process.
- Each thread includes a unique program counter, process stack, and set of processor registers. The kernel schedules individual threads, not processes.
- In traditional Unix systems, each process consists of one thread. In modern systems, however, multithreaded programs those that consist of more than one thread are common.
- But, to Linux, a thread is just a special kind of process.
- People generally are confused that a program is a process but a process is an active program and related resources. Indeed, two or more processes can exist that are executing the same program. In fact, two or more processes can exist that share various resources, such as open files or an address space.
- A process begins its life when, not surprisingly, it is created. In Linux, this occurs by means of the `fork()` system call, which creates a new process by duplicating an existing one.
- The new process is an exact copy of the old process. (Now, a question might come immediately to our mind- who creates the first process?? The first process is the init process which is literally created from scratch during booting).
- The process that calls `fork()` is the parent, whereas the new process is the child. The parent resumes execution and the child starts execution at the same place, where the call returns. The `fork()` system call returns from the kernel twice: once in the parent process and again in the newborn child.
- Often, immediately after a fork it is desirable to execute a new, different, program. The `exec*()` family of function calls is used to create a new address space and load a new program into it. In modern Linux kernels, `fork()` is actually implemented via the

`clone()` system call,

- Finally, a program exits via the `exit()` system call. This function terminates the process and frees all its resources.
- A parent process can inquire about the status of a terminated child via the `wait4()` system call, which enables a process to wait for the termination of a specific process.
- When a process exits, it is placed into a special zombie state that is used to represent terminated processes until the parent calls `wait()` or `waitpid()`.
- The kernel implements the `wait4()` system call. Linux systems, via the C library, typically provide the `wait()`, `waitpid()`, `wait3()`, and `wait4()` functions. All these functions return status about a terminated process, albeit with slightly different semantics.
- Another name for a process is a task. although when we say task we generally mean a process from the kernel's point of view.

fork()

- The `fork()` function is used to create a new process by duplicating the existing process from which it is called.
- The existing process from which this function is called becomes the parent process and the newly created process becomes the child process.
- The child is a duplicate copy of the parent but there are some exceptions to it.

1. The child has a unique PID like any other process running in the operating system.
2. The child has a parent process ID which is same as the PID of the process that created it.
3. Resource utilization and CPU time counters are reset to zero in child process.
4. Set of pending signals in child is empty.
5. Child does not inherit any timers from its parent
6. The child does not inherit its parent's memory locks (`mlock(2)`, `mlockall(2)`)
7. The child does not inherit outstanding asynchronous I/O operations from its parent.

What is the return value of fork()?

- If the `fork()` function is successful then it returns twice.
- Once it returns in the child process with return value zero and then it

returns in the parent process with child's PID as return value.

- This behavior is because of the fact that once the fork is called, child process is created and since the child process shares the text segment with parent process and continues execution from the next statement in the same text segment so fork returns twice (once in parent and once in child).

fork() in action

Process Descriptor and the Task Structure

- The kernel stores the list of processes in a circular doubly linked list called the task list.
- Each element in the task list is a process descriptor of the type struct task_struct, which is defined in <linux/sched.h>. The process descriptor contains all the information about a specific process.
- Some texts on operating system design call this list the task array. Because the Linux implementation is a linked list and not a static array, it is called the task list.
- The task_struct is a relatively large data structure, at around 1.7 kilobytes on a 32-bit machine.

- This size, however, is quite small considering that the structure contains all the information that the kernel has and needs about a process.
- The process descriptor contains the data that describes the executing program open files, the process's address space, pending signals, the process's state, and much more.

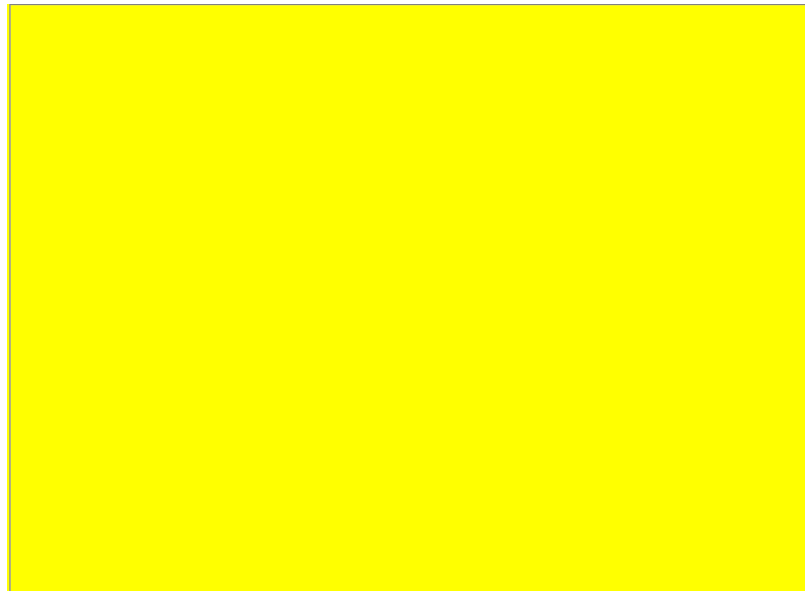
Allocating the Process Descriptor

- The `task_struct` structure is allocated via the slab allocator to provide object reuse and

cache coloring .

- With the process descriptor now dynamically created via the slab allocator, a new

structure, `struct thread_info`, was created that again lives at the bottom of the stack (for stacks that grow down) and at the top of the stack (for stacks that grow up).



The process descriptor and kernel stack.



- The thread_info structure is defined on x86 in <asm/thread_info.h> as

```
struct thread_info {  
    struct task_struct  *task;  
    struct exec_domain  *exec_domain;  
    unsigned long       flags;  
    unsigned long       status;  
    __u32               cpu;  
    __s32               preempt_count;  
};
```

```

        mm_segment_t      addr_limit;
        struct restart_block restart_block;
        unsigned long      previous_esp;
        __u8               supervisor_stack[0];
};

```

- Each task's `thread_info` structure is allocated at the end of its stack.

The `task` element of the structure is a pointer to the task's actual `task_struct`.

Storing the Process Descriptor

- The system identifies processes by a unique process identification value or PID.

The PID is a numerical value that is represented by the opaque type (An opaque type is a

data type whose physical representation is unknown or irrelevant.) `pid_t`, which is typically an `int`.

- Because of backward compatibility with earlier Unix and Linux versions, however,

the default maximum value is only 32,768 (that of a short `int`), although the value can optionally be increased to the full range afforded the type.

- The kernel stores this value as `pid` inside each process descriptor.
- This maximum value is important because it is essentially the maximum number

of processes that may exist concurrently on the system.

Although 32,768 might be sufficient for a desktop system, large servers may require many more processes.

- Inside the kernel, tasks are typically referenced directly by a pointer to their

`task_struct` structure.

It is very useful to be able to quickly look up the process descriptor of the currently executing task, which is done via the `current` macro.

This macro must be separately implemented by each architecture. Some architectures save a pointer to the `task_struct` structure of the currently running process in a

register

, allowing for efficient access.

