shift + k at a function to go to the man page

->instructions in relocatable binary are assigned with offset addresses and a specific address isn't defined;
->instructions in executable binary are assigned more specific load address
->function call instructions in relocatable files refers to function offset address with respect to location of main
->function call instructions in executable files are refers to function load addresses
->runtime/boot strap routines are added to executables
->relocated binary contain translated binary instructions for the code find in source
->executable binary contain machine code equavalent of source with additional runtime(boot strap) instructions

Linker
    ->Instruction relocation
    ->Procedure relocation
    ->Appending bootstrap module
->Linker is specific to machine
->Compiler is archetecture specific
->Linker is of 2 types
    1.)Assigns physical addresses (firmwares)
    2.)Assigns logical addresses(PC's)

->Depending on the type of plotform and hardware linkers can assign physical address to program executable(functions,instructions)
->Linkers can also be configured to assign virtual addresses (hardware abstraction)

Binary File Formats

->fileformats specify how to organize data in files
->Relocatbles,executables have binary file format
->Linux has ELF format
->File formats are standars specifications,which describes organization of the filedata,and filelayout
->for every type of data there is a file format specification available (videos,docs....)
->when assembler and linker produced relocatable and exectable binary files,they would organize the file layout using binary file format standard
->binary file format describes layout of a binary image with in a file
->compile and build tools must be configured to use appropriate binary format which the runtime of thge application support
->windows uses coff file format standard
->File extention assigned to executable and relocatable files depend on specific file format used
->Windows system used coff binary format which requires all the object files to be created using coff
->As per coff standard relocatable files are identified with .obj extention and executable files are identified with .exe extentions

Program in memory

->binary file formats organized various elements of a program into different logical sections of the binary image
->for a instance all the code/text instructions are placed into a section called Code/Text
->global read,write data is placed into data section,read only data is placed into R/O data section
->When ELS data is presented to OS ,it opens up the file formats header to look inti details of code data,and R/O data sections
->kernel loader loads all these sections to appropriate memory areas ,based on policy of the OS
->memory regions allocated to a programe are refered are called program address space
->once address space is allocated program is registered with OS process manger which will identify a program and allocate an id
->program in memory is called process
->runs based on context switching

Exeutable section block diagram

PROCESS PROTECTION

->OS will have to ensure reliability of the entire system by isolating each app process address space from the other,this isolation would not affect kernel or application from the bugs and exceptions rised or occuring execution of a context
Protected Mode Software layout(Diagram)

Micro kernel layout(Diagram)

->Kernel space would contain all the kernel subsystems and it's services
->Among all kernel systems device driver subsystem undergoes frequent updates,to spport new class of devices ,these updates may contain bugs and inturn effect the kernel
->to isolate kernel services offered by 3rd parties from the core kernel ,another memory partition can be reserved into which 3rd party services can be deployed

Process Control Board(PCB)

->Kernel process managment registers a new process by allocationg an instance of a structure called PCB task_strtct
->This structure would be initialized with all the attributes of the process like identification details ,scheeduling,attributes,resource allocation,access controlled previllages

Process Runtime Code

->When a process is assigned CPU time slot address of stock segment is registered into CPU registers,stack pointer and instruction pointer is referenced to entry function of the program
->Entry function for every program is called _start ,it is a default function
->when the linker put runtimr code into executable file it initializes the address of entry function into header
->for all the apps build on linux plotform entry function is _start
->the _start function carrys the following actions

1.)argument stack frame allocation
2.)initializes command line arguments into argument stack frame
3.)initializes standard library context
4.)invocks entry function of the program functionality (main)

5.)invocks destructor to destroy the program(_fini) at the end
->Output ecexutable contains 2 tables
1.)section mata table
2.)Program meta table

Libraries
->Libraries are methods used to package reusable code,that can be linked into any program executable
->Libraries are generally available eiher in source format or relocatable binary format
->Linker deals with library linkage during build time
-> -static during compilation to build executable with static linkage
->linkers are confiured to resolve library linkage using either of the following methods
1.)Static linkage
2.)Dynamic linkage
->Whwn a library is linked statically the symbols of the library are directly appended into a executable image
->When a library is linked dynamically executable image is created with a reference for the library syabol
->Such executables link with a library when they are in memory

