.

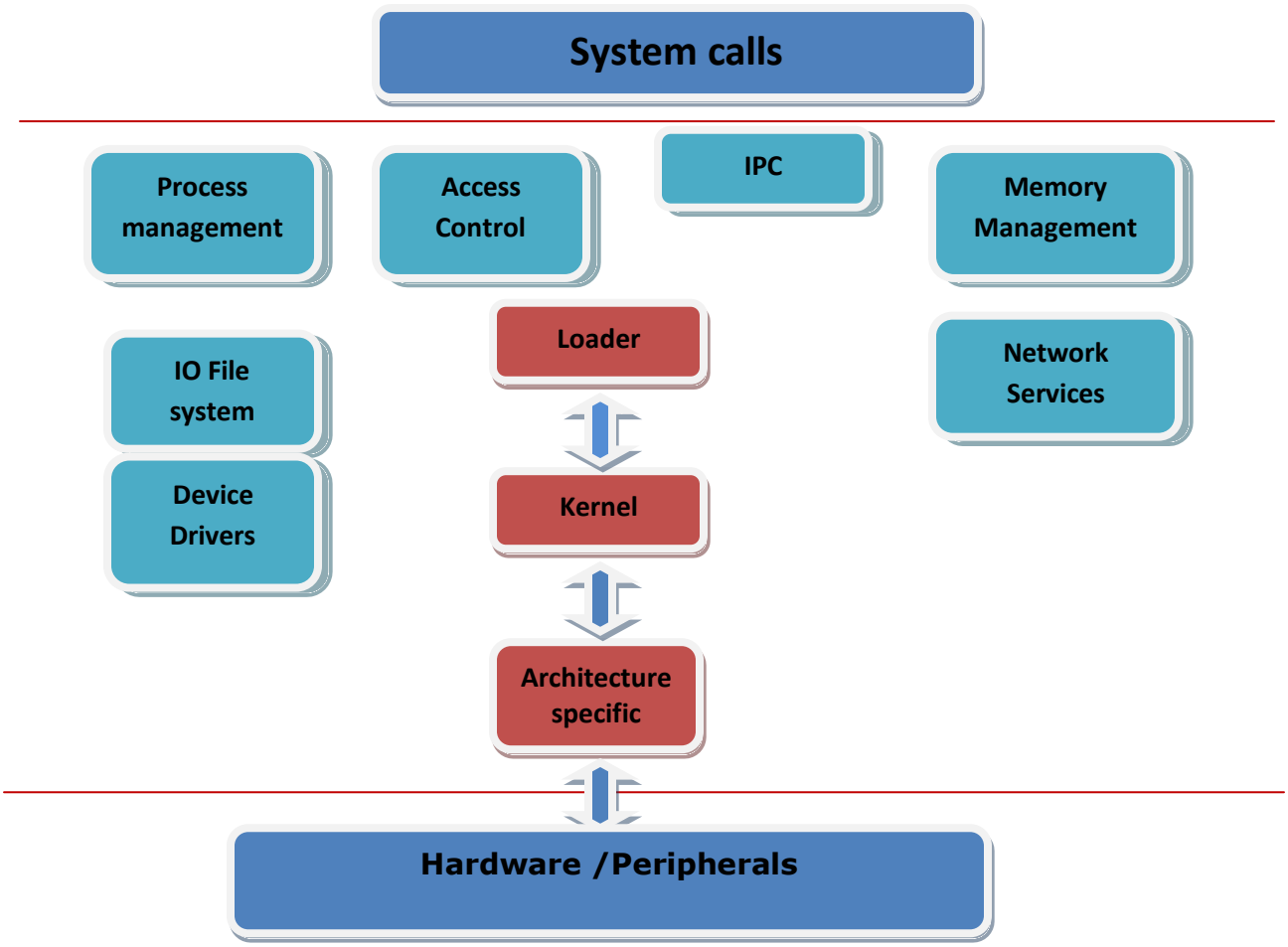
OS Designs –Class Notes

Operating System Designs

Operating system serves as a platform to run all kinds of applications. Basically OS is integration of two modules.

a) User Interface
b) Kernel

Generic Kernel



Kernel is a collection of various subsystems that provides services to an application.

- Loader, Kernel and Architecture specific together called as **Core Subsystem.** They are responsible for system initialization and resource management.

- Remaining all are called **service subsystems**.

Architecture specific (HAL / BSP): This layer is responsible for platform initialization which includes processor and I/O controllers.

In-Kernel: This layer is responsible for data initialization and initializing other kernel subsystems.

Loader: This layer is responsible for loading an application by allocating required memory.

Service Subsystems:

These subsystems are responsible for providing resources for the applications at the runtime.

TYPES OF OPERATING SYSTEMS

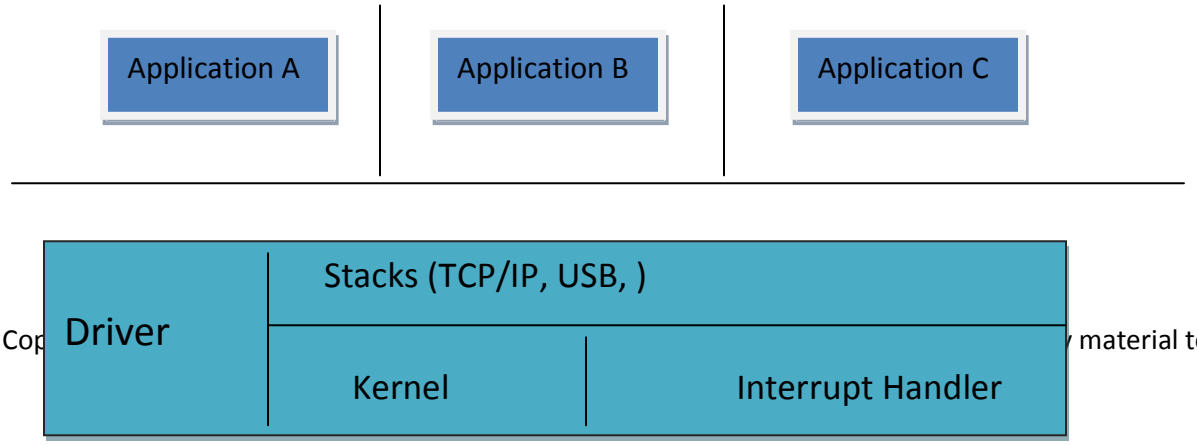
1) **General purpose OS:**
   This type of OS is designed for desktop computers. It provides lots of services and supports any applications. Here the Kernel will be very heavy.
2) **Embedded OS:**
   This type of OS is designed for closed devices usually targeted for RISC (Reduced Instruction Set computing) architecture, and services are fine tuned for specific set of applications. The kernel will be light.
3) **Real Time OS:**
   This OS has fixed time response.

Operating System Design:

In earlier days, user applications and Kernel were aligned together. There by any bug in that application crashed the entire OS.

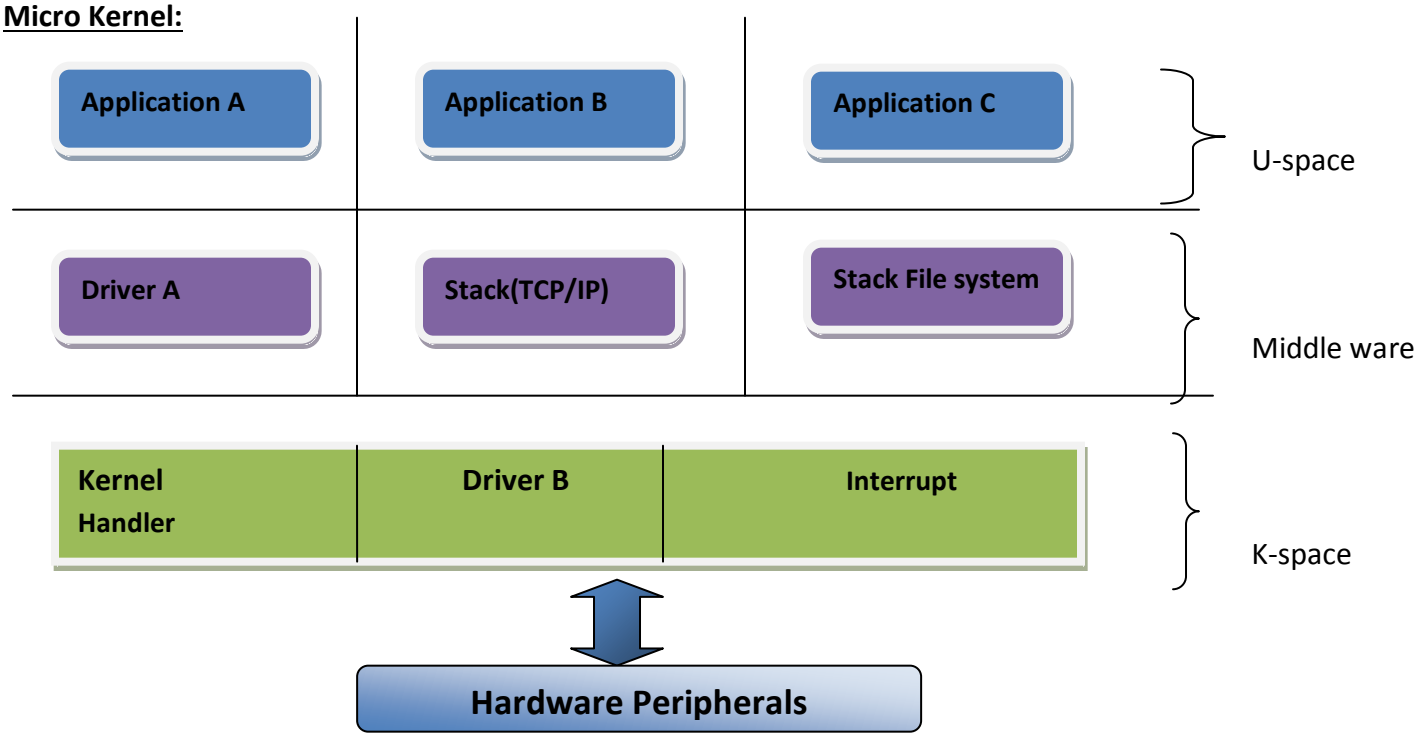
To ensure that exceptions caused by an application do not result in kernel crash, memory (RAM) is logically partitioned into kernel space and User space. Applications are loaded in user space and Kernel is loaded in kernel space.

Monolithic Kernel:



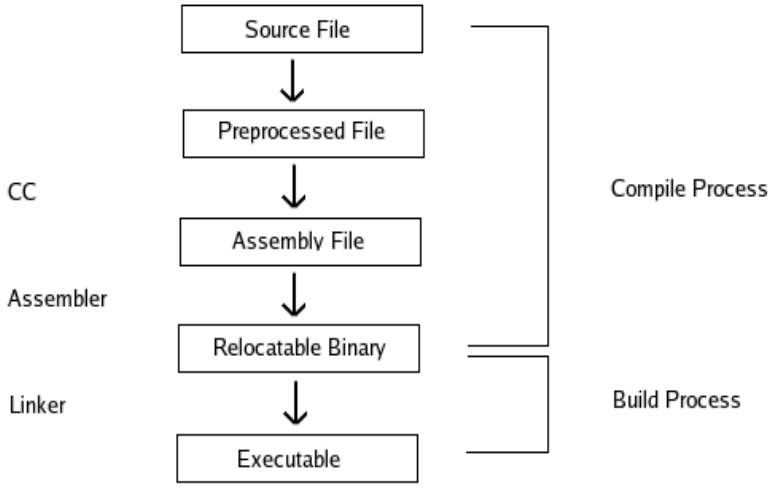
- Most of the Desktop OS are monolithic. The applications are completely abstracted from kernel.
- The main drawback of Monolithic kernel is the buggy drivers installed from un-trusted sources can crash the kernel.

Micro Kernel:



- Here the third party services are put on the middle layer.
- Example: Android.

Compilation Stages of a C program – Class Notes



- The above diagram shows the four compilation stages through which a source code passes to become an executable.

In order to understand the function of each stage we will consider a sample C program **test.c**

$ **vim test.c**

```
#include<stdio.h>
int  main()
{
    printf("Welcome to Veda\n");
    return 0;
}
```

Now let's run gcc compiler to get the executable

$ **gcc test.c -o test**

$ **./test**

Welcome to Veda

- Now we have a basic idea about how gcc is used to convert a source code into binary. We'll review the 4 stages of a C program.

1) **Preprocessor:**

This is the very first stage through which a source code passes. To understand preprocessing better, you can compile the above '**test.c'** program using flag **-E**, which will generate the preprocessed output. Preprocessor converts our .c file into .i file.

$**gcc -E test.c -o test.i**

If we go through the **test.i** file we will notice that **stdio.h** header file will be replaced with its full code. Thus the preprocessor adds all the header files described in the source code.

In the above command if we add a flag -**v** i.e.,

$**gcc -v -E test.c -o test.i**

This will open the specs, in which the order of the execution of each tool and the command line arguments are specified. We can also see that gcc is using cc1 to compile.

2) **Compiler :**

The job of the compiler is to take the preprocessor output as an input, here it will take **test.i** and produce an assembler output. The output of the compiler is architecture dependent. The output file for this stage is '**test.s'**. The output present in **test.s** is assembly level instructions.

$gcc -S test.i -o test.s

If we go through the test.s file, we can find the machine level instructions which the assembler understands.

3) **Assembler:**

At this stage the **test.s** file is taken as an input and an intermediate file test.o is produced. This file is also known as the relocatable file. This file is produced by the assembler that understands

and converts a **'.s**' file with assembly instructions into a **'.o**' object file which contains machine level instructions.

$gcc -c test.s -o test.o

- Here if we give -**v** flag we can find that gcc is using '**as**' i.e., it is invoking assembler.
- Since the output of this stage is a machine level file (test.o). So we cannot view the content of it through editor. This can be viewed through a special tool called **objdump**

$objdump –D test.o | more

4) Linker:

This is the final stage at which all the linking of function calls with their definitions is done. That means it links all the libraries and object files into single executable file. The linker also does some extra work; it adds some extra code to our program that is required when the program starts and when the program ends. We call this code as **runtime code**. Thus it generates executable image from the relocatable file.

$gcc test.o -o test

DIFFERENCE BETWEEN RELOCATABLES AND EXECUTABLES

RELOCATABLES	EXECUTABLES
1. Instructions of relocatable binary are bound to offset address assigned as per the position of the instruction within the procedure Eg: 00000000<main> 0: 55 push %ebp	1. Executable binary contains instructions bound to platform specific load address. Eg: 080483e4<main> 80483e4: 55 push %ebp
2. Function call instructions in relocatable object files referred to called functions offset position Eg: call 12 <main+0x12>	2. Call instructions in executable binaries referred to functions base address Eg: call 8048300<printf@plt>
3. It contains compiled instructions that appear in the source file.	3. It contains functionality plus run time code.

Functionalities of Linker

- Instruction relocation
- Procedure relocation (assigning address to all the functions)
- Append runtime code into executable image.

Significances of Runtime code :

- Runtime code is responsible for allocating stack segment and configuring it on the process address space.

  When a program is loaded from the binary file the memory (address space) allocated for it comprises of code and data segments and a stack segment which is needed for the execution of the program.

- Linux run time code comprises of three key functions
  - Init: This function is responsible for allocation of stack and appending to address space and configuration of stack.
  - Start: This function is invoked after initialization of stack segment is carried out, and is responsible for invoking main function.
  - Fini : When main function returns, start invokes fini, it is responsible for releasing stack segment.
- Run time code invokes kernel system calls and is OS specific. So presence of runtime makes an application binary OS specific.

VEDA Solutions
inspiring linux learning

Creating Libraries – Class Notes

Executables can be categorized into two:

- **Static Executables**: They contain fully resolved library functions that are physically linked into executable image during built.

- **Dynamic Executable**: They contain symbolic references to library functions used and these references are fully resolved either during application load time or run time.