Other architectures, such as x86 (which has few registers to waste), make use of the fact

that `struct thread_info` is stored on the kernel stack to calculate the location of `thread_info` and subsequently the `task_struct`.

- On x86, `current` is calculated by masking out the 13 least significant bits of the stack pointer to obtain the `thread_info` structure.

- This is done by the `current_thread_info()` function.

- The assembly is shown here:

```
movl $-8192, %eax
andl %esp, %eax
```

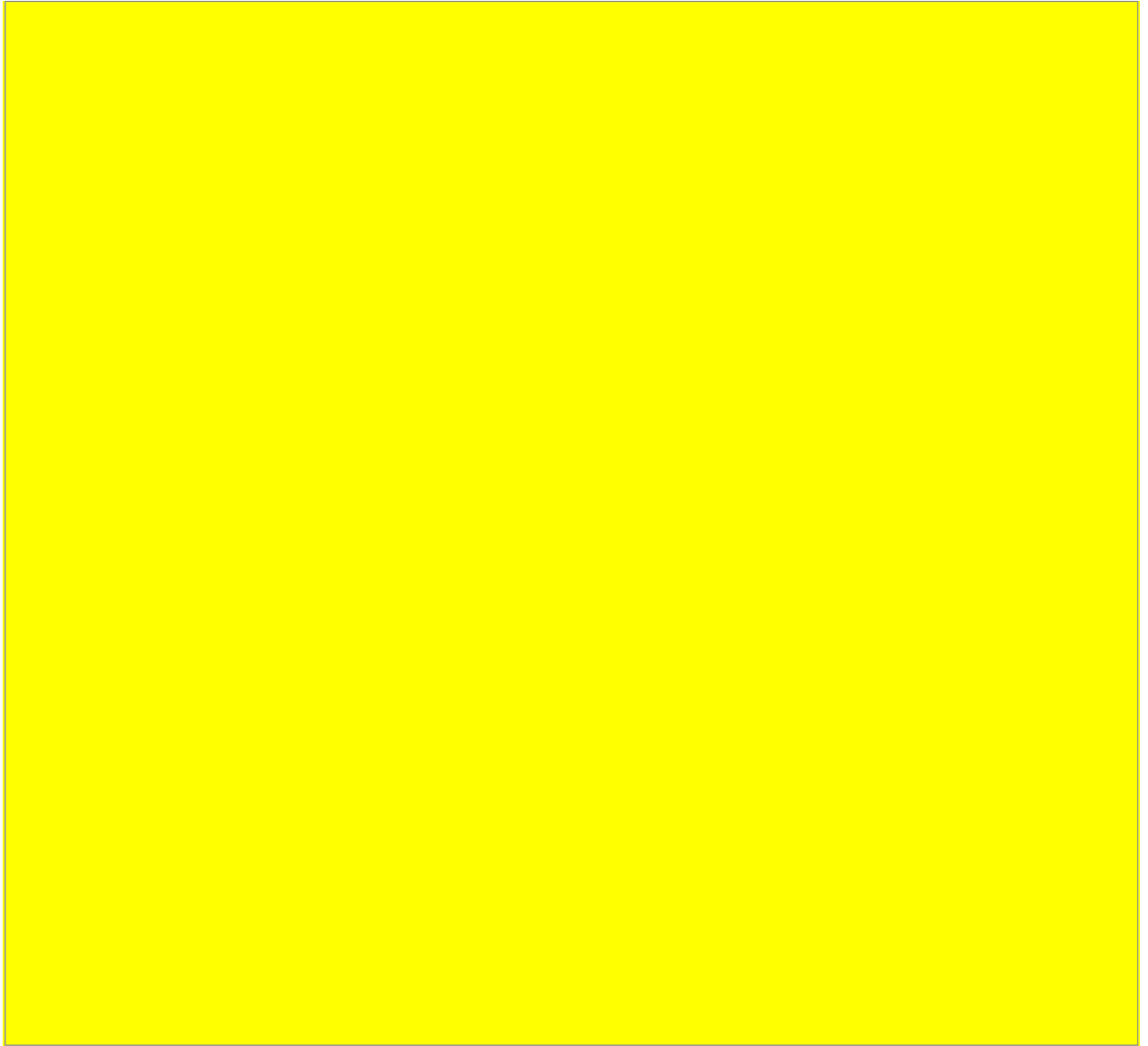
- This assumes that the stack size is 8KB. When 4KB stacks are enabled, 4096 is used in

lieu of 8192.

- Finally, `current` dereferences the `task` member of `thread_info` to return the `task_struct`:

```
current_thread_info()->task;
```

Process State



Flow chart of process states.



- The state field of the process descriptor describes the current condition of the process.
- Each process on the system is in exactly one of five different states. This value is

represented by one of five flags:

1.TASK_INTERRUPTIBLE -The process is sleeping (that is, it is blocked), waiting for some condition to exist. When this condition exists, the kernel sets the process's state to TASK_RUNNING.

The process also awakes prematurely and becomes runnable if it receives a signal.

2.TASK_UNINTERRUPTIBLE- This state is identical to TASK_INTERRUPTIBLE except that it does not wake up and become runnable if it receives a signal. This is used in situations where the

process must wait without interruption or when the event is expected to occur quite quickly.

Because the task does not respond to signals in this state, TASK_UNINTERRUPTIBLE is less often used than TASK_INTERRUPTIBLE

3.TASK_ZOMBIE -The task has terminated, but its parent has not yet issued a wait4() system call.

The task's process descriptor must remain in case the parent wants to access it. If the parent calls wait4(), the process descriptor is deallocated.

4.TASK_STOPPED- Process execution has stopped; the task is not running nor is it eligible to run.

This occurs if the task receives the SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal or if it receives

any signal while it is being debugged.

5.TASK_RUNNING -The process is runnable; it is either currently running or on a runqueue waiting to run .This is the only possible state for a process executing in user-space; it can also apply to a process in kernel-space that is actively running.

Manipulating the Current Process State

- Kernel code often needs to change a process's state. The preferred mechanism is using

```
set_task_state(task, state);    /* set task 'task' to state 'state' */
```

- This function sets the given task to the given state. If applicable, it also provides a

memory barrier to force ordering on other processors (this is only needed on SMP systems). Otherwise, it is equivalent to

```
task->state = state;
```

- The method `set_current_state(state)` is synonymous to `set_task_state(current, state)`.

Process Context

- One of the most important parts of a process is the executing program code.
- This code is read in from an executable file and executed within the program's address

space.

- Normal program execution occurs in user-space. When a program executes a system call

or triggers an exception, it enters kernel-space.

