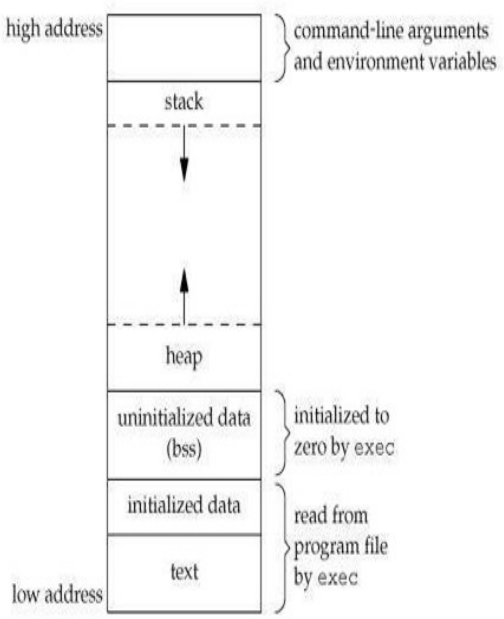
1st Round

- Storage Classes in C

- Memory Layout

This memory layout is organized in following fashion :-



1>Text or Code Segment :-

Text segment contains machine code of the compiled program. Usually, the text segment is **sharable** so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. The text segment of an executable object file is often **read-only segment** that prevents a program from being accidentally modified.

2>Initialized Data Segment :-

Initialized data stores all **global, static, constant**, and external variables **(** declared with **extern** keyword **)** that are initialized beforehand. Data segment is **not read-only**, since the values of the variables can be altered at run time.

This segment can be further classified into initialized **read-only area** and initialized **read-write area**.

```
#include <stdio.h>

char c[]="rishabh tripathi";     /*  global variable stored in Initialized Data Segment in read-write area*/
const char s[]="HackerEarth";    /* global variable stored in Initialized Data Segment in read-only area*/
```

```
int main()
{
   static int i=11;          /* static variable stored in Initialized Data Segment*/
   return 0;
}
```

3>Uninitialized Data Segment (bss) :-

Data in this segment is initialized to arithmetic **0** before the program starts executing. Uninitialized data starts at the end of the data segment and contains all **global** variables and **static** variables that are initialized to **0** or do not have explicit initialization in source code.

```
#include <stdio.h>

char c;              /* Uninitialized variable stored in bss*/

int main()
{
   static int i;     /* Uninitialized static variable stored in bss */
   return 0;
}
```

4>Heap :-

Heap is the segment where ***dynamic memory allocation*** usually takes place. When some more memory need to be allocated using **malloc** and **calloc** function, heap grows upward. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

```
#include <stdio.h>
int main()
{
   char *p=(char*)malloc(sizeof(char));    /* memory allocating in heap segment */
   return 0;
}
```

5>Stack :-

Stack segment is used to store all **local variables** and is used for **passing arguments** to the functions along with the return address of the instruction which is to be executed after the function call is over. Local variables have a scope to the block which they are defined in, they are created when control enters into the block. All recursive function calls are added to stack.

The stack and heap are traditionally located at opposite ends of the process's virtual address space.

STATICALLY LINKED

- The term ‘statically linked’ means that the program and the particular library that it’s linked against are combined together by the linker at link time.
  This means that the binding between the program and the particular library is fixed and known at link time before the program run. It also means that we can’t change this binding,

unless we re-link the program with a new version of the library.
Programs that are linked statically are linked against archives of objects (libraries) that typically have the extension of **.a**. An example of such a collection of objects is the standard C library, **libc.a**.

- You might consider linking a program statically for example, in cases where you weren't sure whether the correct version of a library will be available at runtime, or if you were testing a new version of a library that you don't yet want to install as shared.
- For gcc, the –static option can be used during the compilation/linking of the program.

gcc –static filename.c –o filename

- The drawback of this technique is that the executable is quite big in size, all the needed information need to be brought together.

DYNAMICALLY LINKED

- The term 'dynamically linked' means that the program and the particular library it references are not combined together by the linker at link time.
- Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references.
- This means that the binding between the program and the shared object is done at runtime that is before the program starts, the appropriate shared objects are found and bound.
- This type of program is called a partially bound executable, because it isn't fully resolved. The linker, at link time, didn't cause all the referenced symbols in the program to be associated with specific code from the library.
- Instead, the linker simply said something like: "This program calls some functions within a particular shared object, so I'll just make a note of which shared object these functions are in, and continue on".
- Symbols for the shared objects are only verified for their validity to ensure that they do exist somewhere and are not yet combined into the program.
- The linker stores in the executable program, the locations of the external libraries where it found the missing symbols. Effectively, this defers the binding until runtime.
- Programs that are linked dynamically are linked against shared objects that have the extension **.so**. An example of such an object is the shared object version of the standard C library, **libc.so**.
- The advantageous to defer some of the objects/modules during the static linking step until they are finally needed (during the run time) includes:

- Bitwise Operations
- Diff between structure and unions
- Function Pointer

- String Reverse function
- Services of Memory Management
- Memory Alignment
- Linked List

2nd Round

- Synchronization mechanism and differences

what is spin lock?