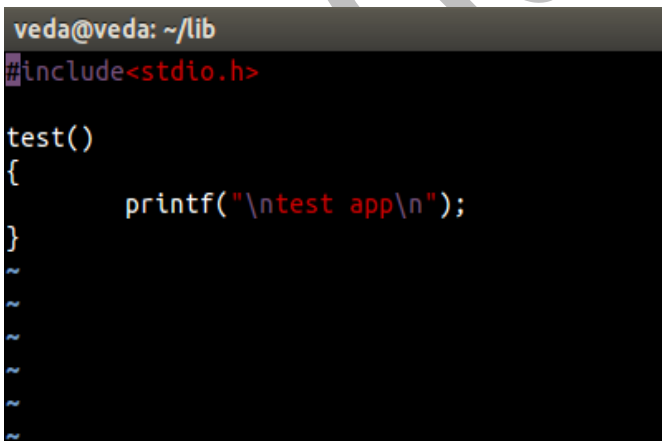
There are mainly two types of libraries:

- Static Libraries **(.a)**
- Dynamically linked libraries **(.so)**

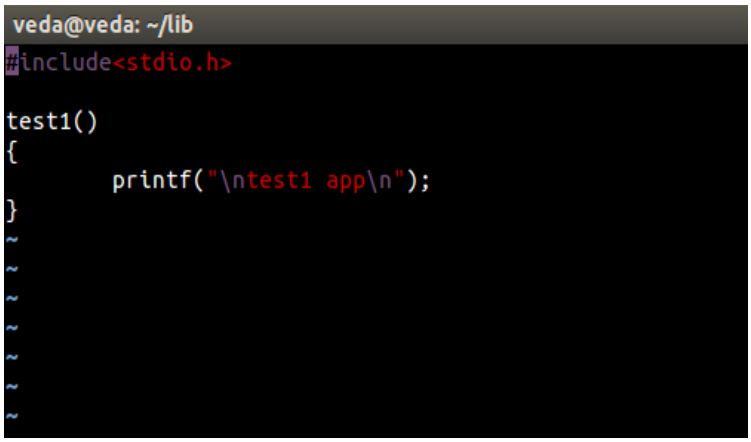
Steps for creating Static Libraries:

1) Implement library sources
   eg.:

one.c



two.c



2) Compile sources into relocatables

$**gcc -c one.c -o one.o**

$**gcc -c two.c -o two.o**

3) Using Unix static library tool archive [**ar**] create static library

$**ar rcs libtest.a one.o two.o**

- To check the number of files in libtest.a and libtest.so, we uses the commands

  $**ar -t libtest.a**

  - In our case it will show one.o and two.o.

  $**nm -s libtest.a**

  - It will list all object files also all the functions inside the object file.

Thus created a static library **libtest.a** and in order to check this library let's write **a C** program **test.c** which calls the two functions **test()** and **test1()** in **one.c** and **two.c.**

$**vim test.c**

```
#include<stdio.h>
int main()
{
test();
test1();
}
```

Then we need to compile **test.c** with the static library **libtest.a** . Generally linker checks the default path only, but our library is in our working directory therefore while compiling we have to specify the path of our library in order to avoid undefined reference error.

$**gcc test.c -o teststat ./libtest.a**

This will create an executable **teststat** which is statically linked

Steps for creating Dynamic libraries:

Generally relocatables are of two types

- **Position dependent:** Relocatables which cannot be shared.
- **Position independent:** Relocatables which can be shared.

➢ First create two **.c** files **one.c** and **two.c** as above.

➢ Compile the sources into position independent relocatables

$gcc -c -fpic one.c -o one.o

$gcc -c -fpic two.c -o two.o

-fpic is a flag which is used to tell the linker as it is position independent.

Using dynamic linker creating shared library

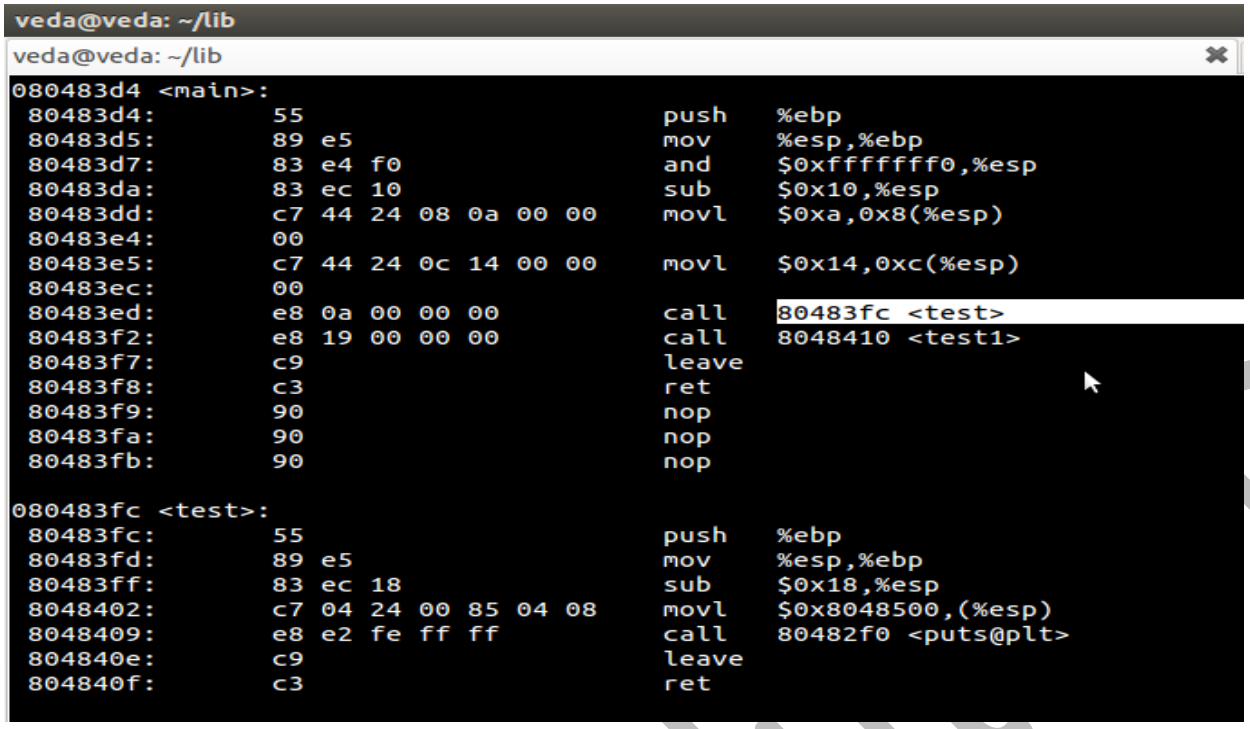
$**gcc -shared -o libtest.so one.o two.o**

➢ Then compile the above **test.c** using dynamic library **libtest.so**.

$**gcc test.c -o testdyn ./libtest.so**

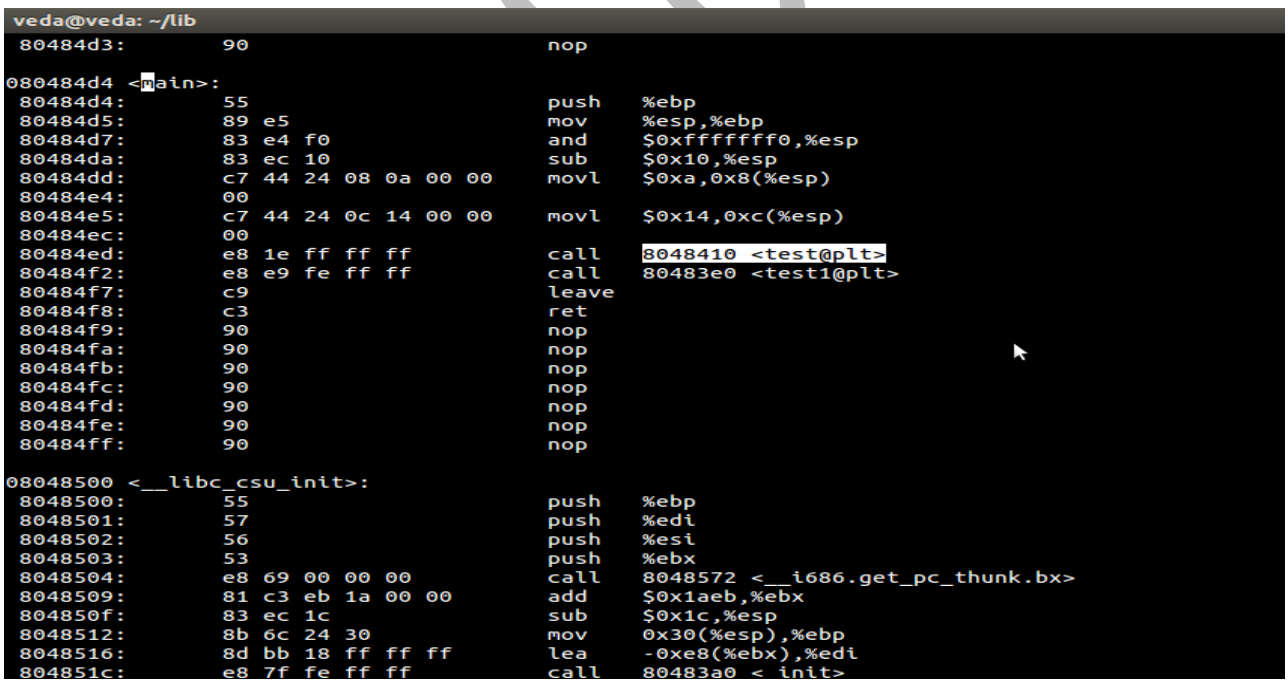
In both the cases output will be same. In order to understand the major differences between static and dynamic linking, we will analyze the executables using objdump tool.

$ **objdump –D teststat** [static executable]

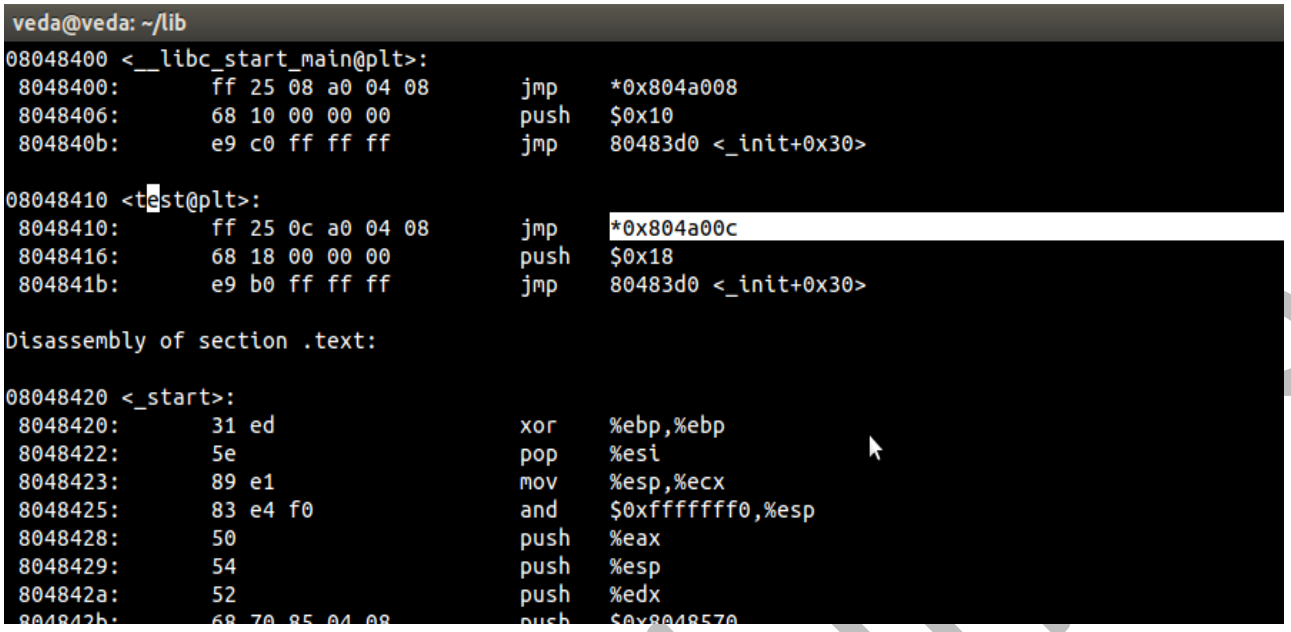


If we analyze the main function in objdump file we could see that test function base address is given and it is called directly.

$ **objdump –D testdyn** [dynamic executable]



Here in dynamic executables main function, it is calling test@plt , (plt= procedure linkage table). Plt table is generated by linker and contains information about dynamic linking.



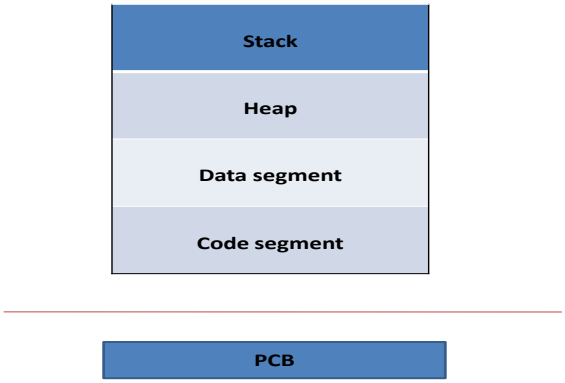
Above figure shows the plt table. In that table we could find a **function pointer**, which derefers to our particular function **test**.

USE CASES

- Static executables occupies more disk space but it has zero initialization time.
- Dynamic executables consumes less disk space but it consumes n amount of cpu cycles for initialization.
- Static builds are preferred if executables are being built for a specific use and will be used in a resource constrained environment where initialization delays are not tolerable.
- Dynamic builds are preferred for easier customization, maintenance and extensions for an application.

Process Control Block (PCB) - Class Notes

- Every process has an address space in user mode and PCB in kernel mode as shown below.
- User mode address space consists of code segment, data segment, heap and stack.



- **PCB** is a kernel structure whose instance is created for each process to store process information.
- PCB is created when the process is loaded and de-allocated when the application is fully terminated
- Linux kernel uses a structure called **task_struct** to implement process control block (PCB). This structure can be found in the Linux source code

  $vim include/linux/sched.h

Important elements of PCB:

- **task_struct** contains following
  - Process identification information (PID ,PPID ,CPID)
  - Process scheduling policy and state
  - Information about number of user level threads mapped to the process.
- **mm_struct** gives process address space information
- **tty_struct** gives the active terminal information

- **files_struct** gives the information about the files which the application currently using. It is called file descriptor table.
- **signal _struct** gives the signal information.

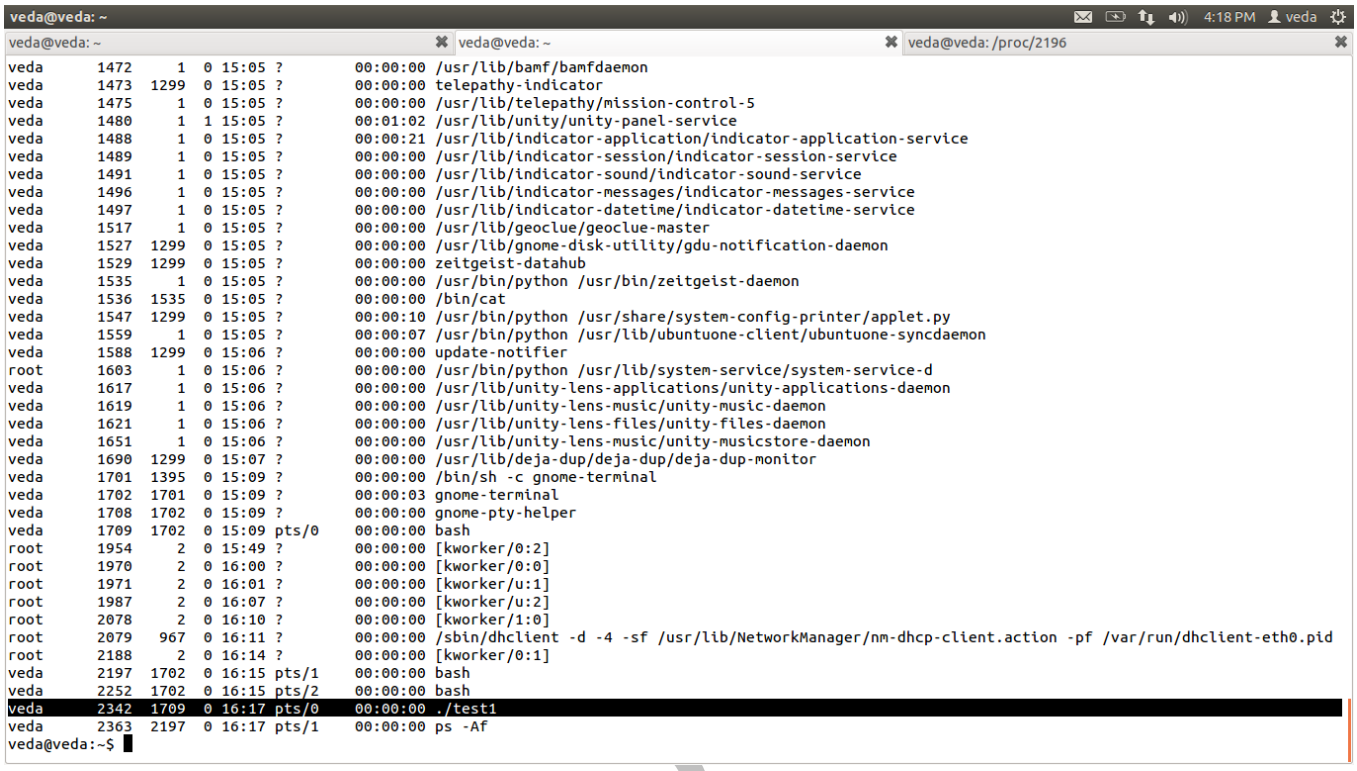
➢ Users can see PCB through **/proc** directory. In order to understand more let's run a simple C file and execute the output. We use a getcar() to put it in a wait state.

```
veda@veda: ~
veda@veda: ~
#include<stdio.h>

int main()
{
        int a = 400;
        printf("%d\n", a);
        getchar();
        return 0;
}
~
~
~
```