INTERRUPTS

- The kernel is responsible for servicing the request of hardware.
- The CPU must process the request from the hardware.
- Since the CPU frequency and the hardware frequency is not the same (hardware is slower) so the hardware can't send the data/request to the CPU synchronously.
- There are two ways in which CPU can check about the request from a hardware-

1. Polling

2. Interrupt

- In polling the CPU keeps on checking all the hardware for the availability of any request.
- In interrupt the CPU takes care of the hardware only when the hardware requests for some service.
- Polling is an expensive job as it requires a greater overhead.
- The better way is to use interrupt as the hardware will request the CPU only when it has some request to be serviced.
- Different devices are given different interrupt values called IRQ (interrupt request) lines.
- For ex. IRQ zero is the timer interrupt and IRQ one is the keyboard interrupt.
- An interrupt is physically produced by electronic signals originating from hardware devices and directed into input pins on an interrupt controller.
- Some interrupt numbers are static and some interrupts are dynamically assigned.
- Be it static or dynamic, the kernel must know which interrupt number is associated with which hardware.
- The interrupt controller, in turn, sends a signal to the processor. The processor detects this signal and interrupts its current execution to handle the interrupt.
- The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.
- Interrupt handlers in Linux need not be reentrant. When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors, preventing another interrupt on the same

line from being received. Normally all other interrupts are enabled, so other interrupts are serviced, but the current line is always disabled.

Comparison between interrupts and exceptions-

- Exceptions occur synchronously with respect to the processor clock while interrupts occur async.
- That is why exceptions are often called synchronous interrupts.
- Exceptions are produced by the processor while executing instructions either in response to a programming error (for example, divide by zero) or abnormal conditions that must be handled by the kernel (for example, a page fault).
- Many processor architectures handle exceptions in a similar manner to interrupts, therefore, the kernel infrastructure for handling the two is similar.
- Exceptions are of two types- traps and software interrupts.
- Exceptions are produced by the processor while executing instructions either in response to a programming error (for example, divide by zero) or abnormal conditions that must be handled by the kernel (for example, a page fault).
- A trap is a kind of exceptions, whose main purpose is for debugging (eg. notify the debugger that an instruction has been reached) or it occurs during abnormal conditions.
- A software interrupt occur at the request of a programmer eg. System calls.

Interrupt Handler

- These are the C functions that get executed when an interrupt comes.
- Each interrupt is associated with a particular interrupt handler.
- Interrupt handler is also known as interrupt service routine (ISR).
- Since interrupts can come any time therefore interrupt handlers has to be short and quick.

- At least the interrupt handler has to acknowledge the hardware and rest of the work can be done at a later time.

Top Halves Versus Bottom Halves

- There are two goals that an interrupt handler needs to perform 1. execute quickly and 2. perform a large amount of work .
- Because of these conflicting goals, the processing of interrupts is split into two parts, or halves.
- The interrupt handler is the top half.
- It is run immediately upon receipt of the interrupt and performs only the work that is time critical, such as acknowledging receipt of the interrupt or resetting the hardware.
- Work that can be performed later is delayed until the bottom half.
- The bottom half runs in the future, at a more convenient time, with all interrupts enabled.
- Let us consider a case where we need to collect the data from a data card and then process it.
- The most important job is to collect the data from data card to the memory and free the card for incoming data and this is done in top half.
- The rest part which deals with the processing of data is done in the bottom half.

Registering an Interrupt Handler

- Interrupt handlers are the responsibility of the driver managing the hardware. Each device has one associated driver and, if that device uses interrupts (and most do), then that driver registers one interrupt handler.
- Drivers can register an interrupt handler and enable a given interrupt line for handling via the function

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
```

```
unsigned long irqflags,  
const char *devname,  
void *dev_id)
```

- The first parameter, `irq`, specifies the interrupt number to allocate. For some devices, for example legacy PC devices such as the system timer or keyboard, this value is typically hard-coded. For most other devices, it is probed or otherwise determined programmatically and dynamically.
- The second parameter, `handler`, is a function pointer to the actual interrupt handler that services this interrupt. This function is invoked whenever the operating system receives the interrupt. Note the specific prototype of the handler function: It takes three parameters and has a return value of `irqreturn_t`.
- The third parameter, `irqflags`, might be either zero or a bit mask of one or more of the following flags:
 - **SA_INTERRUPT** This flag specifies that the given interrupt handler is a fast interrupt handler. Fast interrupt handlers run with all interrupts disabled on the local processor. This enables a fast handler to complete quickly, without possible interruption from other interrupts. By default (without this flag), all interrupts are enabled except the interrupt lines of any running handlers, which are masked out on all processors. Sans the timer interrupt, most interrupts do not want to enable this flag.
 - **SA_SAMPLE_RANDOM** This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy. Do not set this if your device issues interrupts at a predictable rate (for example, the system timer) or can be influenced by external attackers (for example, a networking device). On the other hand, most other hardware generates interrupts at nondeterministic times and is, therefore, a good source of entropy.
 - **SA_SHIRQ** This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line. More information on shared handlers is provided in a following section.
- The fourth parameter, `devname`, is an ASCII text representation of the device associated with the interrupt. For example, this value for the keyboard interrupt on a PC is "keyboard". These text names are used by `/proc/irq` and `/proc/interrupts` for communication with the user, which is discussed shortly.

- The fifth parameter, `dev_id`, is used primarily for shared interrupt lines. When an interrupt handler is freed (discussed later), `dev_id` provides a unique cookie to allow the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass `NULL` here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared (and unless your device is old and crusty and lives on the ISA bus, there is good chance it must support sharing). This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure: This pointer is unique and might be useful to have within the handlers and the Device Model.
- On success, `request_irq()` returns zero. A nonzero value indicates error, in which case the specified interrupt handler was not registered. A common error is `-EBUSY`, which denotes that the given interrupt line is already in use (and either the current user or you did not specify `SA_SHIRQ`).
- Note that `request_irq()` can sleep and therefore cannot be called from interrupt context or other situations where code cannot block.
- On registration, an entry corresponding to the interrupt is created in `/proc/irq`.
- The function `proc_mkdir()` is used to create new `procfs` entries. This function calls `proc_create()` to set up the new `procfs` entries, which in turn call `kmalloc()` to allocate memory
- In a driver, requesting an interrupt line and installing a handler is done via `request_irq()`:

```
if (request_irq(irqn, my_interrupt, SA_SHIRQ, "my_device", dev)) {
    printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
    return -EIO;
}
```

In this example, `irqn` is the requested interrupt line, `my_interrupt` is the handler, the line can be shared, the device is named "my_device," and we passed `dev` for `dev_id`. On failure, the code prints an error and returns. If the call returns zero, the handler has been successfully installed. From that point forward, the handler is invoked in response to an interrupt. It is important to initialize hardware and register an interrupt handler in the proper order to prevent the interrupt handler from running before the device is fully initialized.

Freeing an Interrupt Handler

- When we unregister our device drivers it is compulsory to free the interrupt handler what we have registered for the device.
- This frees the interrupt line.
- `void free_irq(unsigned int irq, void *dev_id)`
- If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via `dev_id` is removed, but the interrupt line itself is disabled only when the last handler is removed.
- A call to `free_irq()` must be made from process context.

How to write an interrupt handler?

- The declaration of interrupt handler:

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs)
```

- The first parameter, `irq`, is the numeric value of the interrupt line the handler is servicing. This is not entirely useful today, except perhaps in printing log messages. Before the 2.0 kernel, there was not a `dev_id` parameter and thus `irq` was used to differentiate between multiple devices using the same driver and therefore the same interrupt handler. As an example of this, consider a computer with multiple hard drive controllers of the same type.
- The second parameter, `dev_id`, is a generic pointer to the same `dev_id` that was given to `request_irq()` when the interrupt handler was registered. If this value is unique (which is recommended to support sharing), it can act as a cookie to differentiate between multiple devices potentially using the same interrupt handler. `dev_id` might also point to a structure of use to the interrupt handler. Because the device structure is both unique to each device and potentially useful to have within the

handler, it is typically passed for `dev_id`.