Creating Library Image

Static Library
Step1:Implement library source code
Step2:Compile source code and create relocatab;e binary
Stem3:Build static library image using ar tool
ar -rcs library.a one.o library.o

Some tools to examine static library files are
ar
nm
nm reads symbol tree of a library

Dynamic Library
Step1:implement library source code
Step2:compile code into posotion independent relocatable
gcc -c -fPIC abc.c xyz.c
Step3:Dynamic library must be created using plotform linker tool,Which ensures library created with ABI support
gcc -shared -o lib.so abc.o xyz.o

->executables that was build by linking statically contain call instructions referes to function directly
->incase of dynamic linkage call instruction is generated to refer to PLT record of the dynamically linked function
->procedure linkage table (PLT) is a additional section created by linker while building a dynamic executable ,this section contains information of all functions which are dynamically linked and whose address can only be known when program is in memory

linker is of two parts
1.)link loader
2.)link tool

->Static librarys are use and throw,where as dynamic librarys are like plugins
->Dynamic libraries are linked to executable files in two ways
1.)Runtime

2.)loadtime
->shared objects are by default loaded during app initialization time,such libraries are called loadtime libraries
->tool thar reads elf files is readelf
readelf -D app
->apps can use shared objects either as load time libraries or runtime libraries
->if a shared object(.so) is loaded and linked into program address space during initialization of a program ,it is reffered as load time library bindig
->load time libraries remain in process address space untill program termination
->if a app is programmed to load a shared object during execution it is referred to as runtime library ,such shared objects can be unloaded from the program address space when they are no longer needed

using a shared object as runtime library
->to use a shared object as a runtime library apps must be programmed to access all the library symbols indirectly through pointers

SampleCode:
Step1:Declare pointers of the appropriate type which can be used to hold the address of the symbols which are required to access
```
void(*funptr)(void)
```
Step2:request link loader to load specified library and return it's start address
```
void *handler;
handler=dlopen("abc.so",RTLD_NOW);
```
Step3:Look up into lobrary space for the address of required symbol
```
funptr=dlsym(handler,"test");
```
Step4:Access symbol through pointer
```
funptr();
```
Step5:Unload library when it is not needed
```
dlclose(handler);
```
Compile the code as following gcc abc.c -o app -ldl

->ldd app to know which type of library app contain
->files are of 2 types
1.)Logical (files in primary memory)
2.)Storage (files stored on a storage device)
->file systems are kernel services implemented to provide file managment operations
->As OS uses logical filesystems for managing files in memory
->Storage file systems to manage files on storage
->cat /proc/filesystems
->on Linux based OS list of file system services can be accessed using above command
->proc is a logical file system which is designed to show kernel data structur files
->ldconfig is a tool that is useful configuring dynamic linker runtime bindings
->ld.so.conf.d :lonkloader checks this path for .so files
->for that ldconfig converts .conf to binart and link loader uses this binary only
->dlopen requires a special flag as a second argument for dealing with unresolved symbols with in library
->RTLD_LAZY
->only resolve symbols as the code that references them is executed.if the symbol is never referenced then it is never resolved

->The following are complimentary flags which can be assigned along with NOW or LAZY

RTLD_GLOBAL:
the symbols referenced by the library with this flag willbe made available for symbols resolution of subsequent loaded libraries

RTLD_LOCAL:
this is default one ,it will not permit subsequent loaded libraries to use it's references

RTLD_NODELET:
do not unload the memory during dlclose

RTLD_NOLOAD:
do not load the library This can be used to test if the library is already resident (dlopen() returns NULL if it is not, or the library's handle if it is resident).