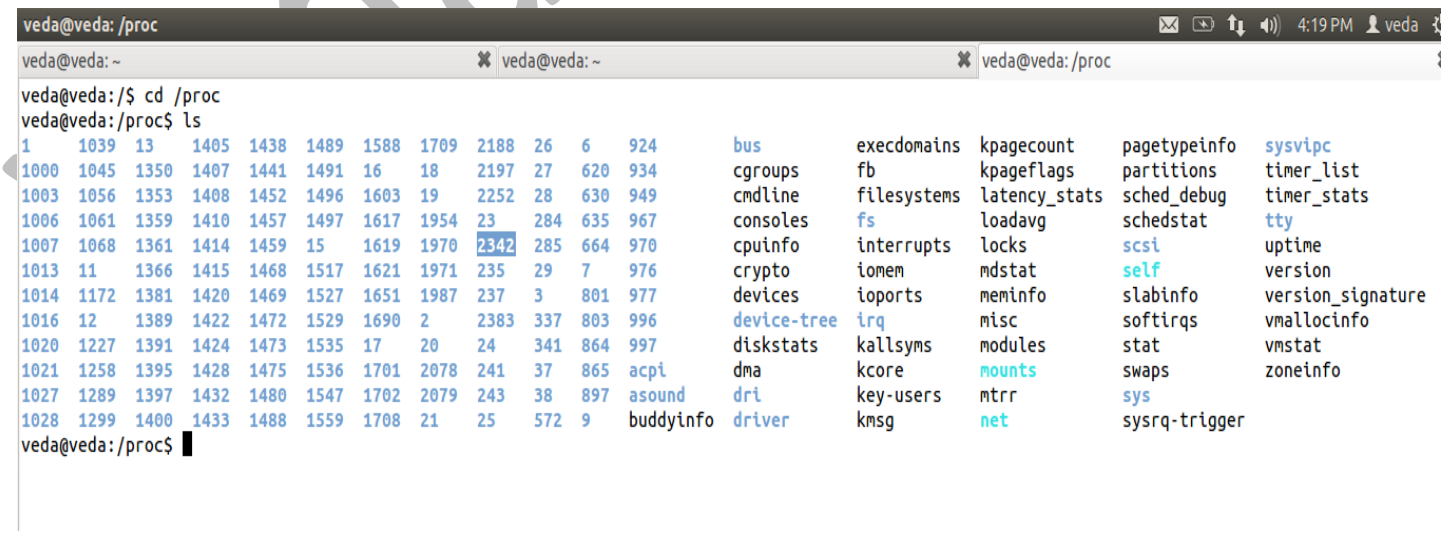
Compile and run the above program.

```
veda@veda: ~
veda@veda: ~
veda@veda:~$ gcc test1.c -o test1
veda@veda:~$ ./test1
400
```

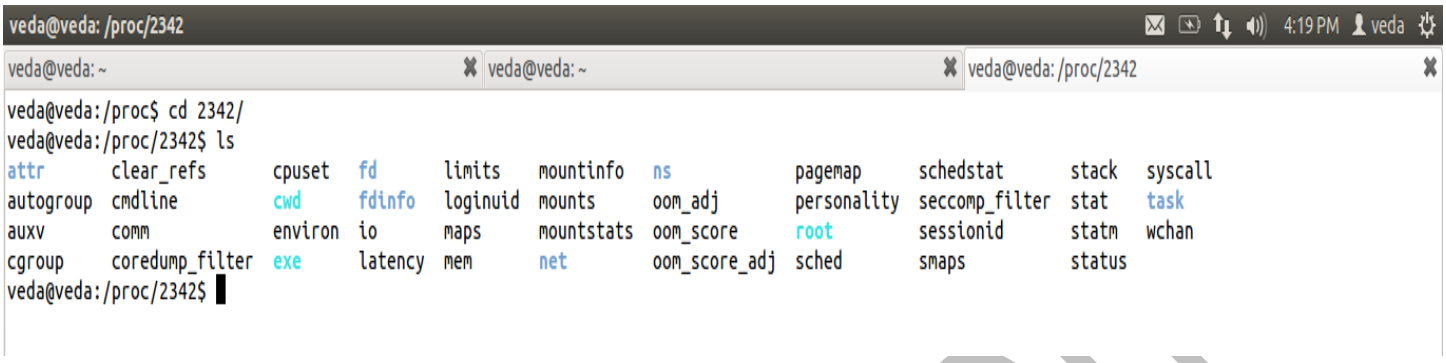
- **Ps -Af** is the command used to see the current running process in the kernel. It also shows the PID of the processes. It gives the **PID** of our program t**est1**



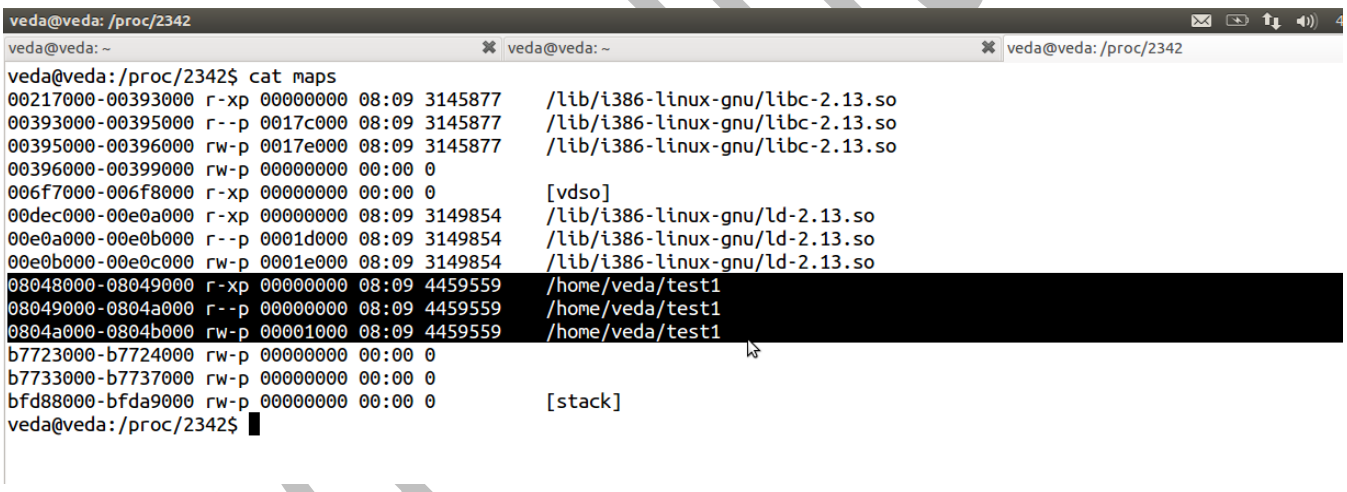
➢ Then go to /**proc**, there we will find our **pid**.



- If we go to our particular pid, it shows **task_struct** tree as shown below.



- In **PID** directory we can see a file maps, if we execute $**cat maps,** which shows **mm_struct** i.e. it shows the addresses for the particular process as shown below. It also shows the addresses of libraries linked to it.

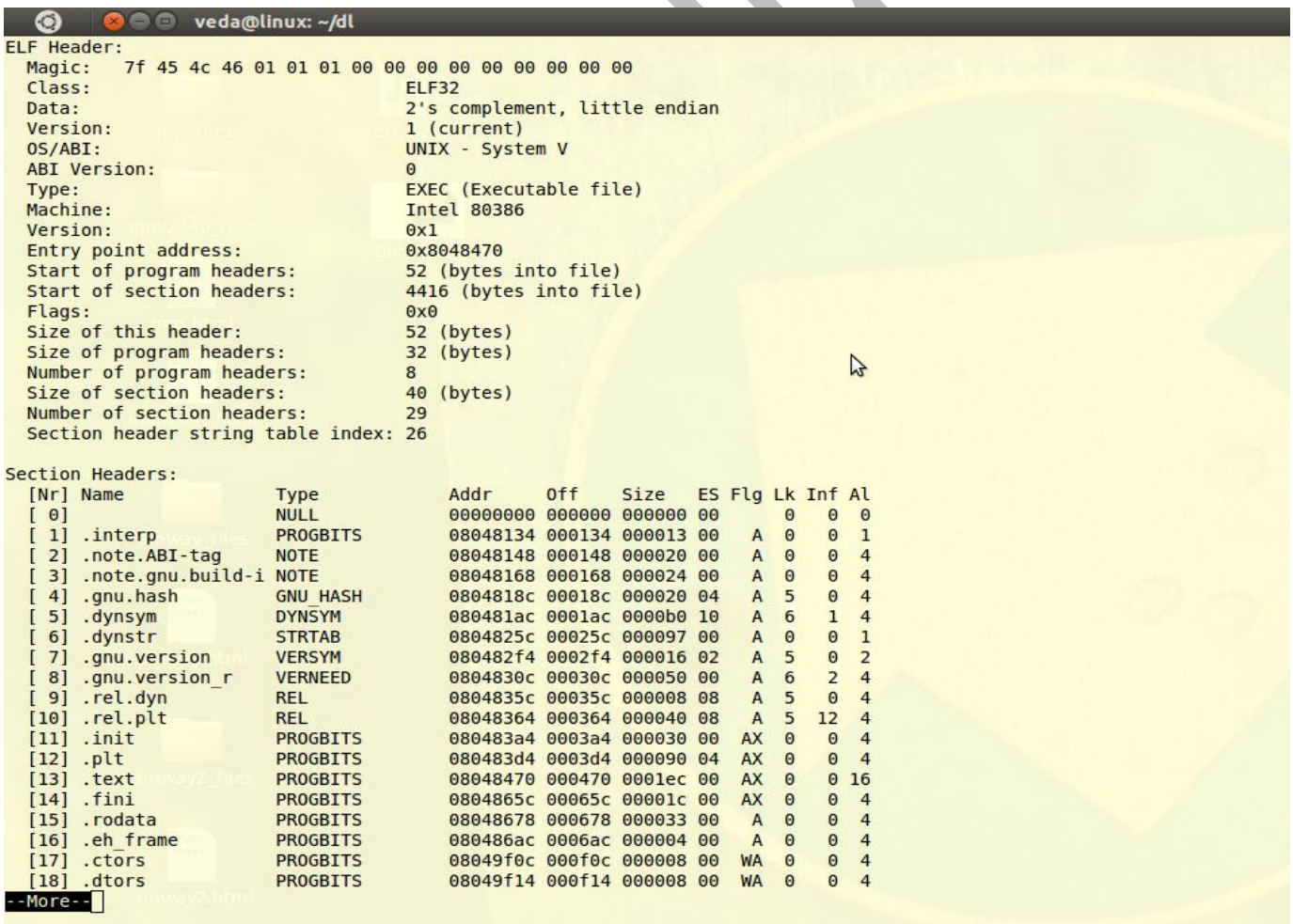


- Thus **/proc** is a directory which shows the process table and addresses mapped to a particular process.

ELF – Class Notes

- Executable binary files are constructed or build using a binary file format standard.
  E.g. ELF
- Most of the binary file formats organize executable image in the binary file as per the platforms memory model.
- Executable file header provides detail about the platform for which the file is built and information about sections and segments that are part of executable image.
- To view the ELF header file of any executable, use a tool called **readelf.** Let's consider one executable image **app.**

$**readelf -a app | more**



Elf header for an executable provides following important details

- Platform's application binary interface standard considered while building the executable. ABI standard specifies the following:
    - Addressing format to be used by assembler and linker while binding machine instructions to a corresponding address
    - Function calling conventions (fast call , standard call , windows calling conventions)
    - Binary file format to be used to build object files
- It specifies object file type, mainly three types relocatable, executable, shared object.
- It specifies base address or entry point address of code segment (usually base address is _start)
- Offset of the program header table in the file

Load time and Run time Libraries

Load time Libraries:

- When a shared object is linked into application address space during application initialization, it is referred as a load time library.
- All symbols linked into executables from a dynamic library are resolved during application initialization.
- Load time libraries remain resident in the process until application terminates.

Run time Libraries:

- Applications at run time could raise a request to load shared objects and bind them into address space. Such shared objects can also be detached again if application makes a request.
- If an application intends to use a shared object as runtime library, all direct references to library symbol must be avoided.

In order to understand **run time libraries** in detail, let's take a simple **C** program **test.c** which calls a shared library **libxyz.so,** where the function '**add'** resides.

$vim test.c

```
veda@linux: ~/dl
#include<stdio.h>
#include<dlfcn.h>

main()
{
        char *ptr;
        int i;
        void *handle;
        int (*fnptr)(int , int);
        getchar();
        handle=dlopen("./libxyz.so",RTLD_NOW);
                if(handle==NULL) {
                        printf("\n Failed \n");
                }
        fnptr = dlsym(handle,"add"); /* it returns the address of the function */
        getchar();
        i=(fnptr)(20,20);
        printf("\n the result: %d",i);
        dlclose(handle);
        getchar();

}
~
~
~
```

In the program **getchar()** is included to put the process under wait stage. Compile it by including **–ldl,** where the **dlopen ,dlsym , dlclose** are defined.

$**gcc test.c -o test -ldl**

Execute the output, on the first **getchar()** . Get the **PID** of our process using **ps –Af** and check $**cat /proc/2104/maps**.

It shows as below

```
veda@linux: ~/dl
veda@linux: ~/dl                                                    veda@linux: ~/
veda@linux:~/dl$ cat /proc/2104/maps
00110000-0026c000 r-xp 00000000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026c000-0026e000 r--p 0015c000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026e000-0026f000 rw-p 0015e000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026f000-00272000 rw-p 00000000 00:00 0
004ed000-004ee000 r-xp 00000000 00:00 0          [vdso]
006fc000-00718000 r-xp 00000000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00718000-00719000 r--p 0001b000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00719000-0071a000 rw-p 0001c000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00b64000-00b66000 r-xp 00000000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
00b66000-00b67000 r--p 00001000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
00b67000-00b68000 rw-p 00002000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
08048000-08049000 r-xp 00000000 08:01 9045280    /home/veda/dl/dlopen
08049000-0804a000 r--p 00000000 08:01 9045280    /home/veda/dl/dlopen
0804a000-0804b000 rw-p 00001000 08:01 9045280    /home/veda/dl/dlopen
b7740000-b7742000 rw-p 00000000 00:00 0
b7754000-b7757000 rw-p 00000000 00:00 0
bfed5000-bfef6000 rw-p 00000000 00:00 0          [stack]
veda@linux:~/dl$
```

It shows our library **libxyz.so** has not loaded.

Execute the second **getchar()** and check the maps again.

```
veda@linux: ~/dl
veda@linux:~/dl$ cat /proc/2104/maps
00110000-0026c000 r-xp 00000000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026c000-0026e000 r--p 0015c000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026e000-0026f000 rw-p 0015e000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026f000-00272000 rw-p 00000000 00:00 0
004ed000-004ee000 r-xp 00000000 00:00 0          [vdso]
006fc000-00718000 r-xp 00000000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00718000-00719000 r--p 0001b000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00719000-0071a000 rw-p 0001c000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00737000-00738000 r-xp 00000000 08:01 9045282    /home/veda/dl/libxyz.so
00738000-00739000 r--p 00000000 08:01 9045282    /home/veda/dl/libxyz.so
00739000-0073a000 rw-p 00001000 08:01 9045282    /home/veda/dl/libxyz.so
00b64000-00b66000 r-xp 00000000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
00b66000-00b67000 r--p 00001000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
00b67000-00b68000 rw-p 00002000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
08048000-08049000 r-xp 00000000 08:01 9045280    /home/veda/dl/dlopen
08049000-0804a000 r--p 00000000 08:01 9045280    /home/veda/dl/dlopen
0804a000-0804b000 rw-p 00001000 08:01 9045280    /home/veda/dl/dlopen
088b8000-088d9000 rw-p 00000000 00:00 0          [heap]
b7740000-b7742000 rw-p 00000000 00:00 0
b7754000-b7757000 rw-p 00000000 00:00 0
bfed5000-bfef6000 rw-p 00000000 00:00 0          [stack]
veda@linux:~/dl$
```

Here we will find our library **libxyz.so** , which has loaded into the process.