- The final parameter, `regs`, holds a pointer to a structure containing the processor registers and state before servicing the interrupt.
- The return value of an interrupt handler is the special type `irqreturn_t`. An interrupt handler can return two special values, `IRQ_NONE` or `IRQ_HANDLED`. The former is returned when the interrupt handler detects an interrupt for which its device was not the originator. The latter is returned if the interrupt handler was correctly invoked, and its device did indeed cause the interrupt. Alternatively, `IRQ_RETVAL(val)` may be used. If `val` is non-zero, this macro returns `IRQ_HANDLED`. Otherwise, the macro returns `IRQ_NONE`. These special values are used to let the kernel know whether devices are issuing spurious (that is, unrequested) interrupts. If all the interrupt handlers on a given interrupt line return `IRQ_NONE`, then the kernel can detect the problem. Note the curious return type, `irqreturn_t`, which is simply an `int`.
- The interrupt handler is normally marked `static` because it is never called directly from another file.
- The role of the interrupt handler depends entirely on the device and its reasons for issuing the interrupt. At a minimum, most interrupt handlers need to provide acknowledgment to the device that they received the interrupt. Devices that are more complex need to additionally send and receive data and perform extended work in the interrupt handler. As mentioned, the extended work is pushed as much as possible into the bottom half handler, which I have discussed in other post on [Bottom Halves](#).

Shared Handlers

A shared handler is registered and executed much like a non-shared handler. There are three main differences:

- The `SA_SHIRQ` flag must be set in the `flags` argument to `request_irq()`.
- The `dev_id` argument must be unique to each registered handler. A pointer to any per-device structure is sufficient; a common choice is the device structure as it is both unique and potentially useful to the handler. You cannot pass `NULL` for a shared handler!
- The interrupt handler must be capable of distinguishing whether its device actually generated an interrupt. This requires both hardware support and associated logic in the interrupt handler. If the hardware did not offer this capability, there would be no way for the interrupt

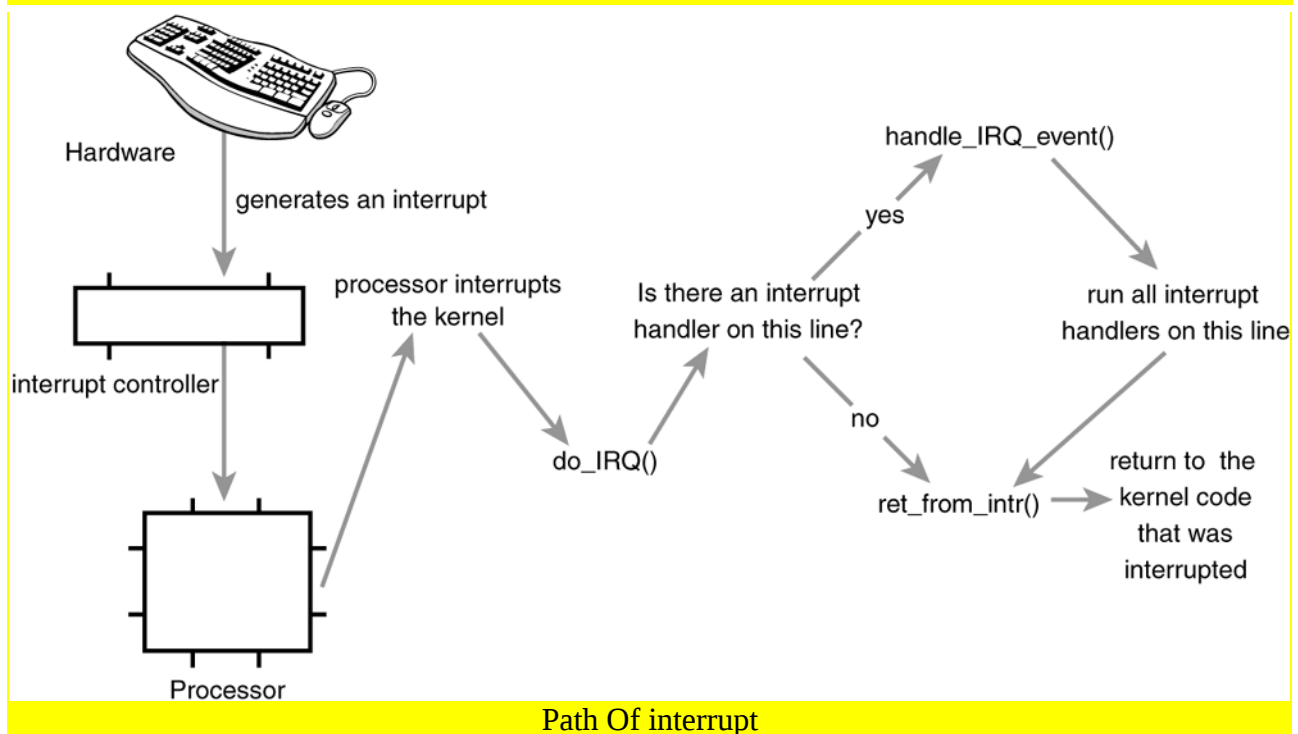
handler to know whether its associated device or some other device sharing the line caused the interrupt.

Interrupt Context

- When executing an interrupt handler or bottom half, the kernel is in interrupt context. (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.)
- In process context, the current macro points to the associated task. Furthermore, because a process is coupled to the kernel in process context, process context can sleep or otherwise invoke the scheduler.
- Interrupt context is not associated with a process. The current macro is not relevant (although it points to the interrupted process). Without a backing process, interrupt context cannot sleep how would it ever reschedule? Therefore, you cannot call certain functions from interrupt context. If a function sleeps, you cannot use it from your interrupt handler this limits the functions that one can call from an interrupt handler.
- Interrupt context is time critical because the interrupt handler interrupts other code. Code should be quick and simple. Busy looping is discouraged. This is a very important point; always keep in mind that your interrupt handler has interrupted other code (possibly even another interrupt handler on a different line!).
- Because of this asynchronous nature, it is imperative that all interrupt handlers be as quick and as simple as possible. As much as possible, work should be pushed out from the interrupt handler and performed in a bottom half, which runs at a more convenient time.
- The setup of an interrupt handler's stacks is a configuration option. Historically, interrupt handlers did not receive their own stacks. Instead, they would share the stack of the process that they interrupted.
- The kernel stack is two pages in size; typically, that is 8KB on 32-bit architectures and 16KB on 64-bit architectures. Because in this setup interrupt handlers share the stack, they must be exceptionally economical with what data they allocate there. Of course, the kernel stack is limited to begin with, so all kernel code should be cautious.