If a Resource is locked , A thread that wants to access that resource may

Repetively check whether the “Resource” is avaliable . During that time the threadmay loop and check the resource without doing any usfull work such a LOCK is

termed as “SPIN LOCK”.

detailed concept of semaphores and difference between binary semaphore and

mutex?

- MUTual EXclusion and synchronization can be done by binary semaphore.

- Mutex is used only for MUTual Exclusion.

- Semaphore value can be changed by other THREAD also.

- A mutex can be relesed by the same thread which acquired it .

- Adv of semphore is that they can be used to synchronize two unrelated processes

trying to access the same resource

- From an ISR a Mutex can NOT be used , beacuse of another proccess goes to

sleep..

“Semaphore are the Synchronization tool to overcome critcal section problem”.

“Mutex is also a tool that is used to provide deadlock free mutual exclusion”.

Semaphore: Use a semaphore when you (thread) want to sleep till some other

thread tells you to wake up. Semaphore 'down' happens in one thread (producer) and

semaphore 'up' (for same semaphore) happens in another thread (consumer)

e.g.: In producer-consumer problem, producer wants to sleep till at least one buffer

slot is empty - only the consumer thread can tell when a buffer slot is empty.

Mutex: Use a mutex when you (thread) want to execute code that should not

be executed by any other thread at the same time. Mutex 'down' happens in one

thread and mutex 'up' must happen in the same thread later on.

e.g.: If you are deleting a node from a global linked list, you do not want another thread to muck around with pointers while you are deleting the node. When you acquire a mutex and are busy deleting a node, if another thread tries to acquire the same mutex, it will be put to sleep till you release the mutex.

Spinlock: Use a spinlock when you really want to use a mutex but your thread is not allowed to sleep.

e.g.: An interrupt handler within OS kernel must never sleep. If it does the system will freeze / crash. If you need to insert a node to globally shared linked list from the interrupt handler, acquire a spinlock - insert node - release spinlock.

- IPC mechanisms
- Process State Diagram
- ARM Modes
- **Interrupt Handling in Linux and explain each argument**

INTERRUPT:-

It is an event in a CPU that causes the CPU to stop exicuting the current progrmm code begin the exicution of Special pice of code called as Interrupt handler.

Interrupt latency:-

iterrupt latency is the “time requied for an ISR responds to an interrupt.”

Following steps could be followed to reduce the latency

- ISRs being simple and short.
- Interrupts being serviced immediately
- Avoiding those instructions that increase the latency period.
- Also by prioritizing interrupts over threads.
- Avoiding use of inappropriate APIs.

Installing an Interrupt Handler

*int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *),*

*unsigned long flags, const char *dev_name, void *dev_id);*

*void free_irq(unsigned int irq, void *dev_id);*

The value returned from request_irq to the requesting function is either 0 to indicate success or a negative error code, as usual. It’s not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

unsigned int irq

The interrupt number being requested.

irqreturn_t (*handler)(int, void *, struct pt_regs *)

The pointer to the handling function being installed. We discuss the arguments to this function and its return value later in this chapter.

unsigned long flags

As you might expect, a bit mask of options (described later) related to interrupt management.

const char *dev_name

The string passed to request_irq is used in /proc/interrupts to show the owner of the interrupt (see the next section).

void *dev_id

Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). If the interrupt is not shared, dev_id can be set to NULL , but it a good idea anyway to use this item to point to the device structure. We’ll see a practical use for dev_id in the sec- tion “Implementing a Handler.” The bits that can be set in flags are as follows:

SA_INTERRUPT

When set, this indicates a “fast” interrupt handler. Fast handlers are executed with interrupts disabled on the current processor (the topic is covered in the sec- tion “Fast and Slow Handlers”).

SA_SHIRQ

This bit signals that the interrupt can be shared between devices. The concept of sharing is outlined in the section “Interrupt Sharing.”

SA_SAMPLE_RANDOM

This bit indicates that the generated interrupts can contribute to the entropy pool used by /dev/random and /dev/urandom. These devices return truly random numbers when read and are designed to help application software choose secure keys for encryption.

```
if (short_irq >= 0) {
result = request_irq(short_irq, short_interrupt, SA_INTERRUPT, "short", NULL);
```

```
if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n", short_irq);
        short_irq = -1;
        }
else { /* actually enable it -- assume this *is* a parallel port */
        outb(0x10,short_base+2);
        }
}
```

- How to Write Handler

- Role in Projects Worked, what implementation done

- Gathering requirements
- planning the functional flow according to gathered information
- Preparing feature documentation
- porting the USB driver, Touch driver
- porting device driver on the new chipset