Then execute the third **getchar()** and check maps again

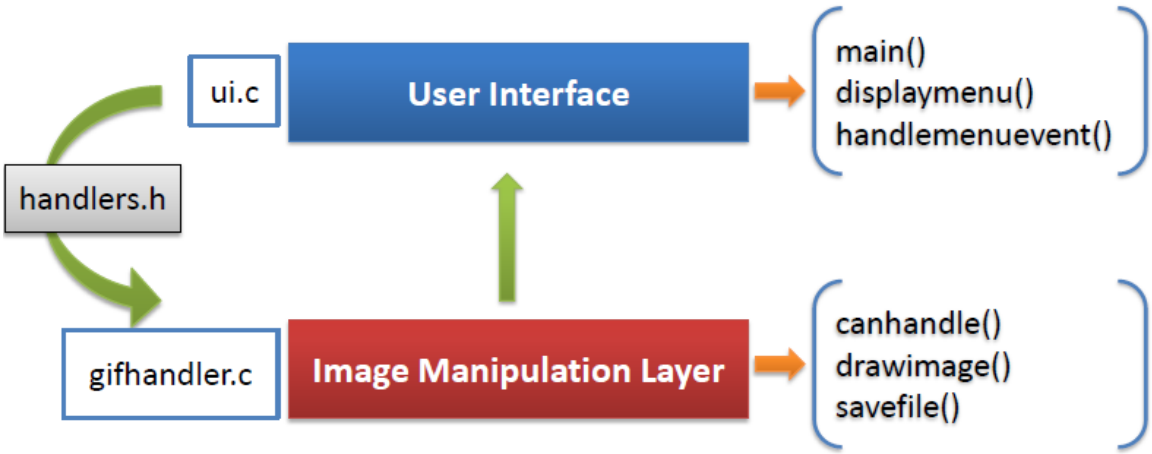
```
veda@linux: ~/dl
veda@linux:~/dl$ cat /proc/2104/maps
00110000-0026c000 r-xp 00000000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026c000-0026e000 r--p 0015c000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026e000-0026f000 rw-p 0015e000 08:01 18623683   /lib/i386-linux-gnu/libc-2.13.so
0026f000-00272000 rw-p 00000000 00:00 0
004ed000-004ee000 r-xp 00000000 00:00 0          [vdso]
006fc000-00718000 r-xp 00000000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00718000-00719000 r--p 0001b000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00719000-0071a000 rw-p 0001c000 08:01 18623680   /lib/i386-linux-gnu/ld-2.13.so
00b64000-00b66000 r-xp 00000000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
00b66000-00b67000 r--p 00001000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
00b67000-00b68000 rw-p 00002000 08:01 18623686   /lib/i386-linux-gnu/libdl-2.13.so
08048000-08049000 r-xp 00000000 08:01 9045280    /home/veda/dl/dlopen
08049000-0804a000 r--p 00000000 08:01 9045280    /home/veda/dl/dlopen
0804a000-0804b000 rw-p 00001000 08:01 9045280    /home/veda/dl/dlopen
088b8000-088d9000 rw-p 00000000 00:00 0          [heap]
b7740000-b7742000 rw-p 00000000 00:00 0
b7753000-b7757000 rw-p 00000000 00:00 0
bfed5000-bfef6000 rw-p 00000000 00:00 0          [stack]
veda@linux:~/dl$
```

Here it doesn't shows our library which clearly explains the functionality of a run time library. It gets loaded when the program calls it and gets unloaded as soon as the function gets over.

Case Study on Application Designs – Class Notes

- Let's design an application that can take an image file as input and display its contents. It should also provide image conversion facility.

Approach 1:

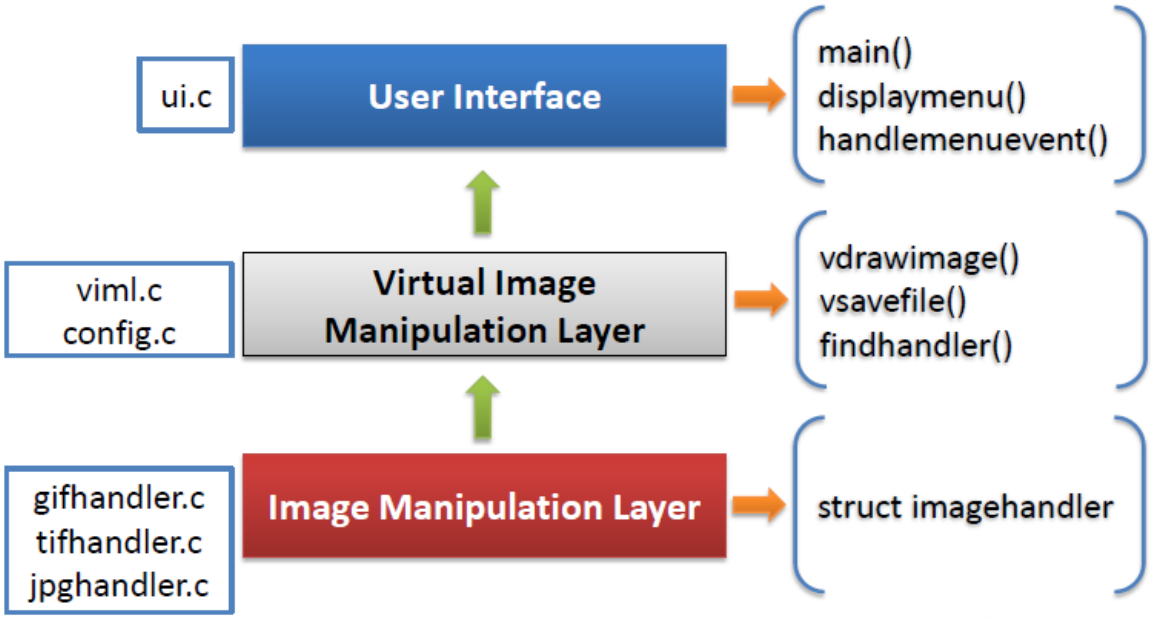


- The above approach displays gif format files. It provides a user interface which provides a display menu and it takes the input from the user and handles it.

The limitation of the above approach is it supports only gif format files; it cannot handle any other file formats.

- Redesign the above application to meet the following objectives.
  - Add support for at least 3 different image formats
  - Proposed new implementation should allow future extensions with minimal or no changes to existing code
  - Achieve objectives 1 & 2 with minimal changes to current implementation.

Approach 2:



Design changes:

- Image manipulation layer will be modified to accommodate three separate image handlers
- A new module called Virtual Image Manipulation layer (VIML) is introduced to abstract user interface from handlers in IML (Image Manipulation Layer).

VIML should provide following:

- A common interface (file format independent) for UI (User Interface) layer to request image processing and conversion.
- An interface for the image handlers to register with VIML.
- When UI submits a request, request processing logic should be able to locate and invoke appropriate handler.

Code changes:

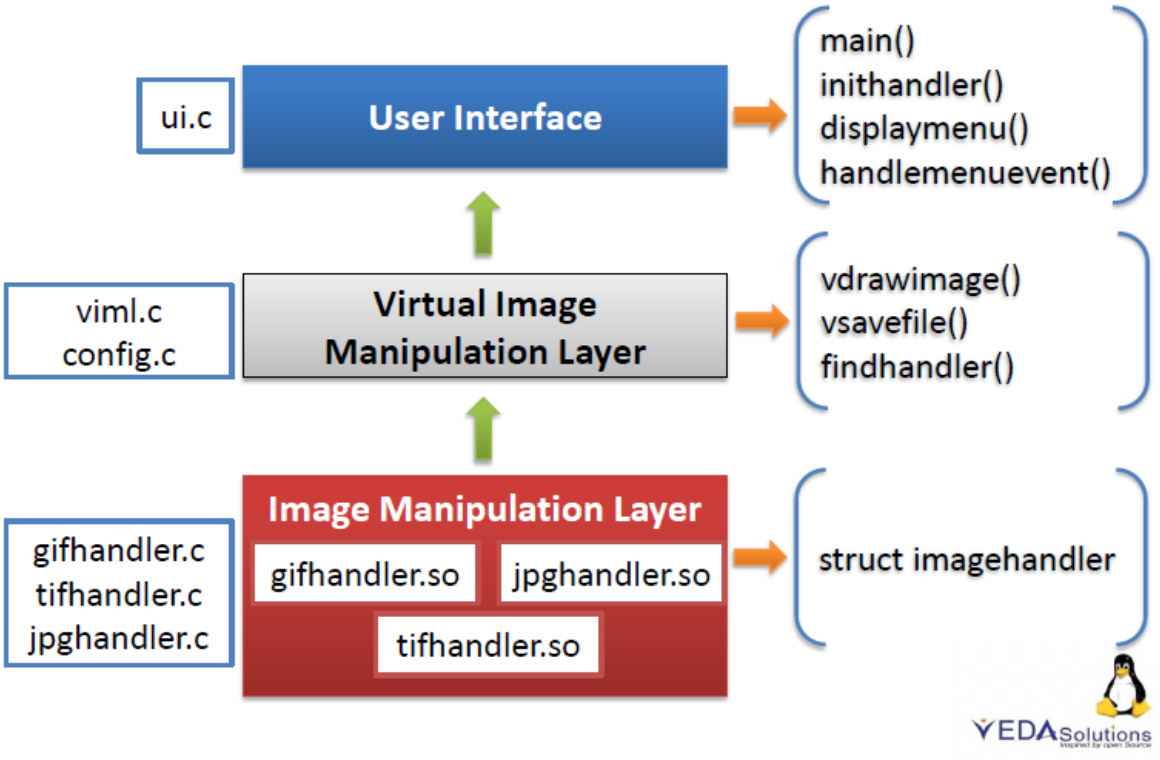
- Modify **ui.c** to invoke functions of VIML.
- Implement VIML source code as per the design needs.
- Each handler should implement its image processing routines and store their addresses in an object of type image handler
- Image handler objects of each handler will be stored in a data structure maintained by VIML.

Limitations of approach 2:

- The current design is extensible but only when plug-in developers have access to source code
- Each extension requires a rebuilt.

Redesign the approach 2 to add support for binary extensibility

Approach 3:



Design changes in approach 3:

- All handlers in IML layer will be built into independent shared objects.
- Application executable image will comprise of user interface and VIML layers.

Code changes:

- Main function in the user interface module shall call a routine init_handler(), that loads all handlers (**.so** s) before invoking display menu.
- VIML handler provides two new functions at **config.c** that can add an image handler object to VIML vector and remove the object when it is no longer needed. They are :

  reg_handler and unreg_handler
- Each of the handlers of the IML layer would provide their own implementations of **_init** and **_fini** (over writing default run time routines).

Linux System Calls

Class Notes

System Calls

- System calls are kernel functions which serve as an interface for kernel services.
- Applications in the user mode use system calls to transmit into kernel mode and execute kernel service operations.

Adding System Call to Linux Source Code

- Download stable kernel source code (www.kernel.org)
- Unzip the downloaded source code using **tar** command and enter into kernel source directory.
- $tar xvf <linux-source>
- I used the kernel linux-3.0. After entering into kernel source, we can see kernel tree as below

- Reserve unique ID for the system call
  - $**vim arch/x86/include/asm/unistd_32.h**
  - **lsproc** is the function name and manually assign the number to our syscall.
  - Add the system call as shown below

- Declare system call prototype

  $**vim arch/x86/include/asm/syscalls.h**

```
asmlinkage int sys_set_thread_area(struct user_desc __user *);
asmlinkage int sys_get_thread_area(struct user_desc __user *);

/* X86_32 only */
#ifdef CONFIG_X86_32

/* kernel/signal.c */
asmlinkage int sys_sigsuspend(int, int, old_sigset_t);
asmlinkage int sys_sigaction(int, const struct old_sigaction __user *,
                             struct old_sigaction __user *);
unsigned long sys_sigreturn(struct pt_regs *);

/* kernel/vm86_32.c */
int sys_vm86old(struct vm86_struct __user *, struct pt_regs *);
int sys_vm86(unsigned long, unsigned long, struct pt_regs *);

asmlinkage int sys_lsproc(void);

#else /* CONFIG_X86_32 */

/* X86_64 only */
```

- Assign system call address to appropriate offset of the system call switch table

  $**vim arch/x86/kernel/syscall_table_32.S**

```
        .long sys_preadv
        .long sys_pwritev
        .long sys_rt_tgsigqueueinfo     /* 335 */
        .long sys_perf_event_open
        .long sys_recvmmsg
        .long sys_fanotify_init
        .long sys_fanotify_mark
        .long sys_prlimit64             /* 340 */
        .long sys_name_to_handle_at
        .long sys_open_by_handle_at
        .long sys_clock_adjtime
        .long sys_syncfs
        .long sys_sendmmsg              /* 345 */
        .long sys_setns
        .long sys_lsproc
```

- Implement syscall routine, syscall routine can be added to any source branch either by modifying existing source file or appending a new source. Here we have added to an existing source file **sys.c**

```
                kernel_power_off();
        }

        return ret;
}
EXPORT_SYMBOL_GPL(orderly_poweroff);

asmlinkage int sys_lsproc (void)
{
        struct task_struct *p;
        printk("\n \t process \t pid \t state \n");
        for_each_process(p){
                        printk("\n %15s \t %u \t %ld",p->comm,task_pid_nr(p),p->state);
        }
        return 0;
}
```

Compiling Linux kernel and building kernel image (x86-32-bit arch)

All of the following commands must be executed in the kernel source root folder

- Assign unique build name: Open root make file and assign build name to EXTRAVERSION macro.
  - $**vim Makefile**

```
VERSION = 3
PATCHLEVEL = 0
SUBLEVEL = 0
EXTRAVERSION =.syscall
NAME = Sneaky Weasel
```

- Choose kernel configuration

- $**make menuconfig**

  Select the required configuration, then save and exit. The configurations are saved on **.config** file.

- Initiate compile and build process
  - $**make**

  If the build got finished successfully, we could find a file **vmlinux** in the kernel source root folder.

- Install kernel headers and module objects.
  - $**make modules_install**

  Above command creates a folder in **cd /lib/modules** with the name of our syscall. Only root user can perform the above operation.

- Modify boot loader configuration with updated kernel image name and path.
  - $**make install**

- After completion, **reboot** the system. After rebooting verify the running kernel using
  - $**uname –r**

Invoking System Call

- Store syscall ID in the processor's accumulator (%eax).
- Starting with right most argument move each argument on to other processor accumulators (**ebx –eix**).
- Generate trap exception using processor specific instruction.
- Read return value of the system call from **eax** accumulator.

```c
#include<stdio.h>
#include<stdlib.h>

int lsproc()
{
        int ret;
        __asm__("movl $347,%eax");
        __asm__("int $0x80");
        __asm__("movl %eax,-4 (%ebp)");
        return ret;
}

int main()
{
        int ret;
        printf("Invoking System Call \n");
        ret=lsproc();
        if (ret < 0)
                exit (1);
        return 0;
}
```

- Using above C program we can invoke our system call. Execute it and check the output. Use **dmesg** to see the **printk** messages.

END

Stack Trace

Class Notes

Primary memory (RAM) is logically divided into two regions

- **Kernel space**: The users cannot read or write to these addresses, doing so will result in segmentation fault.

- **User space** : It consists of **Stack segment**(which grows down) , **memory mapping segment** (file and anonymous mappings), **BSS segment** (uninitialized variables filled with zeros) , **Data segment** (static variables initialized) ,**Text segment** (stores the binary images of processes)

Stack:

- Stack is a segment mapped to address space which is used to allocate stack frames to store local data of the processes.

- A stack frame is initialized during function initialization and it is released when a function returns or terminates.

- Stack frames are of two types: Local data frames, Argument frames.

Translating source functions into assembly equivalent :

- To understand the functionality of stack let us consider one C program.

```
main()
{
 int a=100;
 int b=200;
 int c;
 c = a+b;
}
```

Steps involved in translating C code into assembly equivalent:

Assumption:

- Code translation x86 32-bit architecture with GNU compiler.

Translation steps:

- Identify non executable instructions within source file
- Create a function symbol table and resolve all non executable declarations
- Translate executable instruction into assembly source as per the following template.

Function name:

Pre-amble

Function body;

Post amble

Symbol table:

- A table which holds all the information about the variables declared.

```
main()
{
        int a=100;
        int b=200;        } Non executable statements
        int c;
        c = a+b;
}
```

Symbol Name	Type	Composition	Offset Address
a	int	4	-12(%ebp)
b	int	4	-8(%ebp)
c	int	4	-4(%ebp)
	Total	12	

Translating Source Code to Assembly:

```
main()
{
        int a=100;
        int b=100;
        int c;
        c = a+b;
}
```

```
main:
        pushl   %ebp
        movl    %esp,   %ebp
        subl    $12,    %esp
        movl    $100,   -12(%ebp)
        movl    $100,   -8(%ebp)
        movl    -8(%ebp),   %eax
        addl    -12(%ebp),  %eax
        movl    %eax,   -4(%ebp)
        leave
        ret
```

pre - amble

post - amble

- Above shows the assembly code of a main function, which does not calls any other function.

Steps for translating function-calls:

- Starting from the right most argument push each parameter on to top of stack
- Step into called functions using call instruction
- Read the return value of the function from %eax accumulator
- Release the space allocated for arguments.