Implementation of Interrupt Handling

The implementation of interrupt handler is architecture dependent and hardware dependent.



- A device issues an interrupt by sending an electric signal over its bus to the interrupt controller.
- If the interrupt line is enabled (they can be masked out), the interrupt controller sends the interrupt to the processor.
- In most architectures, this is accomplished by an electrical signal that is sent over a special pin to the processor. Unless interrupts are disabled in the processor (which can also happen), the processor immediately stops what it is doing, disables the interrupt system, and jumps to a predefined location in memory and executes the code located there. This predefined point is set up by the kernel and is the entry point for interrupt handlers.
- The interrupt's journey in the kernel begins at this predefined entry point, just as system calls enter the kernel through a predefined exception handler.
- For each interrupt line, the processor jumps to a unique location in memory and executes the code located there.
- In this manner, the kernel knows the IRQ number of the incoming

interrupt. The initial entry point simply saves this value and stores the current register values (which belong to the interrupted task) on the stack; then the kernel calls `do_IRQ()`. From here onward, most of the interrupt handling code is written in C however, it is still architecture dependent.

- The `do_IRQ()` function is declared as

```
unsigned int do_IRQ(struct pt_regs regs)
```

- Because the C calling convention places function arguments at the top of the stack, the `pt_regs` structure contains the initial register values that were previously saved in the assembly entry routine. Because the interrupt value was also saved, `do_IRQ()` can extract it.
- The x86 code is `int irq = regs.orig_eax & 0xff;`
- After the interrupt line is calculated, `do_IRQ()` acknowledges the receipt of the interrupt and disables interrupt delivery on the line. On normal PC machines, these operations are handled by `mask_and_ack_8259A()`, which `do_IRQ()` calls.
- Next, `do_IRQ()` ensures that a valid handler is registered on the line, and that it is enabled and not currently executing. If so, it calls `handle_IRQ_event()` to run the installed interrupt handlers for the line. On x86, `handle_IRQ_event()` is

```
asmlinkage int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                                struct irqaction *action)
{
    int status = 1;
    int retval = 0;

    if (!(action->flags & SA_INTERRUPT))
        local_irq_enable();

    do {
        status |= action->flags;
        retval |= action->handler(irq, action->dev_id, regs);
        action = action->next;
    } while (action);

    if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);

    local_irq_disable();

    return retval;
}
```

- First, because the processor disabled interrupts, they are turned back on unless `SA_INTERRUPT` was specified during the handler's registration. Recall that `SA_INTERRUPT` specifies that the handler must be run with interrupts disabled. Next, each potential handler is executed in a loop.
- If this line is not shared, the loop terminates after the first iteration. Otherwise, all handlers are executed. After that, `add_interrupt_randomness()` is called if `SA_SAMPLE_RANDOM` was specified during registration.
- This function uses the timing of the interrupt to generate entropy for the random number generator.
- Finally, interrupts are again disabled (`do_IRQ()` expects them still to be off) and the function returns. Back in `do_IRQ()`, the function cleans up and returns to the initial entry point, which then jumps to `ret_from_intr()`.
- The routine `ret_from_intr()` is, as with the initial entry code, written in assembly. This routine checks whether a reschedule is pending (this implies that `need_resched` is set).
- If a reschedule is pending, and the kernel is returning to user-space (that is, the interrupt interrupted a user process), `schedule()` is called. If the kernel is returning to kernel-space (that is, the interrupt interrupted the kernel itself), `schedule()` is called only if the `preempt_count` is zero (otherwise it is not safe to preempt the kernel). After `schedule()` returns, or if there is no work pending, the initial registers are restored and the kernel resumes whatever was interrupted.
- On x86, the initial assembly routines are located in `arch/i386/kernel/entry.S` and the C methods are located in `arch/i386/kernel/irq.c`. Other supported architectures are similar.

/proc/interrupts

- Procfs is a virtual filesystem that exists only in kernel memory and is typically mounted at `/proc`.
- Reading or writing files in procfs invokes kernel functions that simulate reading or writing from a real file.
- A relevant example is the `/proc/interrupts` file, which is populated with statistics related to interrupts on the system. Here is sample output from a uniprocessor PC:

CPU0

```
0: 3602371 XT-PIC timer
1: 3048 XT-PIC i8042
2: 0 XT-PIC cascade
4: 2689466 XT-PIC uhci-hcd, eth0
5: 0 XT-PIC EMU10K1
12: 85077 XT-PIC uhci-hcd
15: 24571 XT-PIC aic7xxx
NMI: 0
LOC: 3602236
ERR: 0
```

- The first column is the interrupt line. On this system, interrupts numbered 02, 4, 5, 12, and 15 are present
- . Handlers are not installed on lines not displayed.
- The second column is a counter of the number of interrupts received. A column is present for each processor on the system, but this machine has only one processor.
- As you can see, the timer interrupt has received 3,602,371 interrupt, whereas the sound card (EMU10K1) has received none (which is an indication that it has not been used since the machine booted).
- The third column is the interrupt controller handling this interrupt. XT-PIC corresponds to the standard PC programmable interrupt controller. On systems with an I/O APIC, most interrupts would list IO-APIC-level or IO-APIC-edge as their interrupt controller.
- Finally, the last column is the device associated with this interrupt. This name is supplied by the devname parameter to request_irq(), as discussed previously. If the interrupt is shared, as is the case with interrupt number four in this example, all the devices registered on the interrupt line are listed.
- procfs code is located primarily in fs/proc. The function that provides /proc/interrupts is, not surprisingly, architecture dependent and named show_interrupts().

What is Virtual Memory?

In computing, **virtual memory** is a **memory** management technique that is implemented using both hardware and software. It maps **memory** addresses used by a program, called **virtual** addresses, into physical addresses in computer **memory**.

LINUX KERNEL INTERNALS

Concurrency and Race Conditions

Introduction

- The problem of concurrency occurs when the system tries to do more than one thing at once.
- With the use of SMP(symmetric multiprocessing, which means more than one processor or cores access the same memory) the chances of concurrent access has increased.
- Race condition means uncontrolled access to the shared data.
- This race condition leads to memory leak because the shared data is overwritten and the first wrote data is lost.
- It is advised to avoid sharing of resource while writing a program, for eg. avoid using global variables.
- It is also not possible to run away from sharing of resource as we have limited hardware and software resources.
- So how do we deal with concurrent access?

Locking/Mutual Exclusion provides access management.

Semaphores and Mutexes

- A process in linux goes to sleep("blocks") when it has nothing to do.
- Semaphores are used when the thread holding the lock can sleep.
- Semaphore framwork consists of a single integer value plus a pair of functions (P and V).
- Lets understand a semaphore operation-

A process which wants to access a critical region calls the function P, if the semaphore value is greater than 0 then we decrement the semaphore value by one and the process continues; if the semaphore ale is less than or equal to 0, then the process waits for some other process to release the lock.

A process which wants to get out of the critical region calls the function V, increments the semaphore value by one and wakes up the processes that are sleeping.