VIRTUALL ADDRESS SPACE

->on reset processors are initialized into real mode
->while operating on this mode CPU direactly address all memory (RAM,and device address space)
->any software written to run in this mode must be fully aware of address region which is available to use and must be built with direct addressing of data and code

->when OS kernels are initiated (boot time) CPU is putting to MMU mode (protected mode)
->while operating in this mode CPU no longer can address durectly and MMU takes over the responsibility of addressing memory
->CPU address space is still available for all the apps running on the processor ,this address space is refferred to virtual address space
->all reference from CPU point of view to any location must be translated to actual addresses of location and this operation must be carried out by MMU chip of processor
->Address translation carried out by MMU based on information provided by OS memory manager
->when MMU is initioated segmentation registers are programmed to treat CPU virtual map as privillage zones (address map available with CPU can be logically organized into seperate partitions for specific software we use)
->on linux 32 bit kernels for X86 archetecture by default MMU organizes 4 GB of address map into 2 zones
->Operating systems specify a standaed called application binary interface (ABI) which provides the layout of the address space of the app within user mode partition

Stack Analysis

->When a program loads into memory and registers with kernel process manager it is assigned a start address for stack segment (Upper most addredd of userspace)
->Stack segment is then used for storing local data of a function in execution
->for each function a stack frame is created where the local data is pushed
->As the function call change ,would expand stack segment would expand with one from on other and call change shrinks the stack segment shrinks
->on 32 bit system with 3GB of app address space ,limit for stack is 8mb

SYSTEM CALLS

->an OS is a plotform software which is responcible for initialization and managment of user apps
->an OS must ensure that an app in execution must have access to all required resources needed for it's successfull completion
->an OS software is generally build intrgrating 2 major modules
1.)user interface
2.)Kernel
->kernel module of a OS is implimented to handle all managment operations
1.)ap managment
2.)CPU
3.)Memory
4.)Network
5.)Device
->UI is implimented to present an interactive interface for a user ,in unix UI is called shell,in windoows it is called GUI
->kernel is devided into 2 layers
1.)Startup layer
2.)Service layer
->Startup layer:Loader,Kernel,Architecture specific code execution
->Kernel in built integrating startup code with service modules
->StartUp code is executed during OS initialization and is responcible for the following
1.)CPu initialization
2.)Processor
3.)Clock
4.)cache controller
5.)TLB initialization
6.)Chipset,Timer,Intrerrupt controller
->initialization of Device and Bus controller
->in kernel module of OS the following archetecture specific operations are done
1.)Setting up core data structurs
2.)interrept descriptor tyables
3.)Page table
4.)Device list
5.)File system status
6.)CPU scheduling tables
->initialization of service modules
->initialization of system loader

Archetecture Specific Code execution----------->Kernel------------>loader
->when applications are initiated and executed they require resources from kernel services through system calls
->system calls are functions in kernel space which serve as am interface for the kernel services
->applications are allowed to access any service only through relative system call

Invocking System Call
->system calls are in kernel address spsce apps cannot invoke them using function call syntax
->to invoke ststem calls apps have to use the following steps
1.)Move system call id into primary accumulator
2.)Starting with right most argument move arguments into available accualator
3.)Force the processor to move into kernel mode using a software interrupt (trap)
4.)read return value of system call from primaty accumulator

System call in App

```
int main(){
int res;
__asm__("movl $338,%eax");
__asm__("int $0x80");
__asm__("movl %eax,-4(%ebp)");
printf("value of res is :%d\n",res);
return 0;
}
```

system call and library call association architecture

application------->language library--------------->API--------------->system call----------->kernel service

application------>printf---------------->write----------->sys_write------------>driver

User to kernel mode Transaction

->Apps need to step into kernel mode for executing system calls and kernel services
->to push processor into kernel mode at runtime a soft interrupt instruction must be required
->for X86 archetecture processors INT is an machine instructions to rise a software interrupr
->this instruction must be invoked with a vector number argument to indiacte specific software interrupt
->for system call instruction ,trap (vector number 128) is designated for software interrupt
->During kernel boot interrupt vector table allocated with appropriate vector offset and handler address
->when trap interrupt is rised the following steps would executes