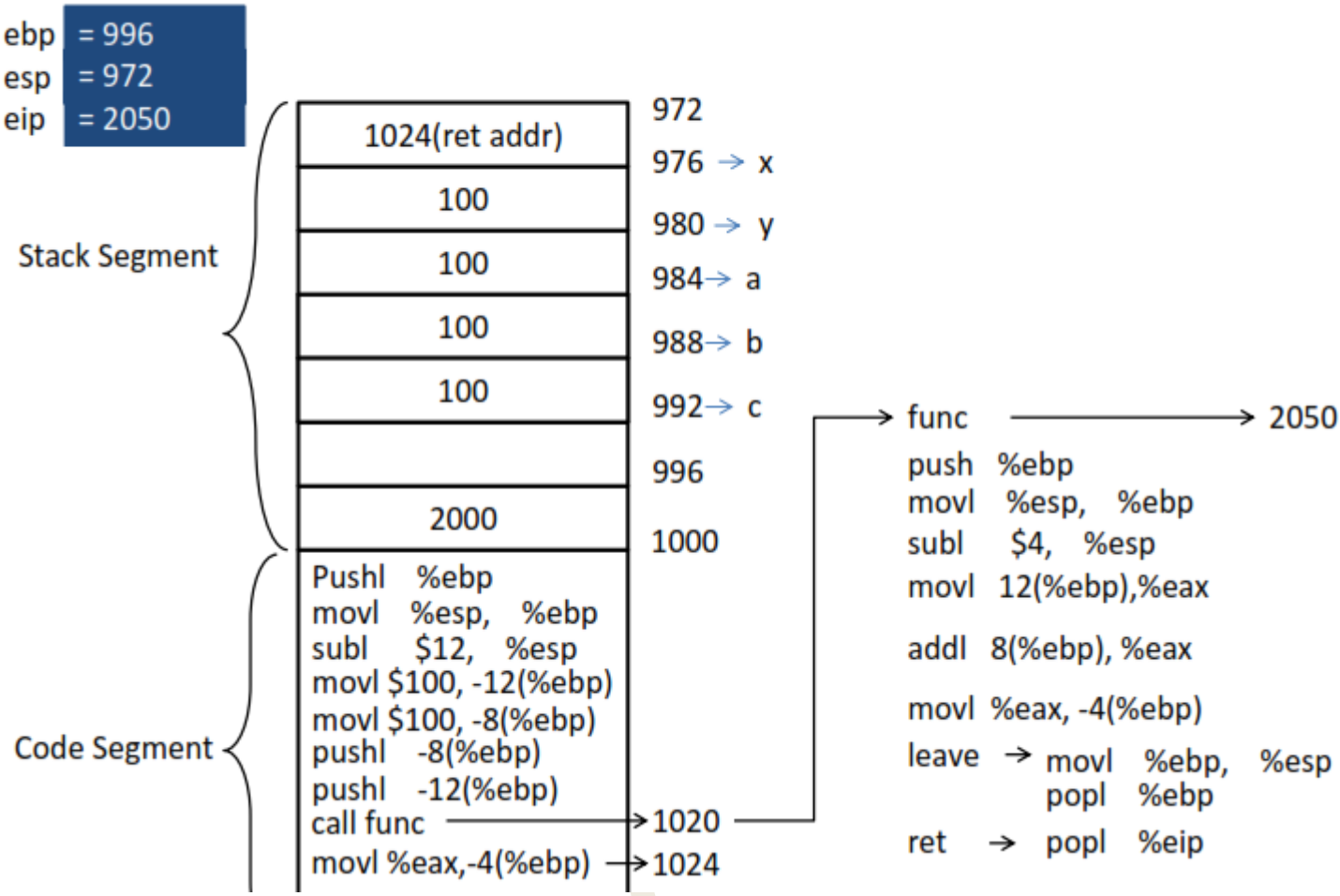
Now let's see the assembly code for a C program which calls a function.

```
main()
{
        int a=100;
        int b=100;
        int c;
        c = func(a,b);
}
```

```
main:
        pushl   %ebp
        movl    %esp,   %ebp
        subl    $12,    %esp
        movl    $100,   -12(%ebp)
        movl    $100,   -8(%ebp)
        pushl   -8(%ebp)
        pushl   -12(%ebp)
        call    func
        movl    %eax, -4(%ebp)
        addl    $8,   %esp
```

- The pre-amble (**{**) are a pair of instructions responsible for initializing stack frame.
- **Call** instruction in the above code executes the following steps
    1) Stores the return address of the caller function on top of the stack.
    2) Assigns the instruction pointer with the base address of called function.
    3) With respect to position of **ebp**, arguments reside in higher addresses. Local data is allocated in local addresses.

The stack for the above program is shown below



Heap Allocation

Class Notes

Heap allocation:

- Each process has virtual address space reserved for heap allocations.
- Process memory descriptor contains a record of current start address of heap (**start_brk**) and current break address (**program_brk**).
- To allocate heap program **brk** should be moved to higher address. Linux provides an API called **brk** to alter heap break point.

```
BRK(2)                                                    Linux

NAME
       brk, sbrk - change data segment size

SYNOPSIS
       #include <unistd.h>

       int brk(void *addr);

       void *sbrk(intptr_t increment);
```

Memory Mapped Segment Allocation:

- Applications can allocate dynamic memory in memory mapped segment of the virtual address space.
- mmap can be used to map any of the following buffer regions to process address space.
  1) Anonymous buffer
  2) File buffer
  3) Device buffer (address region)

```
MMAP(2)                          Linux Programmer's Manual                          MMAP(2

NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #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);
```

- The **first** argument takes the start address of the offset. **Second** argument takes the size of the mapping, **third** argument takes the memory access attributes, **fourth** argument tells the size and scope of mapping, **fifth** argument takes the file descriptor mapped to the buffer and the **sixth** argument takes the start offset of the file region to be mapped.

Memory mapping vs. heap allocation:

- Allocation from **heap** segment is achieved by altering heap break point.
- Releasing memory from **heap** segment would require the break point to be decremented by 'n' bytes.
- When multiple allocations are carried out through **brk** and **sbrk,** second block would begin, right at address where the first block ends. It will not be possible to free the first block without releasing entire zone.
- **mmap** calls create a physical region mapping of a specified size into address space.
- Any **memory mapped** region can be released to system.

Note:

- Physical memory mapped to heap (memory mapped region) are maintained in the process **pcb** memory description table, using an **API** called **mincore** an application can probe physical mapping status.

```
MINCORE(2)                                                Linux Programmer's Manu

NAME
       mincore - determine whether pages are resident in memory

SYNOPSIS
       #include <unistd.h>
       #include <sys/mman.h>

       int mincore(void *addr, size_t length, unsigned char *vec);
```

Malloc implementation in glibc

- GNU C library provides a set of routines to optimize, tune and observe dynamic memory allocation of C library.

  mallopt, malloc_trim, malloc_usable_size
- Glibc malloc allocates approximate 128k bytes on the first malloc call from the application.
- The region is retained until the program termination and all dynamic memory requests would be processed from this block of 128k.
- Malloc uses mmap interface to serve requests beyond a fixed threshold (configurable). The default threshold value is set to 128k (i.e. up to 128 k size) and it uses heap and mmap interface beyond 128k

Applications running on Linux can choose to disable demand paging either on a region of the address space or the entire process address space using **mlock** and **mlockall**

```
MLOCK(2)                        Linux Programmer's Manual                       MLOCK(2)

NAME
       mlock, munlock, mlockall, munlockall - lock and unlock memory

SYNOPSIS
       #include <sys/mman.h>

       int mlock(const void *addr, size_t len);
       int munlock(const void *addr, size_t len);

       int mlockall(int flags);
       int munlockall(void);

DESCRIPTION
       mlock()  and  mlockall()  respectively lock part or all of the calling process's virtual address space into RAM, preventing that memory from
       being paged to the swap area.  munlock() and munlockall() perform the converse operation, respectively unlocking part or all of the  calling
       process's  virtual  address  space,  so that pages in the specified virtual address range may once more to be swapped out if required by the
       kernel memory manager.  Memory locking and unlocking are performed in units of whole pages.
```

Heap Allocation

Class Notes

Heap allocation:

- Each process has virtual address space reserved for heap allocations.
- Process memory descriptor contains a record of current start address of heap (**start_brk**) and current break address (**program_brk**).
- To allocate heap program **brk** should be moved to higher address. Linux provides an API called **brk** to alter heap break point.

```
BRK(2)                                                    Linux

NAME
       brk, sbrk - change data segment size

SYNOPSIS
       #include <unistd.h>

       int brk(void *addr);

       void *sbrk(intptr_t increment);
```

Memory Mapped Segment Allocation:

- Applications can allocate dynamic memory in memory mapped segment of the virtual address space.
- mmap can be used to map any of the following buffer regions to process address space.
  1) Anonymous buffer
  2) File buffer
  3) Device buffer (address region)

```
MMAP(2)                          Linux Programmer's Manual                          MMAP(2

NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #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);
```

- The **first** argument takes the start address of the offset. **Second** argument takes the size of the mapping, **third** argument takes the memory access attributes, **fourth** argument tells the size and scope of mapping, **fifth** argument takes the file descriptor mapped to the buffer and the **sixth** argument takes the start offset of the file region to be mapped.

Memory mapping vs. heap allocation:

- Allocation from **heap** segment is achieved by altering heap break point.
- Releasing memory from **heap** segment would require the break point to be decremented by 'n' bytes.
- When multiple allocations are carried out through **brk** and **sbrk,** second block would begin, right at address where the first block ends. It will not be possible to free the first block without releasing entire zone.
- **mmap** calls create a physical region mapping of a specified size into address space.
- Any **memory mapped** region can be released to system.

Note:

- Physical memory mapped to heap (memory mapped region) are maintained in the process **pcb** memory description table, using an **API** called **mincore** an application can probe physical mapping status.

```
MINCORE(2)                                                Linux Programmer's Manu

NAME
       mincore - determine whether pages are resident in memory

SYNOPSIS
       #include <unistd.h>
       #include <sys/mman.h>

       int mincore(void *addr, size_t length, unsigned char *vec);
```

Malloc implementation in glibc

- GNU C library provides a set of routines to optimize, tune and observe dynamic memory allocation of C library.

  mallopt, malloc_trim, malloc_usable_size
- Glibc malloc allocates approximate 128k bytes on the first malloc call from the application.
- The region is retained until the program termination and all dynamic memory requests would be processed from this block of 128k.
- Malloc uses mmap interface to serve requests beyond a fixed threshold (configurable). The default threshold value is set to 128k (i.e. up to 128 k size) and it uses heap and mmap interface beyond 128k

Applications running on Linux can choose to disable demand paging either on a region of the address space or the entire process address space using **mlock** and **mlockall**

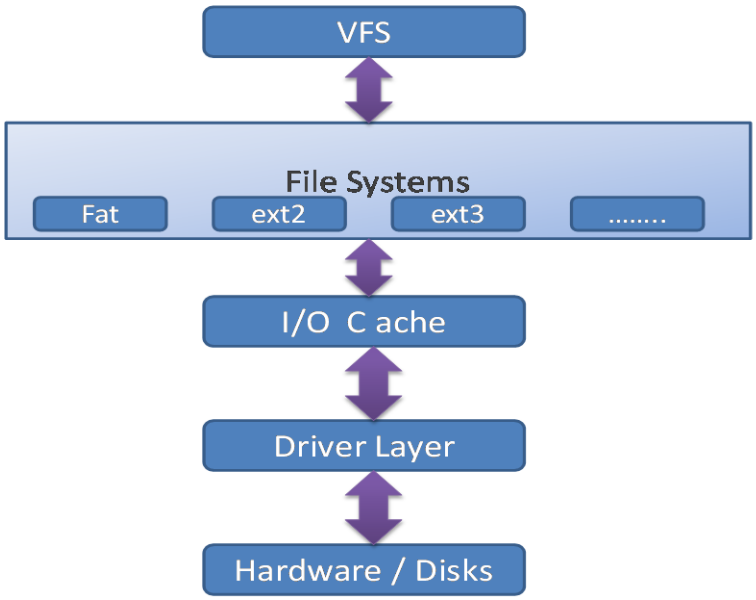
```
MLOCK(2)                        Linux Programmer's Manual                       MLOCK(2)

NAME
       mlock, munlock, mlockall, munlockall - lock and unlock memory

SYNOPSIS
       #include <sys/mman.h>

       int mlock(const void *addr, size_t len);
       int munlock(const void *addr, size_t len);

       int mlockall(int flags);
       int munlockall(void);

DESCRIPTION
       mlock()  and  mlockall()  respectively lock part or all of the calling process's virtual address space into RAM, preventing that memory from
       being paged to the swap area.  munlock() and munlockall() perform the converse operation, respectively unlocking part or all of the  calling
       process's  virtual  address  space,  so that pages in the specified virtual address range may once more to be swapped out if required by the
       kernel memory manager.  Memory locking and unlocking are performed in units of whole pages.
```

Linux I/O Architecture

Class Notes

Linux I/O Architecture

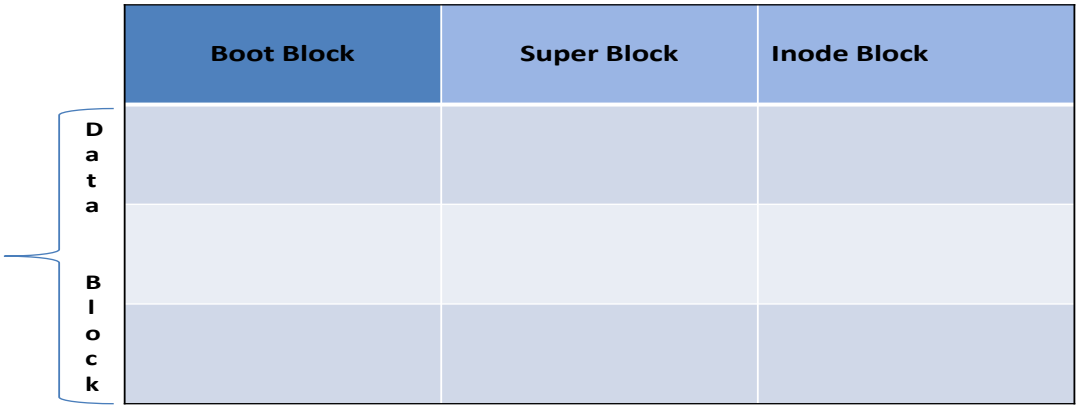
- Linux I/O architecture provides a common API interface for the applications to initiate I/O transfers on various soft and hard resources.



File systems

- File systems organized disk into four
  1) Boot block
  2) Super block
  3) Inode block
  4) Data block
- Mounting is the process of loading super block and inode block from disk to memory which is done by compatible file system.
- For a mount operation to succeed a compatible file system service that can understand the format of the super block and inode table is required.
- Command for mounting in Linux

  mount -t<fs type> **<**source device**>** **<**mount point**>**



- From Boot block, a set of blocks (super block & inode block) are reserved for use of file system called FS block.

Virtual File System (VFS)

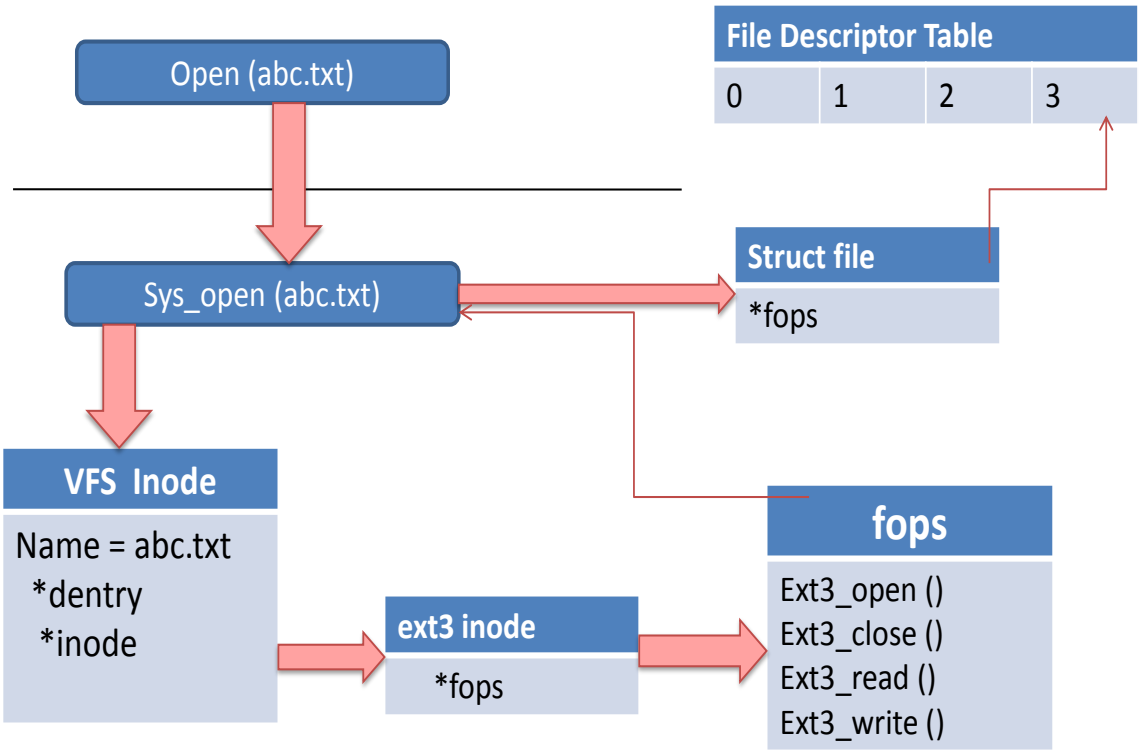
- VFS is a kernel abstraction layer that hides file system operations from the applications.
- This method of abstraction provides the following benefits for the application developers and users.
    1) A limited common API interface to initiate operations on any type of file (irrespective of file system).
    2) All mounted files (disks) can be unified into a single file tree (instead of drives).
    3) Changes to file system layer will not affect existing applications.
    4) Support for new file systems will not require changes to application layer.
- VFS can be used as an application interface to provide other services like IPC, device access, provided respective services register with the VFS.