- When we want our semaphore to be accessed by only one process at a time then such semaphore is called a Mutex(mutual exclusion).

Linux Semaphore Implementation

- To use semaphore in our file we must include the header file <asm/semaphore.h> header file.
- We create semaphore directly and then set it to some value. **//void sema_init(struct semaphore *sem, int val);**
- For mutex also we can do similar things-
- **DECLARE_MUTEX(name);** //initialises the mutex with 1
- **DECLARE_MUTEX_LOCKED(name);** //initialises the mutex with 0

If the mutex has to be allocated dynamically then we use either of these two functions-

```
void init_MUTEX(struct semaphore *sem);  
void init_MUTEX_LOCKED(struct semaphore *sem);
```

In linux terminology the P function is called down and V function is called up function.

Semaphore Methods

sema_init(struct semaphore *, int) //Initializes the dynamically created semaphore to the given count

init_MUTEX(struct semaphore *) //Initializes the dynamically created semaphore with a count of one

init_MUTEX_LOCKED(struct semaphore *) //Initializes the dynamically created semaphore with a count of zero (so it is initially locked)

down_interruptible (struct semaphore *) //Tries to acquire the given semaphore and enter interruptible sleep if it is contended

down(struct semaphore *) //Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended

down_trylock(struct semaphore *) //Tries to acquire the given semaphore and immediately return nonzero if it is contended

up(struct semaphore *) //Releases the given semaphore and wakes a waiting task, if any

- For different critical sections we use different semaphore names.

ReaderWriter Semaphores

- rwsems is a special type of semaphore, we need to include <linux/rwsem.h>
- It is used rarely but is useful.
- rwsem allows either one writer or an unlimited number of readers to hold the semaphore
- rwsem must be initialised with **void init_rwsem(struct rw_semaphore *sem);**
- For read only access

```
void down_read(struct rw_semaphore *sem);  
int down_read_trylock(struct rw_semaphore *sem);  
void up_read(struct rw_semaphore *sem);
```

- Similarly for writers we have

```
void down_write(struct rw_semaphore *sem);  
int down_write_trylock(struct rw_semaphore *sem);  
void up_write(struct rw_semaphore *sem);  
void downgrade_write(struct rw_semaphore *sem); // used when we want writers to  
finish write fast and read to take major timeslice
```

- By default writers are given more priority over readers.
- That's why we use rwsemaphore when write access is used rarely and that too for a short period of time.

Completions

- This feature was added in kernel version 2.4.7
- This is a lightweight mechanism with only one task, it allows one thread to tell other one that the task is done.
- The headerfile to be used for this is <linux/completion.h>

- A completion can be created with:

```
DECLARE_COMPLETION(my_completion);
```

- Or, if we want the completion to be created and initialized dynamically:

```
struct completion my_completion;  
/* ... */  
init_completion(&my_completion);
```

- We wait for the completion by calling:

```
void wait_for_completion(struct completion *c);
```

- This function performs an uninterruptible wait.
- If our code calls wait_for_completion and nobody ever completes the task, the result will be an unkillable process.

- The actual completion event may be signalled by calling one of the following:

```
void complete(struct completion *c);  
void complete_all(struct completion *c);
```

- The two functions behave differently if more than one thread is waiting for the same completion event. complete wakes up only one of the waiting threads while complete_all allows all of them to proceed.

- If we use complete_all, we must reinitialize the completion structure before reusing it.
- The macro: INIT_COMPLETION(struct completion c); can be used to quickly perform this reinitialization.
- Lets see an example of completion variable being used.

```
DECLARE_COMPLETION(comp);
```

```

ssize_t complete_read (struct file *filp, char __user *buf, size_t count, loff_t
*pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",
current->pid, current->comm);
    wait_for_completion(&comp);
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* EOF */
}
ssize_t complete_write (struct file *filp, const char __user *buf, size_t count,
loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
current->pid, current->comm);
    complete(&comp);
    return count; /* succeed, to avoid retrial */
}

```

- We can see that the read operation will wait if the write operation is being performed(till it finishes).

Spinlocks

- Unlike semaphores, spinlocks can be used in the code that can't sleep eg. interrupt handlers
- Spinlock offers better performance than semaphore but there are some constraints also.
- A spinlock is a mutual exclusion device that has only 2 states; locked/unlocked.
- In coding perspective this lock sets/unsets a bit.
- If the lock is unavailable then the code goes into a tight loop where it repeatedly checks the lock until it is available.
- Care must be taken for setting and unsetting the bit as many threads are in loop that may set.
- Spinlocks are designed to be used on a multiprocessor system.
- Single processor system running a preemptive code also behaves as an SMP system from the point of view of concurrency.
- If a nonpreemptive code running on a processor takes a lock then it would run till

eternity as no other thread would ever be able to obtain the CPU to release the lock. For this reason, spinlock operation on a uniprocessor system is designed to perform NOOP(nooperation) and at someplaces whe change the masking statuses of some IRQs.

- To use the spinlock we must include the header file s <linux/spinlock.h>
- This initialization can be done at compile time by using `spinlock_t my_lock = SPIN_LOCK_UNLOCKED;`
- Or the initialization can be done at runtime with `void spin_lock_init(spinlock_t *lock);`
- Before entering a critical section our code must obtain the requisite lock with:

void spin_lock(spinlock_t *lock);

- Since all spinlock waits are, by their nature, uninterruptible so, once we call `spin_lock`, we will spin until the lock becomes available to our code .
- To release a lock that we have obtained we call `void spin_unlock(spinlock_t *lock);`
- Lets us consider a practical scenario

We have taken a spinlock in our driver code and while holding the lock our driver experiences a function which will change the context of code for eg. `copy_to_user()` or put the process to sleep or say any interrupt comes that throws our driver code out of the processor. Our driver code is still holding the lock and won't free it for some other task.that wants it.

- This situation is undesirable so we acquire spinlock where the code is atomic and it can't sleep.
- For example we dont use spinlock for malloc as it can sleep.
- The kernel code takes care of spin lock and preemption, whenever spinlock is called the kernel blocks the preemption(interrupts) on that processor
- Even in case of uniprocessor system preemption is disabled to avoid the race condition.
- We must take care that our process must holdthe spinlock for a small period of time, beacuse longer is the hold longer is the wait for other process which whishes to acquire the lock,(the process might be a high priority one).
- There are four functions that can lock a spinlock

void spin_lock(spinlock_t *lock);//Normal one

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags); //disables interrupts (on the local processor only) before taking the spinlock and the previous interrupt state is stored in flags

void spin_lock_irq(spinlock_t *lock);//used when we are sure nothing else might have disabled the interrupts already, it also disables interrupt

void spin_lock_bh(spinlock_t *lock);//disables the software interrupt before taking the lock

- We must use the proper lock in proper situation- eg. hardware interrupt and software interrupt or normal case.
- Just as we saw the 4 ways to acquire the spinlock we have 4 ways to release also

```
void spin_unlock(spinlock_t *lock);
```

```
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
```

```
void spin_unlock_irq(spinlock_t *lock);
```

```
void spin_unlock_bh(spinlock_t *lock);
```

- There is also a set of nonblocking spinlock operations:
- **int spin_trylock(spinlock_t *lock);**
- **int spin_trylock_bh(spinlock_t *lock);**
- These functions return nonzero on success (the lock was obtained), 0 otherwise.
- There is no “try” version that disables interrupts.