1.)Current context is preempted and it's user mode state is saved on to kernel stack
2.)Disable process scheduling at hard interrupt at present Core (CPU)
3.)passes control to interrupt hadler found at 0X80 vector
4.)looks up into eax accumalator for system call id requested
5.)finds the address of system call in system call table (system call id is used to identify offset of table where system call address is stored)
6.)Moves arguments found in accumalator into kernel stack
7.)Updates the instruction pointer to refer to baseaddress of system call in the PCB of preemptive process
8.)enable interrupts,schedulers
9.)when CPU slice is available to the application it would execute system call instruction
10.)on return from system call user mode context is restored from kernel stack

HEAP MANAGMENT

->Virtual address space of a process is maintained in memory descriptor of a process
->PCB process contains a reference to a memory descriptor
->start_brk refers to start address of heap segment and program_brk refers to end of allocated heap
->During start of the new program both start_brl and program_brk refers to same address
->increasing program_brk----------->allocating memory
->decreasing program_brk----------->deallocating memory
->brk and sbrk change location of program_brk

Sample code

```
void *cur_brk,*def_brk,new_brk;
cur_brk=sbrk(0);//get start of program_brk
brk(cur_brk+100);/malloc
brk(cur_brk);/free
```

->since heap allocation required program_brk to be incremented when a process allocates multiple blocks,they are stacked one on top of otjer
->deallocation/free of any blocks would only be possible in the reverse order,random deallocation leads to freeing deallocation of all blocks above the block being freed;
->avoid using brk(),sbrk() ,the malloc memory allocations package is portable and comfortable way of allocating memory

glibc malloc memory managment

malloc_stats,mallinfo

->the following sample code uses function calls of malloc package

```
void *p;
malloc_stats();
p=malloc(10);
malloc_stats();
free(p);
malloc_stats();
```

->for larger size allocation (>132k) request malloc falls back on mmap region of virtual address space
->allocating memory from mmap has significant advantages that the allocated memory blocks can always be independently released back to the system to contract the heap can be trimmed only if memory is freed at the top end
->malloc package can be configured with various allocation parameters using a function mallopt
M_MAP_MAX
->this parameter is used to specify maximim Number of allocations that may server using mmap:default 65 if it was zero we are forcing all allocations from heap
M_MAP_THRESHOLD
->this parameter specify the size of memory chunk to be considered as M_Map thresold.Any request equal to or greater than the thresold size that can't be satisfied from the free list(list) allocation is carried using M_MAP

Linux I/O Archetecture

->Linux I/O archetecture is desu=igned and implemented to faciliate common API for the applications to initiate I?O operations on various Resources

- ->Persistant files
- ->Logical files
  - ->pipes
  - ->sockets
  - ->message queues
  - ->devices

->a storage device when formatted the following blocls are created

1.)Boot block
2.)File System (FS) block
3.)data blocks

->boot block is used to store bootable image i.e OS loader
->FS blocks are for the use of filesystem service which are assigned for the disk
->Data blocks are for storing user data
->File system services store info about disk usage in one of the block called super block
->super blocks store the information about No of blocks and available sizes
->They store an entry describing each fil in a block called inode block
->each entry describing the file is called an inode .it is a structure defined by file system implimentation
->mount is an operation of copying inode info (File System)from Disk into Memory
->unmount is an operation of synchronizing memory image of inode with storage
->mount requires FS that can understand FS of media [compatable filesystem]
->mount -tntfs /dev/sbd   example of mounting

VIRTUAL FILE SYSTEM

->vfs is an abstraction layer that hides file system implimentation from user mode apps.application file API calls invokes the system calls of vfs which inturn switch appliacation requests into an appropriate file system
->mount really mounting to kernel file system cache (fat cache,NFC cache),vfs gets an image out of it
->open--->sys_open----->vsf inode-------->file ops
How vfs Resolves common Api call to a particular file-system
(Conceptual flow)

```
-----------------------------------------------------------------------------------
open()--> sys_open()---> fs_open()

int open(const char * path)
{
   step 1: validates physical presence of file
   step 2: invokes sys_open() call of vfs to process open request
   step 3: returns value that system call returns
}

int sys_open(const char * path)
{
   step 1: locate specified file inode in vfs tree(root file system)
   step 2: find file-system specific inode for the file (through
           vfs_inode fields)and invokes open operation bound to inode.

        fptr = vfs_inode -> fs_inode ->fops -> open()
```

```
        int a = fptr(); /* invoking file system's open call */
        if( a >= 0)
        {
            step 3: allocate instance of struct file
          step 4: initialize file object with attributes and address
                    of file system operations (fops)
          step 5: map address of file object to caller process
                    file descriptor table.
          step 6: return offset of the file descriptor table
        }
      else
         {
         step  7: return  a;
         }
}
```