VFS Request Switch

- When a disk is mounted, inode tree is copied from disk into appropriate file system cache. (Term **cache** referred to reserved region of memory for a specific kernel service usage).

- Virtual file system maintains a file tree in the VFS cache made up of virtual inode instances. Each virtual inode represents a file system inode in VFS cache.

File Systems



- When file system open call gets invoked, it verifies the caller applications request to access file and returns zero or a negative number based on application privilege validation.

- When file system open routine returns zero, sys_open allocates a new file descriptor instance and stores the reference of file operations.
- File descriptor address is stored in the caller process's file descriptor table (Table is implemented as an array).
- The offset number of the file descriptor table is returned to the caller.

File descriptor

- It is an instance of a structure that contains pointer to relevant file operations.
- It also holds application specific file access attributes.
- The address of the file descriptor is stored in the process PCB file descriptor table.
- File descriptors are process local data structures.

File I/O Calls

Class Notes

File I/O calls:

Linux kernel supports the following types of file I/O operations

- Standard I/O
- Synchronized I/O
- Direct I/O
- Read ahead I/O
- Memory mapped I/O
- Vectored I/O

Standard I/O

1) Read

```
#include<unistd.h>
ssize_t  read (int fd, void * buf ,size_t count);
```

Read api internally invokes VFS system call sys_read, which inturn steps into FS specific read call. FS read operation executes the following steps.

- Looks up into specified file node to identify data block number
- Verifies for the data of specified block in the I/O cache
- Invokes storage driver and initiates physical transfer of the data from the specified block.
- Polls I/O cache for arrival of requested data and transfers the data to application when received.

2) Write

```
#include<unistd.h>
ssize_t  write (int fd, void * buf ,size_t count);
```

- Write API invokes FS specific write operation through VFS system call sys_write.
- FS write transfer the data from the application and writes it to specified files buffer located in IO cache.
- Initiate disk sink operation.

Synchronized I/O:

This mode of I/O allows applications to force the file systems to update the target disc immediately when a file write is initiated. The write operations of the application returns only after the disc is updated.

Sample code:

```
fd= open("Demo.txt",O_RDWR |O_SYNC, S_IRWXU |S_IRWXG);
```

Direct I/O:

This mode of I/O disables the kernel I/O cache and allows application to setup usermode buffer that can be used as an I/O cache.

```
#include <stdlib.h>

int posix_memallign (void ** memptr, size_t allignment ,size_t size);
```

Read ahead I/O:

- FS are designed to instruct storage drivers to carry out data pre-fetch operations while transferring data from disc to I/O cache.
- Pre-fetched data could be useful to serve subsequent I/O request of an application and avoid/minimize physical I/O transfer operations.
- Applications can instruct file systems data access pattern on a specified file which in-turn is used by a file system as a parameter to configure pre-fetch limit (read ahead limit).

Sample code:

```
rev=posix_fadvise(fd, 0,20,POSIX_FADV_SEQUENTIAL);  /* Enable read ahead */

posix_fadvise (fd ,0,20,POSIX_FADV_RANDOM);  /* Disable read ahead */
```

Memory mapped file I/O (mmap)

- This method allows application to directly access buffer associated with a file without the need of system calls.
- Mmap invokes FS specific operations which appends a new page table entry, granting access to file buffer.

Sample code:

```
File data=(char *)mmap(void*)0, 100,PROT_READ |PROT_WRITE |MAP_PRIVATE ,fd,0);
```

- Application can indicate data access pattern on a memory mapped file using madvise.
- Madvise takes the address of IO cache as an argument along with access pattern constant.

```
madvise(temp,100,MADV_SEQUENTIAL);
```

Vectored I/O:

- This is widely used in user space drivers.

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

Ssize_t readv(int fd ,const struct iovec *iov ,int iovcnt);

Ssize_t writev(int fd ,const struct iovec *iov ,int iovcnt);
```

- The readv() function works just like read except that multiple buffers are filled.
- The writev() function works just like read except that multiple buffers are written out.

Multiplexed I/O:

- This mode helps an application to perform IO operations on a group of file descriptors based on readiness of the descriptor instead of sequence of the I/O calls.
- Linux provides three separate API 's for multiplexed I/O

A. Select:

```
#include <sys/select.h>
Int select (int nfds , fd _set*readfds ,fd_set*writefds, fd_set*exceptfds,struct timeval
*timeout);
```

Usage:

1) Identify total number of file descriptors to be monitored

```
int fd1,fd2;
fd1=open("./pone",O_RDWR);
fd2=open("./ptwo",O_RDWR);
```

2) Allocate file descriptor categories based on the event to be monitored.

```
fd_set read_set,write_set,ecept_set;
FD_ZERO(&read_set);
FD_SET(fd1,&read_set);
FD_SET(fd2,&read_set);
```

3) Initiate watch for the specific event using **select** API.

```
struct timeval timeout;
n=select (FD_SETSIZE, &read_set,NULL,NULL,&timeout);
```

4) Verify each fd and execute I/O operation

```
FD_ISSET (fd1, &read_set);

FD_ISSET (fd2, &read_set);
```

B. Poll:

```
#include<poll.h>

int poll (struct pollfd*fds, nfds_t nfds ,int timeout);

struct pollfd {

        int fd;

        short events;

        short revents;

};
```

- Both select and poll copy file descriptors to be monitored into kernel space and then setup watch.
- If an application intends to monitor fds using select and poll, using an infinite loop. Each iteration of the loop would begin with select or poll calls, which will copy the fd into kernel mode.
- Linux provides an alternate optimized I/O multiplexing call called **epoll**.

C. Epoll:

- Epoll interface provides two separate functions to copy file descriptor objects and watch them for a specific event.

Steps:

1) Allocate kernel buffer to copy epoll_event instances.

```
int epfd;

epfd =epoll_create (SIZE);
```

2) Allocate instance of epoll_event, initialize with fd details and add it to epoll_buffer.

```
struct epoll_event ev;
ev.evnts = EPOLLIN;
ev.data.fd = fd1;
epoll_ctl(epfd,EPOLL_CTL_ADD , fd1,&ev);
```

3) Set up watch

```
struct epoll_event evlist [COUNT]
ready=epoll_wait (epfd, evlist ,COUNT , MILLISECS);
```

4) Verify **fd** status and execute appropriate IO calls.

```
evlist[0].events  &EPOLLIN

n= read (evlist[0].data.fd ,buf,10);
```

Signals

Class Notes

Signals

Signals are asynchronous messages delivered to a process or group of processes from kernel's process manager. Linux supports 64 signals. UNIX and its variants provide two different categories of signals.

1) **General purpose** (Event notification) signals
2) **Real time** (Process communication) signals

- A general purpose signals are mapped to specific system events and they are identified with a specific name and name is assigned as per the event to which they are bound.
- Set of signals assigned to job control events are triggered when specific job control operations are initiated.

General purpose signals are of five categories:

- **Job control**: SIGCONT (18), SIGSTOP (19), SIGSTP (20).

- **Termination:** These signals are used to interrupt or terminate a running process.
  SIGINT (2), SIGQUIT (3), SIGABRT (6), SIGKILL (9), SIGTERM (15).

- **Async I/O**: These signals are generated when data is available on a specific device or when kernel services wish to notify applications about resource state.
  SIGURG (23), SIGIO (29), SIGPOLL (29).

- **Timer:** These signals are generated when application chooses timers alarms.
  SIGALRM (14), SIGPROF (27), SIGVTALRM (26).

- **Error reporting**: These signals occur when application code runs into an exception.
  SIGHUP (1), SIGILL (4), SIGTRAP (5), SIGBUS (7), SIGFPE (8), SIGPIPE (13), SIGSEGV (11), SIGXCPU (24).

Real time signals are exclusively meant for process communication. These signals do not have pre assigned names. They are priority ordered (33-64).

Signal flow:

- The **source** of a signal can be
    1) Process
    2) Exception handler (error reporting signals)
    3) Kernel services (drivers)
- The **destination** of the signal would be a process always.

- Signal generation and signal delivery may not take place in sequence. Signaling subsystem may delay the delivery of a signal when destination process is not ready to receive signals. The main reasons for the signal delay are
    1) Destination process is in wait state.
    2) Destination process has explicitly disabled signal delivery.

- An application can respond to a signal using either of the following methods.
    1) Execution of kernel defined signal handler (default).
    2) Execution of application defined (specific) signal handler.
    3) Ignore.

Register application specific signal handler:

```
#include<signal.h>

signal (SIGINT,sighand);
void sighand(int signum)
{
        <body>
}
```

Ignoring the signal:

```
#include<signal.h>

signal (SIGINT,SIG_IGN);    /* ignoring the signal */

signal (SIGINT,SIG_DFL);    /* switching  back to default */
```

Sigaction:

Posix standard provides a function **sigaction()** to register signal handlers or to change signal disposition.

```
NAME
       sigaction - examine and change a signal action

SYNOPSIS
       #include <signal.h>

       int sigaction(int signum, const struct sigaction *act,
                     struct sigaction *oldact);
```

```
The sigaction structure is defined as something like:

    struct sigaction {
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
    };
```

- Applications can choose to block signals from being delivered while handling priority signal handlers. Sigaction provides an option of blocking chosen signals while a critical handler is in execution.

  Sample code:

```
struct  sigaction act;
sigset_t sigmask;
int rc;
rc = sigemptyset (&sigmask);
rc=sigaddset (&sigmask ,SIGQUIT);
act.sa_handler = handler;
act.sa_mask = sigmask;
sigaction (SIGALRM ,&act,NULL);
```

- If a signal is generated while its delivery is blocked, it will be stored in the process signal pending queue. Pending signals are delivered immediately when the process unblocks signal delivery.
- Linux kernel allows applications to be in either of the following states of wait states.
    1) **INTERRUPTABLE_WAIT**: while a process is in this state, it can be interrupted by signals.
    2) **UNINTERRUPTABLE_WAIT**: while a process is in this state, all signals are blocked.
- Signal handlers allocate stack frames with in stack segment of the process address space. Applications can program a signal handler to use an alternate stack for its local data allocation.

Sample code**:**

Set up alternate stack:

```
stack_t sigstack;
sigstack.ss_sp = malloc(SIGSTKSZ);
sigstack.ss_size = SIGSTKSZ;
sigstack (&sigstack ,NULL);
```

Register signal handler using sigaction with flag SA_ONSTACK:

```
sa.sa_handler = sigsegvHandler;
sa.sa_flags = SA_ONSTACK;
sigaction (SIGSEGV ,&sa ,NULL);
```

Sigprocmask:

Applications can block delivery of signals during execution of main thread.

```
#include<signal.h>
int sigprocmask (int how, const sigset_t*set ,sigset_t*oldset);
```

Sample code:

```
sigset_t s_set;
sigemptyset (&s_set);
sigaddset (&s_set,4);
sigprocmask (SIG_BLOCK | SIG_SETMASK , &s_set ,NULL);
{
body;
}
sigprocmask (SIG_UNBLOCK, &s_set ,NULL);
```

- Sigprocmask when used with unblock command clears the specified signal from the block list and delivers them if they are found in the pending queue.
- Sigprocmask can be used to extract current blocked signals. Applications can append new signals to current blocked signal list.
- Applications can overwrite current blocked signal list by setting SIG_SETMASK flag as the first argument of the sigprocmask call.
- Applications can block or ignore or change signal disposition application defined handler for any signal except 9 and 19.

When signal handlers are in execution, application's primary thread is blocked. Application can resume executing only after termination of signal handler. When implementing application-defined-signal handlers the following issues are to be considered.

- Avoid longer work (instructions) in handlers.
- Avoid blocking function calls and operations.
- Avoid accessing global data
- Avoid calling async routines

Communication signals:

```
#include<signal.h>

int sigwaitinfo (const sigset_t*set , siginfo_t*info);

int sigtimedwait (const sigset_t*set , siginfo_t*info ,const struct timespec*timeout);
```

Sample code:

Step1: Block the signals which need to handle synchronously

```
sigset_t waitset;

sigemptyset (&waitset);

sigaddset (&waitset ,SIGALRM);

sigprocmask (SIG_BLOCK, &waitset ,NULL);
```

Step2: Wait for the chosen signal to occur

```
sigfinfo_t info;

sigwaitinfo (&waitset , &info);
```

Step3: Assign timeout. Sigtimedwait puts the application into wait until specific timeout.

```
struct timespec timeout;

timeout.tv_sec = 10;

timeout.tv_nsec = 1000;
```

signalfd() creates a file descriptor that can be used to accept signals targeted at the caller. This provides an alternate to the use of signal handler or sigwaitinfo and has got the advantage that the file descriptor can be monitored by select, poll and epoll.

Sample code:

```
sigset_ t waitset;

sigemptyset (&emptyset);

sigaddset (&waitset ,SIGALRM);

sigprocmask (SIG_BLOCK ,&waitset ,NULL);

struct  signalfd_siginfo info;

read (sfd, &info ,sizeof(signalfd_siginfo));
```

Timers

Class Notes

Timers

Timers are events delivered to a process at specific time intervals that allows a process to execute periodic or specific work on a time out.

Linux supports two interfaces:

- BSD timer API
- POSIX timer

BSD timer:

```
NAME
       getitimer, setitimer - get or set value of an interval timer

SYNOPSIS
       #include <sys/time.h>

       int getitimer(int which, struct itimerval *curr_value);
       int setitimer(int which, const struct itimerval *new_value,
                     struct itimerval *old_value);

DESCRIPTION
       The  system  provides  each  process  with  three interval timers, each
       decrementing in a distinct time domain.  When any timer expires, a sig-
       nal is sent to the process, and the timer (potentially) restarts.

       ITIMER_REAL    decrements in real time, and delivers SIGALRM upon expi-
                      ration.

       ITIMER_VIRTUAL decrements only  when  the  process  is  executing,  and
                      delivers SIGVTALRM upon expiration.

       ITIMER_PROF    decrements  both  when the process executes and when the
                      system is executing on behalf of the  process.   Coupled
                      with  ITIMER_VIRTUAL, this timer is usually used to pro-
                      file the time spent by the application in user and  ker-
                      nel space.  SIGPROF is delivered upon expiration.

       Timer values are defined by the following structures:

           struct itimerval {
               struct timeval it_interval; /* next value */
               struct timeval it_value;    /* current value */
           };

           struct timeval {
               long tv_sec;                /* seconds */
               long tv_usec;               /* microseconds */
           };

       The  function  getitimer() fills the structure pointed to by curr value
```

Sample code:

```
struct  itimerval  itv;
itv.it_value.tv_sec = 1;
 itv.it_value.tv_usec = 10;
 itv.it_interval.tv_sec = 1;
itv.it_interval.tv_usec = 0;
settimer (ITIMER_REAL ,& sa ,NULL);
```

Limitations:

- There is a possibility of missing timer events when application's periodic response takes undeterministic time. [This can fix by setting the handler SA_NODEFER flag]
- Applications designed to use multiple timers will have to be coded to verify timer elapsed before executing response [since all timers generate SIGALRM] .

POSIX timer:

Posix real time timers are optimized to suit the needs of huge applications which may use multiple timers. The following facilities are provided by Posix timer interface:

- Each timer registered by the application is identified using unique ID.
- Application can customize and choose signal to be delivered to notify expiration of timer (need not be SIGALRM).
- Applications can also program for a user level thread to be executed in response to a timer event.
- Applications can assign pre created user level threads in response to timer events.
- Posix interface supports nano seconds resolution.
- API is provided to find information of lost events.