Reader/Writer Spinlocks

- This is analogous to what we had seen in case of semaphores, a single writer and any number of readers.
- **rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */**
- **rwlock_t my_rwlock;**
- **rwlock_init(&my_rwlock); /* Dynamic way */**
- For readers, the following functions are available:

```
void read_lock(rwlock_t *lock);
```

```
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
```

```
void read_lock_irq(rwlock_t *lock);
```

```
void read_lock_bh(rwlock_t *lock);
```

```
void read_unlock(rwlock_t *lock);
```

```
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
```

```
void read_unlock_irq(rwlock_t *lock);
```

```
void read_unlock_bh(rwlock_t *lock);
```

```
void write_lock(rwlock_t *lock);
```

```
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

Alternatives to Locking ?

There are situations in kernel programming where atomic access can be set up without the need for full locking.

Atomic Operation- These operations are performed in a single machine cycle. An `atomic_t` holds an `int` value on all supported architectures.

Bit operations-

The `atomic_t` type is good for performing integer arithmetic. It doesn't work as well, however, when you need to manipulate individual bits in an atomic manner. For that purpose, instead, the kernel offers a set of functions that modify or test single bits atomically. Because the whole operation happens in a single step, no interrupt (or other processor) can interfere.

Seqlocks-

Used when the resource to be protected is small, simple and frequently accessed. Here readers are allowed the free access to the resource but they check for their collision with the writers. And if a collision is detected then they retry to read.

7) what is NFS

The Network File System (NFS) is a way of mounting Linux discs/directories over a network. An NFS server can export one or more directories that can then be mounted on a remote Linux machine. Note, that if you need to mount a Linux filesystem on a Windows machine, you need to use Samba/CIFS instead.

Makefile and Kconfig

make file

Define the files to be built and link to other files, Makefiles within the kernel are kbuild Makefiles that use the kbuild infrastructure, these make evaluates to either y (for built-in) or m (for module) If it is neither y nor m, then the file will not be compiled nor linked.

<https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt>

Kconfig

Every entry has its own dependencies. These dependencies are used to determine the visibility of an entry. Any child entry is only visible if its parent entry is also visible.

Menu entries

Most entries define a config option; all other entries help to organize them. A single configuration option is defined like this:

```
config MODVERSIONS
bool "Set version information on all module symbols"
depends on MODULES
help
```

Usually, modules have to be recompiled whenever you switch to a new kernel.

Every line starts with a key word and can be followed by multiple arguments. "config" starts a new config entry. The following lines define attributes for this config option. Attributes can be the type of the config option, input prompt, dependencies, help text and default values. A config option can be defined multiple times with the same name, but every definition can have only a single input prompt and the type must not conflict.

What is pre-emptive and non-preemptive scheduling?

Answer

Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemptive scheduling.

Eg: Round robin

In non-preemptive scheduling, a running task is executed till completion. It cannot be interrupted.

Eg First In First Out

Operating system - pre-emptive and non-preemptive scheduling - Jan 07, 2010 at 15:00 PM by Vidya Sagar

What is pre-emptive and non-preemptive scheduling?

Preemptive scheduling: The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

Non-Preemptive scheduling: When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.

What is a semaphore?

Answer

A semaphore is a variable. There are 2 types of semaphores:

Binary semaphores

Counting semaphores

Binary semaphores have 2 methods associated with it. (up, down / lock, unlock)

Binary semaphores can take only 2 values (0/1). They are used to acquire locks. When a resource is available, the process in charge set the semaphore to 1 else 0.

Counting Semaphore may have value to be greater than one, typically used to allocate resources from a pool of identical resources.

What is difference between binary semaphore and mutex?

The differences between binary semaphore and mutex are:

- Mutex is used exclusively for mutual exclusion. Both mutual exclusion and synchronization can be used by binary.
- A task that took mutex can only give mutex.
- From an ISR a mutex can not be given.
- Recursive taking of mutual exclusion semaphores is possible. This means that a task that holds before finally releasing a semaphore, can take the semaphore more than once.
- Options for making the task which takes as DELETE_SAFE are provided by Mutex, which means the task deletion is not possible when holding the mutex.

Threads differ from process and thread:

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerable state information, whereas multiple threads within a process share state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication mechanisms.
- Context switching between threads in the same process is typically faster than context switching between processes.

6 Stages of Linux Boot Process (Startup Sequence)

Press the power button on your system, and after few moments you see the Linux login prompt.

Have you ever wondered what happens behind the scenes from the time you press the power button until the Linux login prompt appears?

The following are the 6 high level stages of a typical Linux boot process.

| | |
|----------|---|
| BIOS | Basic Input/Output System executes MBR |
| MBR | Master Boot Record executes GRUB |
| GRUB | Grand Unified Bootloader executes Kernel |
| Kernel | Kernel executes /sbin/init |
| Init | Init executes runlevel programs |
| Runlevel | Runlevel programs are executed from /etc/rc.d/rc*.d/ |

1. BIOS

- BIOS stands for Basic Input/Output System
- Performs some system integrity checks
- Searches, loads, and executes the boot loader program.
- It looks for boot loader in floppy, cd-rom, or hard drive. You can press a key (typically F12 or F2, but it depends on your system) during the BIOS startup to change the boot sequence.
- Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
- So, in simple terms BIOS loads and executes the MBR boot loader.

2. MBR

- MBR stands for Master Boot Record.
- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
- MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
- It contains information about GRUB (or LILO in old systems).
- So, in simple terms MBR loads and executes the GRUB boot loader.

3. GRUB

- GRUB stands for Grand Unified Bootloader.
- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen, waits for few seconds, if you don't enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn't understand filesystem).
- Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.