Read/Write Operations

->applications Read/Write API calls invokes file system specific Read/Write calls through system call layer (vfs)
->File system R/W operations are configured to do the following actions

read---->sys_read------>fs_read

fs_read
1.)identify data region of file on disk (through inode)
2.)llok up iocache for requested data
      if(true)
      go to step (5)
3.)allocate buffer (new i/o cache block)
4.)instruct storage buffer driver to transfer file data to buffer
5.)transfer data to caller application buffer

fs_write()
1.)identify buffer of the specified file in the iocache
2.)update iocache
3.)schedule disk sync

IOCACHE(iocache)

->it is a list of buffers maintained by file system to store recently accessed file data

->allocation and deallocation of these buffers is carried out by file system

Limitation

->above method of dea;ing R/W operations on a file is called Standard I/O or Streaming I/O

->standard I/O is not suitable for applications dealing with critical storage data,since there is a possibility of disk sync to fail ,after writes are committed

->critical applications can ensure that the write operation of a file are synchronized to storage this can be achieved using an explict sync API along with standard I/O calls or changing the mode of filesyytem I/O into synchronized mode

->fsync,fdatasync synchronize a files incore state with storage device

->fsync flashes (Transfers) all modified in core data of file referred to by the fd to the disk device so that all changed information can be retrieved even after system crash or reboot.this includes writing through or flushing a disk cache if present,this call blocks untill the device reports that the transfer is complete

Method2

->enabling synchronozed I/O
->in this mode fs performs read write operations directly from storage onto application file buffer [I/O cache optimization is disabled]
->to enable this, app has to open the file descriptor with O_SYNC flag
->O_SYNC flag can also be enabled or disabled using fcntl API

Memory Map file I/O

->this method of i/o allows an app dirct access to i/o cache for a specified file
signature

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

->mmap creats a new file mapping in the virtual address space of the calling process

Argument1:void *addr ,start address of new mapping ,if addr is NULL then kernel chooses the start address
Argy=ument2:size_t length ,the length of file we want to map
Argument3:Access flags: for new memory map
Argument4:a special flags which indicates the scope of the mapping (shared mapping or private mapping)
Argument5:file descriptor (if we didn't specify file descriptor it will allocate a memory this type of allocation is used in malloc calls)
Argument6:start offset with in the file where the mapping should taken

usage

->file descriptor is used to find appropriate buffer in the cache
->mprotect is a API will be used to change the access permissions of an existing memory map
->mremap to be used ro resize an existing memory map
->MAP_ANNONYMOUS with -1 as file descriptor is used to select idle buffer
->strace will shows what API's a program is using

PROCESS MANAGMENT

->process managment subsystem is composed of the following modules
1.)process initialization(loader) and representation(PCB)
2.)process scheduler
3.)event/control managment

Process creation API
1.)These API's are used to initialize a new process
->process creation calls are required while implimenting any of the following
1.)App initialization software
e.g:shell,debugger,virtualization engine,profiler
2.)for concurant apps
eg:Browser,gaming
concurancy

concurancy can be achieved in 2 ways

1.)User level threading
2.)Kernel support threading

->user level threads have the following content
->Thread obj
->code
->Stack
These threads are implimented using pthread library
->kernerl supported threads are devided into two types
1.)Process
2.)Light weight process (LWP)
Process has the following content
->Address space
->PCB

Process creation calls are implimented using fork calls and these belongs to Unix family
Light Weight Process has the following content
->code
->stack
->PCB

Light weight processes are implimented using clone and these belongs to Linux family
concurant apps
->apps programmed with the ability to initialize dynamic execution context during runtime are referred as concurant apps
->concurancy can be achieved in 2 ways
->multithreading
->poarallel processing
->The differencve