```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *sevp,
    timer_t *timerid);
```

Link with -lrt.

timer_create() creates a new per-process interval timer. The ID of the new timer is returned in the buffer pointed to by timerid, which must be a non NULL pointer. This ID is unique within the process, until the timer is deleted. The new timer is initially disarmed.

The clockid argument specifies the clock that the new timer uses to measure time. It can be specified as one of the following values:

CLOCK_REALTIME
A settable system-wide real-time clock.

CLOCK_MONOTONIC
A nonsettable monotonically increasing clock that measures time from some unspecified point in the past that does not change after system startup.

CLOCK_PROCESS_CPUTIME_ID (since Linux 2.6.12)
A clock that measures (user and system) CPU time consumed by (all of the threads) in the calling process.

CLOCK_THREAD_CPUTIME_ID (since Linux 2.6.12)
A clock that measures (user and system) CPU time consumed by the calling thread.

Step 2 :

```
#include <time.h>

int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *new_value,
                  struct itimerspec * old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);

Link with -lrt.
```

```
struct timespec {
    time_t tv_sec;                /* Seconds */
    long   tv_nsec;               /* Nanoseconds */
};

struct itimerspec {
    struct timespec it_interval;  /* Timer interval */
    struct timespec it_value;     /* Initial expiration */
};
```

- Linux kernel provides a preparatory timer interface that is mapped to a file descriptor.
- Applications designed to wait for timers to elapse and execute response work in the main thread can choose to use this interface.

NAME

timerfd_create, timerfd_settime, timerfd_gettime - timers that notify via file descriptors

SYNOPSIS

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

int timerfd_create(int clockid, int flags);

int timerfd_settime(int fd, int flags,
                    const struct itimerspec *new_value,
                    struct itimerspec *old_value);

int timerfd_gettime(int fd, struct itimerspec *curr_value);
```

Fork

Class Notes

Fork ()

- Fork is a part of a library called UNIX threads library.

```
#include <unistd.h>
pid_t fork (void);
```

- Fork creates a new process by duplicating the calling process

```
Usage: main ()
        {
                pid_t  pid;
                pid = fork();
        }
```

- Code instructions coming after fork call shall execute both in the contexts of parent and child.
- On success of the fork call, PID of the child process is returned to the parent and 0 is returned to the child. On failure -1 is returned to the parent and no child process gets created.
- It is a common practice to follow up a fork call with a conditional construct. It could check for the return value of fork and execute appropriate work in the child process.

```
Example:
        pid_t  pid;
        pid = fork();
        if (pid==0)
                printf("code running in child");
        else
                printf("code running in parent");
```

Fork implementation under Linux

- Fork API invokes system call **sys_fork (),** which invokes the kernel routine **do_fork ()** to start a new process. It does the following functions.
    1) Allocates new PCB (task_struct) instance.
    2) Map caller task_struct elements to new task_struct (except PID).
    3) Return 0.
- Child process acquires instruction of caller process and starts executing from return statement of do_fork().
- Under Linux, fork () is implemented using Copy-On –Write (COW). So the only penalty it acquires is the time and memory required to duplicate the parent's page table and to create a unique task structure for the child.
- Process created using fork can continue to run even if parent process created them is terminated.
- When a process terminates, kernel's process manager puts it in to exit state and notifies immediately to parent. When parent process signals kernel, child process in exit state shall be removed.

- **ps** tool lists out the processes in exit state with a flag <defunct>. Defunct process can be cleaned up by either of the following **methods**.
    1) Write parent code to wait for the child to terminate and read exit value on child's termination.
    2) Register a signal handler in the process context to respond to a signal SIGCHILD.

- The first method can be implemented as below

```
        pid_t pid;
        int childstatus;
        int k=10;
        pid = fork();
// parent
        wait(&childstatus);
```

The parent will wait until the child gets terminated and exit status is returned to it.

- The second method can be implemented in two ways. In one way, the signal handler waits for the child status as shown below.

Sample code:

```
        int childstatus;

        void sighand (int signum) {
                printf(" I am in sig handler \n");
                wait(&childstatus);
        }

main(){
        pid_t pid;

        struct sigaction act;
        pid = fork();
        if( pid == CHILD){
                printf(" I am in child task \n");
                sleep(10);
        }
        // parent
                else{
                act.sa_flags = SA_NOCLDSTOP;
                act.sa_handler = sighand;
                        if( sigaction(SIGCHLD,&act,NULL) == -1);
                                perror("sigaction: ");
```

The flag SA_NOCLDSTOP is used to wait for the child status and it runs the signal handler.

- The second way is not to wait for the child status and runs the default handler. This can be done using a flag SA_NOCLDWAIT.

  Sample code:

```
main(){
        pid_t pid;

        struct sigaction act;
        pid = fork();
        if( pid == CHILD){
                printf(" I am in child task \n");
                sleep(10);
        }
        // parent
                else{
                act.sa_flags = SA_NOCLDWAIT ;
                act.sa_handler = SIG_DFL;
                if( sigaction(SIGCHLD,&act,NULL) == -1);
                        perror("sigaction: ");
```

- File descriptors are shared between parent and child processes as per Copy On Write basis.
- Any attempt to close an open file descriptor either by parent or by child process will trigger a copy operation.

```
main(){
        int fd;
        char buf[2];
       pid_t pid;
        int childstatus;
        fd = open("./test",O_RDONLY);
        pid = fork();
        if( pid == CHILD){
                read(fd,buf,2);
                printf(" in child %c\n",buf[0]);
                printf(" in child %c\n",buf[1]);
                close(fd);   // COW

                }
```

```
        // parent
        else{
                wait(&childstatus);
                        read(fd,buf,2);
                printf(" in parent %c\n",buf[0]);
                printf(" in parent %c\n",buf[1]);
                close(fd);
        }
}
```

- In the above program, a test file is opened and reads both in parent and child process. In the child process after read operation, the opened file gets closed. The output of the both the process will be same as the close operation in the child creates a Copy on Write.
- Any resource acquired by the parent process after fork operation will not be available to the child process created earlier.

POSIX Threads

Class Notes

Posix Threads (Pthreads)

- Pthreads are user level threading library for UNIX and its variants.
- Posix standard categorizes various threading APIs provided by the library implementation into two groups.

```
NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.
```

Thread attributes:

Each thread created using pthread_create has the following attributes.

1. **Detach state**: This attribute defines thread clean up mode on exit.

   Possible values

   1) PTHREAD_CREATE_JOINABLE: This creates a joinable thread which requires any thread in the current thread group to read the exit value for this thread to be destroyed. This is default.
   2) PTHREAD_CREATE_DETACHED: Start this thread in the auto clean up mode.

2. **Scope** : Defines thread visibility to kernel's process scheduler

   Possible values

   1) PTHREAD_SCOPE_SYSTEM: Thread is scheduled by kernel's process scheduler. This is default.
   2) PTHREAD_SCOPE_PROCESS: Thread is scheduled by a process level scheduler.

3. **Stack**: Defines the stack region to be used by current thread.
   Default: 2mb stack per thread. Each thread can be assigned from 16K to 8mb of stack size.

4. **Scheduling**: Defines scheduling policy and priority. Linux kernel supports the following policies by default.
   a) Non real time: SCHED_OTHER
      SCHED_BATCH
      SCHED_IDLE
   b) Real time: SCHED_FIFO
      SCHED_RR

- All the above scheduling policies are pre-empted (highest priority runs first)
- Non real time policies perform scheduling based on dynamic priority process of thread.
- Dynamic priorities are dynamically assigned to a process by the scheduler. This value is not fixed and bound to change each time the process is ready to run.
- Dynamic priorities are decided on the following parameters.
  a) Nice value of the process
  b) Amount of time the process spent in the ready queue.
  c) Type of process (IO bound ,CPU bound)
- Real time policies are based on static priorities assigned. Priority range is 1-99.
- Under real time ,SCHED_FIFO is a co-operative scheduler between same priority process and priority pre-emptive across other process groups
- SCHED _RR is a time scheduler across same process priorities and priority pre-emptive across other process.

- Sample code for thread attributes:

```
pthread_t thr;
pthread_attr_t attr;
int ret;

ret = pthread_attr_init(&attr);

 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

  pthread_create(&thr, &attr, threadFunc, (void *) 1);
```

- Detach state can be altered to joinable or to detached either during thread creation (using attribute instance) or at thread run time (directly making change in the thread object)

- Sample code for scheduling :

```
int inherit,policy,priority,rc;
pthread_t tcb;
pthread_attr_t attr;
struct sched_param param;

pthread_attr_init(&attr);

pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);

/* Now assign Sched policy and priority */

policy = SCHED_FIFO;
pthread_attr_setschedpolicy(&attr,policy);

param.sched_priority = 20;
pthread_attr_setschedparam(&attr,&param);

pthread_create(&tcb, &attr, t_routine, NULL);


param.sched_priority = 10;

policy = SCHED_RR;
pthread_setschedparam(tcb, policy, &param);
```

- Stack attribute is widely used to assign custom stack size based on thread's local data. Using default stack (2mb) may result in unused bytes and wastage of memory when threads do not contain huge local data structures.

Sample code:

1) Allocating memory (buffer) for stack

```
size_t stacksize =  16900;  /* in Bytes minimum 16 KB */
void *stackaddr;
int align = getpagesize();

posix_memalign(&stackaddr, align, stacksize);
```

2) Fill the attribute structure with stack details

```
pthread_t tcb;
pthread_attr_t attr;

pthread_attr_init(&attr);

pthread_attr_setstack(&attr, stackaddr, stacksize);

pthread_create(&tcb, &attr, t_routine, NULL);
```

LOCKS

Class Notes

Atomic operations:

Operations that read or change data in a single uninterrupted step are called atomic operations. Common atomic operations are:

- Test and set: Returns the current value of a memory location and replaces it with a given new value.
- Compare and swap (CAS): Compares the contents of a memory location with a given value and if found equal, it replaces with the given new value.
- Load link / store conditional instruction pair (LL/SC)
- Atomic arithmetic operations.

If two or more threads try to access a same shared data in parallel, race conditions will occur. Using atomic counter, software mutual exclusion locking protocols can be designed to protect large amounts of shared data from data races.

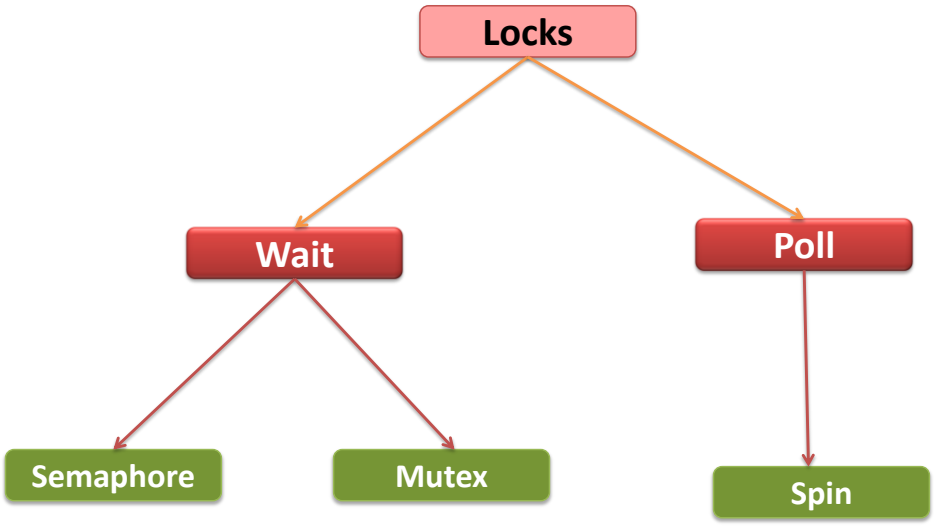
Locks

All thread libraries will provide locking and unlocking functions which can be used by the developers to synchronize threads. Locking and unlocking primitives are implemented using atomic instructions. Mainly there are two kinds of locking primitives provided by the thread libraries.

1) Wait locks
2) Poll locks.

- Wait locking interface push the caller process into wait state when lock acquisition fails. These are recommended when critical sessions are long and undeterministic.

- Poll locking interface put the caller process into a loop until lock is available. These are recommended when the critical sessions are short and deterministic.



Semaphores

Semaphores are shared counters that are incremented or decremented with atomic operations. Semaphores can be used in three different scenarios.

1) Data synchronization (mutual exclusion)
2) Resource counter
3) Wait / Event notification

Posix IPC semaphores

```
#include<semaphore.h>

sem_t *sem_open (char *name , int flag);

sem_t *sem_open (char *name , int flag, mode_t mode,  int value );
```

- **sem_wait** and **sem_post** are used to decrement and increment semaphore counter.
- **sem_wait** blocks the caller if semaphore value is 0, if it is 1 decrements to 0.
- There are three ways to get semaphore

```
int sem_trywait(sem_t*sem);

int sem_wait(sem_t*sem);

int sem_timedwait(sem_t*sem , struct timespec*abs_timeout);
```

- When semaphores are to be applied as a locking counter to enable synchronization of shared data, the following verifications must be supported
    1) Accidental release: lock and unlock operations must not be allowed to execute independently.
    2) Recursive dead lock: thread which has locked semaphore trying again to lock the same semaphore will result in recursive dead lock. This should be verified.
    3) Owner death dead lock: owner state must be trapped and any exceptions on the owner process must be reported to other processes waiting for the same semaphore.
    4) While destroying the semaphore its current state must be verified and must not be allowed to be released if it is in the active state.

Mutex

- Pthreads provide ready to use binary semaphore implementation with all required validations as a frame work called mutex.
- Pthreads mutex framework includes
    1) Mutex object (pthread_mutex_t)
    2) Mutex attribute object (pthread_mutexattr_t)
    3) A set of operations for locking, unlocking, setting and getting mutex attributes.

Sample code:

```
static int glob = 0;
int local;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mtx);
 local = glob;
 local++;
glob = local;
pthread_mutex_unlock(&mtx);
```

- Pthread mutex operation validations for recursive lock, accidental release, and owner death are disabled by default to allow lock and unlock operations to execute faster.
- Developers are advised to enable these validations by using an attribute object attached with mutex.
- Attribute instants can be disabled for production code and can be enabled for test code.

Pthread mutex attributes

Attribute object categorizes validation into following types

- Error check
- Recursive
- Robust
- Consistent

Error check mutex:

- This type enables recursive dead lock validation that returns an error when recursive lock attempt is detected.
- This type also verifies accidental release validations that do not allow a thread to unlock the mutex that is not owned by it.

Sample code:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_ERRORCHECK);
pthread_mutex_init(&mutex,&attr);
```

Recursive Mutex

This type allows a recursive lock to succeed provided it is unlocked same number of times. This type also adds validation for accidental release.

Sample code:

```
 pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&mutex,&attr);
```

Robust mutex:

This type of mutex adds validation for owner death dead locks. With the attribute set, a call to pthread_mutex_lock shall return EOWNER DEAD in the containing thread.

Sample code:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_setrobust_np(&attr,PTHREAD_MUTEX_ROBUST_NP);
pthread_mutex_init(&mutex,&attr);
```

Consistent mutex

- When mutex lock operation returns error code EOWNERDEAD, it is responsibility of the current thread to ensure that shared data and mutual lock set back to a valid consistent state.
- To arrays ownership records of the mutex lock and to be able to set it back to unlock state, the recovery code should call,

```
Pthread_mutex _consistent_np(&mutex);
```

Reader /writer locks (rw)

- Standard mutual exclusion protocols are not recommended on shared data that is frequently read and rarely updated.
- For reader intensive applications, rw mutual exclusion lock protocols should be used.
- rw locks allows sharing the lock in between multiple reader threads and mutual exclusion in between writer threads and rw threads.
- Pthreads provides implementation of rw locks using **pthread_rwlock_t**.

Spin locks

- Spinlock interface implements poll mode locking protocols.
- Pthread spinlocks can be used to protect shared data accessed by multiple process or a set of user level threads with in a process.

Sample code :

```
pthread_spinlock_t spin ;

pthread_spin_lock(&spin);
```

Design patterns

- Shared data synchronization using appropriate locking resources does have an impact on the overall performance of the application since all major locking protocols are implemented around mutual exclusion.

The following patterns have been observed across many applications that use various types of locking resources.

1) **Giant locking**: This pattern involves using a single lock counter to protect entire shared data.
2) **Coarse –grained locking**: This method of locking involves large shared data being logically divided into independent modules that can be parallely accessed and protected with local locks.
3) **Fine-grained locking**: This implementation involves identifying smallest possible units of data that can be parallely accessed and protecting them with individual locks.

Thread synchronization

When a group of threads need to achieve a common task by breaking down execution into multiple parallel threads, we need to synchronize their execution with appropriate methods.

Condition variables are one of the thread synchronization primitives provided by pthread library.

Condition variables

- They are used between producer consumer threads for unicast or broadcast event notifications.