```
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-194.el5PAE)
    root (hd0,0)
    kernel /boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/
    initrd /boot/initrd-2.6.18-194.el5PAE.img
```

- As you notice from the above info, it contains kernel and initrd image.
- So, in simple terms GRUB just loads and executes Kernel and initrd images.

4. Kernel

- Mounts the root file system as specified in the "root=" in grub.conf
- Kernel executes the /sbin/init program
- Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a 'ps -ef | grep init' and check the pid.
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

5. Init

- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels

- 0 – halt
 - 1 – Single user mode
 - 2 – Multiuser, without NFS
 - 3 – Full multiuser mode
 - 4 – unused
 - 5 – X11
 - 6 – reboot
- Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.
 - Execute 'grep initdefault /etc/inittab' on your system to identify the default run level
 - If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
 - Typically you would set the default run level to either 3 or 5.

6. Runlevel programs

- When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail OK”. Those are the runlevel programs, executed from the run level directory as defined by your run level.
- Depending on your default init level setting, the system will execute the programs from one of the following directories.
 - Run level 0 – /etc/rc.d/rc0.d/
 - Run level 1 – /etc/rc.d/rc1.d/
 - Run level 2 – /etc/rc.d/rc2.d/
 - Run level 3 – /etc/rc.d/rc3.d/
 - Run level 4 – /etc/rc.d/rc4.d/
 - Run level 5 – /etc/rc.d/rc5.d/
 - Run level 6 – /etc/rc.d/rc6.d/
- Please note that there are also symbolic links available for these directory under /etc directly. So, /etc/rc0.d is linked to /etc/rc.d/rc0.d.
- Under the /etc/rc.d/rc*.d/ directories, you would see programs that start with S and K.
- Programs starts with S are used during startup. S for startup.
- Programs starts with K are used during shutdown. K for kill.
- There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
- For example, S12syslog is to start the syslog daemon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.

There you have it. That is what happens during the Linux boot process.

Deadlocks

A deadlock is a condition involving one or more threads of execution and one or more resources,

such that each thread is waiting for one of the resources, but all the resources are already held. The threads are all waiting for each other, but they will never make any progress toward releasing the resources that they already hold. Therefore, none of the threads can continue, which means we have a deadlock.

A good analogy is a four-way traffic stop. If each car at the stop decides to wait for the other cars before going, no car will ever go and we have a traffic deadlock.

The simplest example of a deadlock is the self-deadlock[4]: If a thread of execution attempts to acquire a lock it already holds, it has to wait for the lock to be released.

What is interrupt?

In systems programming, an interrupt is a signal to the processor.

It can be emitted either by hardware or software indicating an event that needs immediate attention.

Interrupts are a commonly used technique in real-time computing and such a system is said to be **interrupt-driven**.

| Hardware | Software (Exception/Trap) | Software (Instruction set) |
|--|--|---|
| Interrupt Request (IRQ) sent from device to processor. | Exception/Trap sent from processor to processor, caused by an exceptional condition in the processor itself. | A special instruction in the instruction set which causes an interrupt when it is executed. |

- **Hardware interrupt**

A hardware interrupt is a signal which can tell the CPU that something happen in hardware device, and should be immediately responded. Hardware interrupts are triggered by peripheral devices outside the microcontroller. An interrupt causes the processor to save its state of execution and begin execution of an interrupt service routine.

Unlike the software interrupts, hardware interrupts are **asynchronous** and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an **interrupt request (IRQ)**.

- **Software interrupt**

Software interrupt is an instruction which cause a context switch to an interrupt handler similar to a hardware interrupt. Usually it is an interrupt generated within a processor by executing a special instruction in the instruction set which causes an interrupt when it is executed.

Another type of software interrupt is triggered by an exceptional condition in the processor itself. This type of interrupt is often called a **trap** or **exception**.

Unlike the hardware interrupts where the number of interrupts is limited by the number of interrupt request (IRQ) lines to the processor, software interrupt can have hundreds of different interrupts.

What is interrupt latency?

Interrupt latency refers primarily to the software interrupt handling latencies. In other words, the amount of time that elapses from the time that an external **interrupt arrives** at the processor until the time that the **interrupt processing begins**. One of the most important aspects of kernel real-

time performance is the ability to service an interrupt request (IRQ) within a specified amount of time.

IRQ vs FIQ

- **IRQ (Interrupt Request)**

An (or IRQ) is a hardware signal sent to the processor that temporarily stops a running program and allows a special program, an interrupt handler, to run instead. Interrupts are used to handle such events as data receipt from a modem or network, or a key press or mouse movement.

- **FIQ (Fast Interrupt Request)**

An FIQ is just a higher priority interrupt request, that is prioritized by disabling IRQ and other FIQ handlers during request servicing. Therefore, no other interrupts can occur during the processing of the active FIQ interrupt.

What is IOCTL?

An `ioctl`, which means "input-output control" is a kind of device-specific system call. There are only a few system calls in Linux (300-400), which are not enough to express all the unique functions devices may have. So a driver can define an `ioctl` which allows a userspace application to send it orders. However, `ioctls` are not very flexible and tend to get a bit cluttered (dozens of "magic numbers" which just work... or not), and can also be insecure, as you pass a buffer into the kernel - bad handling can break things easily.

`ioctl` function is useful when one is implementing a device driver to set the configuration on the device. e.g. a printer has configuration options to check and set font, font size etc. `ioctl` could be used to get current font as well as set font to another one. In user application make use of `ioctl` to send a code to a printer telling it to return the current font or to set font to a new one.

```
int ioctl(int fd, int request, ...)
```

1. `fd` is file descriptor, the one returned by `open`
2. `request` is request code. e.g `GETFONT` will get current font from printer, `SETFONT` will set font on a printer.
3. third argument is void *. Depending on second argument third may or may not be present. e.g. if second argument is `SETFONT`, third argument may give font name as `ARIAL`.

How system call works

wrapper routine pushes trap number to user stack and calls special instruction called `trap`.

1. `syscall` copy args from user stack to kernel stack.
2. saves state or process and store it in kernel stack by accessing `u-area`.
3. execute in kernel stack
4. copy return values and errors.
5. restore process state and set the mode accordingly

Deadlocks with Two or More Threads

Deadlocks can occur when two (or more) threads are each blocked, waiting for a condition to occur that only the other one can cause. For instance, if thread A is blocked on a condition variable waiting for thread B to signal it, and thread B is blocked on a condition variable waiting for thread A to signal it, a deadlock has occurred because neither thread will ever signal the other. You should take care to avoid the possibility of such situations because they are quite difficult to detect.

One common error that can cause a deadlock involves a problem in which more than one thread is trying to lock the same set of objects. For example, consider a program in which two different threads, running two different thread functions, need to lock the same two mutexes. Suppose that thread A locks mutex 1 and then mutex 2, and thread B happens to lock mutex 2 before mutex 1. In a sufficiently unfortunate scheduling scenario, Linux may schedule thread A long enough to lock mutex 1, and then schedule thread B, which promptly locks mutex 2. Now neither thread can progress because each is blocked on a mutex that the other thread holds locked.

This is an example of a more general deadlock problem, which can involve not only synchronization objects such as mutexes, but also other resources, such as locks on files or devices. The problem occurs when multiple threads try to lock the same set of resources in different orders. The solution is to make sure that all threads that lock more than one resource lock them in the same order.

Interrupts & Exceptions

An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Such events corresponds to electrical signals generated by hardware circuits which are inside and outside the CPU.

Interrupts are often divided into synchronous and asynchronous interrupts.

Synchronous interrupts(Exceptions) are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction. Exceptions on the other hand are caused by programming errors or by anomalous conditions that must be handled by the kernel. In the first case, the kernel handles the exception by delivering to the current process one of the signals familiar to Unix programmer. In the second case the kernel performs all the steps needed to recover from anomalous condition, such as a page fault or a request(via an int instruction) for a kernel service.

Asynchronous interrupts(Interrupts) are generated by other hardware devices at arbitrary times with respect to the CPU clock signals. Interrupts are issued by either interval timer or I/O devices. For instance, the arrival of a keystroke from a user sets off an interrupt.