Usage:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/*producer notification*/

pthread_cond_broadcast(&cond);
pthread_cond_signal(&cond);

/*consumer */

pthread_cond_wait(&cond, &mtx);
```

- Above function releases the mutex lock held by the consumer and puts consumer thread into wait until condition is signaled by the producer.
- When condition signal is notified, it acquires the mutex lock and resumes execution of consumer thread.

IPC

Class Notes

Inter Process Communication

- Linux kernel supports inter process communication mechanisms by providing three different kernel managed resources.
    1) Message queues
    2) Shared memory
    3) Semaphores
- BSD and POSIX standards have different specifications for allocations and management of the above resources. Linux kernel supports both the standards and implementations.
- Posix's implementation of IPC resources is file system oriented. System V and BSD implementations are specific system calls for allocation and access of these resources.

Message queues (msgq)

Message queues are kernel managed list of buffers that can be used to exchange messages between n processes.

Reasons for using msgq than files

- Persistent data files can be used as messaging resources between applications/processes. But persistent files are sourced from a storage device and read/write operations on them include disk I/O.
- Data return to persistent file is retained in the file even after it is read by the intended receiver.
- Special logical files called pipes can be used as a messaging resource but pipes do not provide message boundaries and are not recommended to be used in one-to-many communication scenarios.

System V message queues

- Each message is identified with message id.
- Reading process can read a message using message id.
- Message can be read in any order.
- Supports one-to-many communication

Posix message queues

- These are managed and implemented under **mqueue** file system.
- Applications are provided with file API's to initiate message queue operations.
- Each message must be assigned with priorities.
- Reader can read only highest priority message from queue.
- Reader can register for message arrival notifications.

- Allocating message queues

```
#define NAME "/my_mq"

mqd_t mqd;
 struct mq_attr attr;

 attr.mq_maxmsg = 50;
 attr.mq_msgsize = 2048;

 mqd = mq_open(NAME,O_RDWR | O_CREAT, 666, &attr);
```

  - To mount message queue

    mount –tmqueue none /media

  - To view message queue

    cd /media

    cat my_mq

- Writing messages on message queue

```
mqd_t mqd;
unsigned int prio = 50;

flags = O_WRONLY ;

 mqd = mq_open(NAME , flags);
mq_send(mqd,"writing message to queue", 25 , prio);
```

- Reading from message queue

```
struct mq_attr attr;
flags = O_RDONLY;
 prio = 50;
mq_getattr(mqd, &attr)
buffer = malloc(attr.mq_msgsize);
mq_receive(mqd, buffer, attr.mq_msgsize, &prio);
```

- Posix message queue interface provides message arrival notification facility. Receiver process can register for notifications to deliver on arrival of fresh message.

Sample code:

```
struct sigevent sev;
sev.sigev_notify = SIGEV_SIGNAL;
sev.sigev_signo = NOTIFY_SIG;

mq_notify(mqd, &sev);
```

- Notification type can also be set for a thread to run on arrival of thread.

Sample code:

```
static void notifySetup(mqd_t *mqdp);
struct sigevent sev;

 sev.sigev_notify = SIGEV_THREAD;        /* Notify via thread */
 sev.sigev_notify_function = threadFunc;
 sev.sigev_notify_attributes = NULL;
/* Could be pointer to pthread_attr_t structure */

 sev.sigev_value.sival_ptr = mqdp;

 mq_notify(*mqdp, &sev) ;
```

- Posix message queues can be deallocated using **mq_unlink().**

Shared memory

- Message queues cannot be used for sharing persistent data between concurrent processes or user level threads.
- Shared memory allows n number of processes to map and share a common buffer with shared data.
- Accessing data in shared memory does not involve user-kernel mode transition, since shared memory is directly mapped to the requesting process's address space.

Posix shared memory

- Posix IPC standard provides shared memory through a file system "shmfs".
- Each shared memory block is mapped to an inode of "shmfs".

Allocating shared memory

- Allocate a new shm_fs inode and assign buffer

```
fd=shm_open("/myshm" , O_CREAT | O_RDWR , 666);
ftruncate(fd , size);
```

- Map shared memory to the process address space

```
map(NULL , size ,PROT_READ|PROT_WRITE, MAP_SHARED, fd ,0);
```

- Shared memories are mounted automatically on **/dev/shm**.
- Process accessing shared memory need to use **mmap** with shared memory file descriptor.
- Process sharing common data using shared memory must ensure validity

IPC

Pipe is one of the message passing IPC resource widely used on *nix operating system platforms. Pipes provide unidirectional interprocess communication channel. A pipe has a read end and a write end. Data written to the write end of a pipe can be read from the read end of the pipe.

pipes can be created using *pipe* api, which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end. Pipes can be used to create a communication channel between related processes

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

The array fd[2] returns two file descriptors referring to two ends of the created pipe: fd[0] for reading and fd[1] for writing. Data written to the pipe is buffered by the kernel until it is read by the read end. The communication channel provided by a pipe is a byte stream: there is no concept of message boundaries. it is not possible to seek on a pipe(lseek).

In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096 bytes on IA32). Since Linux 2.6.11, the pipe capacity is 65536 bytes.

The following program creates a pipe, and then forks to create a child process; the child inherits pipe file descriptors that refer to the same pipe. After fork() each process closes the descriptors that it doesn't need for the pipe . The parent then waits on pipe1 for data and child process sends a string message , for which parent sends response message through pipe2, and the child reads this string a byte at a time from the pipe and echoes it on standard output.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

void client(int readfd, int writefd)
{
        char msg[100];
        int n;
        char eof = EOF;
        printf("%d enter msg to be sent to server: ", getpid());
        fgets(msg, 100, stdin);
        if (write(writefd, msg, strlen(msg) - 1) == -1){
                perror("error writing...");
                exit(0);
        }
        if ((n = read(readfd, msg, 100)) < 0) {
                perror("error reading...");
                exit(0);
        }
        msg[n] = '\0';
        printf("received from server: %s\n", msg);
}

void server(int readfd, int writefd)
```

```
{
        char msg[100];
        int n;
        if ((n = read(readfd, msg, 100)) < 0) {
                perror("error reading...");
        }
        msg[n] = '\0';
        printf("%d server received from client: %s\n", getpid(), msg);
        printf("enter msg to be sent to client: ");
        fgets(msg, 100, stdin);
        write(writefd, msg, strlen(msg) - 1);
}

int main()
{
        int pipe1fd[2], pipe2fd[2];
        int pid;

        /* create two pipes  */
        if (pipe(pipe1fd) == -1) {
                perror("pipe:");
                return 0;
        }
        if (pipe(pipe2fd) == -1) {
                perror("pipe:");
                return 0;
        }
   /* start child process and run client code in it */
        pid = fork();
        if (pid == 0) {
                close(pipe1fd[1]);      //close the write end of the first
pipe
                close(pipe2fd[0]);      //close the read end of the second
pipe
                client(pipe1fd[0], pipe2fd[1]);
                exit(0);
        }
        else {
                /* code that runs in parent ; runs as server */
                close(pipe1fd[0]); // close read end of the first pipe
                close(pipe2fd[1]); // close write end of the second pipe
                server(pipe2fd[0], pipe1fd[1]);
                wait(NULL);     //wait for child process to finish
                return 0;
        }
}
root@ubuntu:~# gcc pipe1.c
root@ubuntu:~# ./a.out
24978 enter msg to be sent to server: hello techveda
24977 server received from client: hello techveda
enter msg to be sent to client: hi! what have you learnt today ?
received from server: hi! what have you learnt today ?
root@ubuntu:~#
```

The program above creates two pipes and use them for two way communication between client and server

1. Create two pipes, pipe1 (pipe1fd[2]) and pipe2 (pipe2fd[2]).
2. call fork to create a child process and let child process run client code
3. In child process close write end of pipe1 (pipe1fd[1]) and read end of pipe2 (pipe2fd[0])
4. parent process close read end of pipe1 (pipe1fd[0]) and write end of pipe2 (pipe2fd[1])

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe will see end-of-file (read will return 0).If all file descriptors referring to the read end of a pipe have been closed, then a write will cause a SIGPIPE signal to be generated for the calling process.

popen and pclose

Linux provides two functions (popen and pclose) to create pipe from a process. It avoids the usage of pipe, fork, close, and wait.

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

popen() function starts a process by creating a pipe, forking and invoking a shell. The command argument accepts a shell command line, and the command is passed to /bin/sh using -c flag. The type argument expects either “r” or “w”. Note that since a pipe is unidirectional, we cannot pass both “r” and “w”, and the returned pipe stream is either read-only or write-only.

If the returned pipe stream is read-only, the calling process reads from the standard output. If the returned pipe stream is write-only, the calling process writes to the standard input. pclose() function waits for the associated process to terminate and returns the exit status of the command passed to popen().

Note that with popen() and pclose(), it is not convenient to create two pipes for two way communication. Below is a sample program using popen() and pclose(),

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void createTest() {
    FILE *f;
    f = fopen("test", "w");
    fprintf(f, "hello techveda\n");
    fclose(f);
}

int main(int argc, char **argv) {
    char buf[100], command[100];
    FILE *pf;
    createTest();
    sprintf(command, "cat test");
```

```
    pf = popen(command, "r");    //read using cat
    while (fgets(buf, 100, pf) != NULL) {
        printf("%s", buf);
    }
    pclose(pf);
    return 0;
}
root@ubuntu:~# gcc pipe2.c
root@ubuntu:~# ./a.out
hello techveda
root@ubuntu:~#
```

PIPE_BUF

Writing less than PIPE_BUF bytes is atomic. Writing more than PIPE_BUT bytes may not be atomic: the kernel may interleave the data with data written by other processes. For example, if two processes trying to write “aaaaaa” and “bbbbb” respectively to the same pipe. If the writes are atomic, the content is either “aaaaaabbbbb” or “bbbbbaaaaaa”. But if it’s not atomic, the content can be something like “aaabbaabbba”.

In Linux, PIPE_BUF macro is defined at limits.h, and the program below can print it out.

```
#include <stdio.h>
#include <limits.h>

int main() {
    printf("%d\n", PIPE_BUF);
    return 0;
}
root@ubuntu:~# gcc pipelim.c
root@ubuntu:~# ./a.out
4096
root@ubuntu:~#
```

Non Block Pipe

The default pipe blocks both reads and writes. A non blocking pipe can be created using fcntl() function with O_NONBLOCK flag.

FIFO's

Pipes can be used between processes that have common parent process. We refer to processes that have common parent process as related process. But for unrelated processes, pipe cannot be used, because one process has no way of referring to pipes have been created by another process.

When two unrelated processes share some information, an identifier must be associated with the shared information. Therefore, one process can create the IPC object and other processes can refer to the IPC object by the identifier. Linux provides named pipe (also called FIFO) to communication through pipe between two unrelated processes.

A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the file system. It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the file system. Thus, the FIFO special file has no contents on the file system; the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system.

FIFO's can be created using mkfifo function

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo creates a special file with name pathname. mode specifies the special FIFO file's permissions. It is modified by the process's umask: the created file will have the permission (mode & ~umask).

Once a FIFO is created, it can be operated like a usual file with the exception that both ends of FIFO need to be open first before reading and writing can begin. In other words, opening a file for reading blocks until another process open it for writing, and vice versa.

Following is an example shows how FIFO's can be used for communication

programs communicating using FIFO's need to agree on the FIFO names. It is a common practice to define them in a header file that can be included by both of the program sources.

```
root@ubuntu:~# vim scfifo.h

#ifndef TEST_FIFO_H
#define TEST_FIFO_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define CS_FIFO_NAME "./cs"
#define SC_FIFO_NAME "./sc"

#define FIFO_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

#endif
```

Next, the server program creates two pipes for communication and opens one for read and one for write,

```
root@ubuntu:~# vim server.c
```

```
#include "scfifo.h"

void createfifo() {
    int rv;
    if ((rv = mkfifo(CS_FIFO_NAME, FIFO_MODE)) == -1) {
        perror("error creating cs fifo: ");
        return;
    }
    if ((rv = mkfifo(SC_FIFO_NAME, FIFO_MODE)) == -1) {
        perror("error creating sc fifo: ");
        return;
    }
}

void server(int readfd, int writefd) {
    char msg[100];
    int n;
    if ((n = read(readfd, msg, 100)) < 0) {
        perror("error reading...");
    }
    msg[n] = '\0';
    printf("%d server received from client: %s\n", getpid(), msg);
    printf("server: enter msg to be sent to client: ");
    fgets(msg, 100, stdin);
    write(writefd, msg, strlen(msg)-1);
}

int main() {
    int readfd, writefd;
    createfifo();
    readfd = open(CS_FIFO_NAME, O_RDONLY, 0);
    writefd = open(SC_FIFO_NAME, O_WRONLY, 0);
    server(readfd, writefd);
    close(readfd);
    close(writefd);
    return 0;
}
```

The client program also opens the two fifos for write and read.

```
root@ubuntu:~# vim client.c
#include "scfifo.h"

void client(int readfd, int writefd) {
    char msg[100];
    int n;
    char eof = EOF;
    printf("%d client: enter msg to be sent to server: ", getpid());
    fgets(msg, 100, stdin);
    if (write(writefd, msg, strlen(msg)-1) == -1) {
        perror("error writing...");
        exit(0);
    }
    if((n = read(readfd, msg, 100)) < 0){
        perror("error reading...");
        exit(0);
```

```
    }
    msg[n] = '\0';
    printf("client received from server: %s\n", msg);
}

void removefifo() {
    unlink(SC_FIFO_NAME);
    unlink(CS_FIFO_NAME);
}

int main() {
    int readfd, writefd;

    writefd = open(CS_FIFO_NAME, O_WRONLY, 0);
    readfd = open(SC_FIFO_NAME, O_RDONLY, 0);
    client(readfd, writefd);
    close(readfd);
    close(writefd);
    removefifo();
    return 0;
}
```

Note that if we swap the order of the two lines in the client source code, a deadlock will occur.

This is because the server blocks at opening CS_FIFO_NAME for reading and waits for client to open it for writing, if the client opens SC_FIFO_NAME for reading first, it also blocks and waits for server open it for writing. Both programs stuck forever.

Under Linux, opening a FIFO for read and write will succeed both in blocking and nonblocking mode. POSIX leaves this behavior undefined. This can be used to open a FIFO for writing while there are no readers available. PIPE_BUF limits apply to FIFOs.

Process and Threads

Process & threads:

Process creation: Posix provides various APIs for process creation. Applications designed to execute concurrent tasks and applications designed for user interaction generally need such APIs

Concurrent program applications:

- Applications designed to execute parallel task are referred as concurrent programs.
- Such programs start with single thread of execution and have the ability to dynamically start a parallel context.
- Concurrent applications are designed to achieve either of the following objectives.
    1) Application of concurrency to achieve primary functionality of the software (multi threading).
    2) Application of concurrency to achieve faster execution and better utilization of CPU (parallel programming)

Case study to better explain concurrency (a game software)

Description:
The balloon shooter game, displays set of balloons on the console and the action key is configured to shoot the balloons, depending on the result of the action event the score is either incremented or decremented.
The following modules needs to be implemented to write this software

1) User Interface
2) Graphics
3) Action
4) Score

Windows version pseudo code

```
    int hit =0;
    int inc = 0;
    int  main()
{
    ui(();
    createthread(display);
    createthread(action);
    createthread(score);
}
```

- Create thread is a windows API that starts a parallel task with a function provided as argument.
- Task created using createthread() is called user level threads.
- Each user level thread contains a code segment of its own and shares rest of address space with parent (main thread).
- User level threads get scheduled (allocating CPU time) by user level scheduler. User level scheduler maintains a thread object to identify and store state of a user level thread.
- Kernel or its subsystem doesn't identify user level threads.

Unix version pseudo code

- Each task created by fork is a kernel supported thread.
- Kernel supported threads contains its own address space in user mode and PCB in kernel mode.
- Kernel and its subsystems identify kernel supported thread using PID and PCB.
- Kernel supported thread are scheduled and managed by scheduler.

Advantages of user level threading

- Better response time on average hardware

  User level threading is better because of lack of context switches.

- Developer friendly frame work (easier to write code, debug, maintain)
  1) All shared data can be accessed by user level thread implicitly by making it available in the data segment (as they have common PCB)
  2) Attaching the parent process to debugger allows accessing stack or data of any thread

Advantages of kernel level threading

- Kernel supported threads are more fault tolerant because the process which causes threat is halted while other process will run.
- Optimal usage of resources.

Fork()

- Fork creates a new process. It is done by duplicating the calling process.

- Since fork creates a child process as an exact duplicate of the calling process whatever instructions appear after fork call are executed twice, one in the parent and other in the child context. The order of execution is not guaranteed.
- On success, the PID of the child process is returned in the parent and 0 is returned in the child. On failure, -1 is returned in the parent and no child process is created.
- It is a common practice to follow fork call with a conditional construct. It helps assigning unique job to execute in parent and child.
- The stack and data segment is copied to child process.
- Child process created using fork can continue to run even after termination of parent process.
- When the immediate parent terminates init process (PID 1) takes over as parent.
- When a process terminates after executing instructions in the core segment, it is put into exit state. A process in exit state cannot content for CPU time (i.e. it can never become a context)
- Resources allocated by a process are not released /freed until PCB of the process is destroyed.
- It is always parent process's responsibility to ensure that the child process is destroyed on termination. To destroy terminated child processes the following methods can be applied.
    1) Synchronous clean up
    2) Asynchronous clean up
    3) Auto clean up.

1) Synchronous

- This method requires parent process to be suspended until termination of child process.
- When child is terminated instruct the kernel to destroy the child and resume execution.
- Wait call can also be used to collect the exit value of the child process. Exit value collected can be inspected to know the actual cause of termination using any of the following resources
    1) WIFEXITED (status) : returns true if the child terminated normally
    2) WEXITSTATUS(status): returns the exit status of the child
    3) WIFSIGNALED(status): returns true if the child process was terminated by a signal.
    4) WTERMSIG (status): returns the number of the signal that caused the child process to terminate.

2) Asynchronous

- This method requires parent to register a signal handler for SIGCHILD.
- Process manger delivers SIGCHILD to parent process whenever child process is terminated, stopped, continued.

3) Auto

- Terminated child process can be set to automatically destroyed mode without any further instruction from the parent.
- This requires the parent process to enable the default handler for SIGCHILD with a flag SA_NOCLDWAIT.
- Auto clean up is recommended to be applied only when child exit status is not important to parent.