User Space is very easy to learn, but hard to remember.
Let’s start easy remembering…

Memory controller
|
Address decoder (resolve address)
|
Address Map

Address space ← Bunch of addresses

Virtual Address space :-

Figure: Physical Memory access

1. Process performs access operations on memory through memory controller

2. CPU processor raises the address of the location to be accessed on the address bus and appropriate access of code on control bus.

3. Memory controller looks up the location requested by the processor, by resolving address through memory address decoder

4. Memory controller are pre configured with a physical address map that contains details of addresses assigned from each type of memory.

5. When a operating systems are loaded and initialized processor is put into protected mode by initializing MMU circuit

6. MMU abstracts physical address maps from software programs and it presence virtual address space (Virtual Address map)

→ Operating systems ABI (Application Binary Interface) Standard specifies virtual address split for use of kernel and applications
→ This is achieved by programming kernel bootstrap code to initialize MMU by configuring to range of addresses reserved for kernel (Kernel segment) and range of addresses are reserved for user applications
→ For 32 bit linux systems address space is divided into 3:1 ratio, (3GB address space for application user mode and 1GB for address space for kernel mode)
In 64 bit off/ off
→ Each user mode applications build by linked as per virtual address space layout.

Figure: virtual address space

Stack:

1) stack is a segment of virtual addresses used to allocate stack frames for the procedures in a execution

2) Each stack frame is identified with a base address and top address and it holds local data of the procedure.

```
Int main()
{
int a=10; |                                    pushl %ebp
int b=20; |--- Non executable statements        movl %esp,%ebp
int c;    |
c= a+b;
return c;

}
```

* Steps to translate source procedure into equivalent assembly
→ Identify non executable statements
→ Resolve all non executable statements using symbol table
→ Translate executable statements into assembly instructions

(ABI) Application binary interface will divide the data types, compiler nor top to bottom or bottom to top

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

c bp 100

b 88

a

sp

CPU register	Stack segment		push %ebp
		100	
edp			movl %esp, %ebp
esp		96	subl $12, %esp
eax		92	src , dest
edx	10	88	movl $10, -12(%ebp)
	10	84	movl -8(%ebp), %eax
			movl -12(%ecp), %edx
			addl %edx, %eax
			movl %eax, -4(%ebp)
			mov $0, %eax
			leave ret

Translating function call

```
int main()
{
int a,b;
a=10;
b=20;
swap(a,b);
return 0;
}
```

x-86 64 args passed through accumulator
32 std c caller stack frame

1) Function call translations depend on function call convention stds

2) Preferred function calling convention is specified for application binary interface at enforced by the compiler.

3) Linux 32-bit systems use c calling convention as a default standard.

4) As per c calling conventions (32bits arguments are always allocated in the caller function frame, called function would access arguments by moving out into caller frame.

5) For linux 64 bit systems c calling conventions passes the arguments through CPU accumulator registers

Steps to translate function call into equivalent assembly
→ Push each argument on top of the stack starting from right most
→ Invoke the function using call instruction
→ Read return value from eax
→ Release the space allocated for arguments

```
pushl -4(%edp)
pushl -8(%edp)
call swap
addl $8,%esp
movl $0, %eax
leave
ret
```

System Call

System call are interfaces to kernel services, system calls reside in kernel segment

→ since system calls reside kernel code segment taken are be both using standard function calling conventions
→ Application binary interface standard defines standard defines system call procedures

Invoking System Call

Kernel Mode bit =0 , User mode bit = 1

User process executing Call System Call Return from system call

User Space

Kernel Space

trap mode bit = 0 return mode bit = 1

\ /

→ Execute System Call →

→ Copy system caller ID to eax accumulator

→ Starting with right most argument more each parameter into each accumulator starting with ebx (max & accumulator in intel) initiate trap execepting using processor specific instruction

→ Read return value of system calls for eax

In 8086		mov $338 %eax
intel system call	←	int $ox80
		movl %eax, -4(%ebp)

```
#include<stdio.h>
int main()
{
int res;
res= syscall();

__asm__("movl $338, %eax");
__asm__("int $0x80");
__asm__("movl %eax, -4(%ebp)");

printf("res %d", res);

return 0;
}

int syscal()
{
        api(wrapper of system calls ); return res;
}
```

1) Apis are user mode functions which contains instructions to invoke a specific system call applications are programmed to invoke apis for requesting a specific kernel service linux syscall path.

Linux System call Path

Application	**Apps**	**Apps**	**ApI's**	
Library language (Most of These functions relay on syscalls)	**printf**	**malloc**		
API	**write**	**brk()**	**brk**	**User Space**
system call	**sys_write**	**sys_brk**	**sys_brk**	**Kernel Space**
Kernel Space System	**Driver**	**Buddy System**		**Buddy**

```
movl $0x88 %eax
int $0x80
mul %eax, -4(%ebp)
```

User Space

Kernel Space

→ Read eax
→ Look up syscall switch call
→ move argument to kernel stack
→ Update Sp register in stack pointer

Sys_newcal

Have some structure that having function pointers of addresses of syscalls ← Data segment
Code segment

Heap Management :-

1) Heap memory is a segment of virtual address space reserved for dynamic memory mapping (run time)

2) Heap allocations are used to store run time data

3) Memory descriptor structure of the process (nm_struct) contains pointers to start and end address of allocated heap
* Start_brk
* brk (Program break)

start_brk refers to start address of the heap and brk refers to current allocated top address of the heap

Allocating up words, de allocating downwards

int brk(void *ptr);

* brk and sbrk change the location of the program break

→ Program break is the BSS
→ increasing the program break has the effect of allocating memory to process decreasing program break is the de allocating memory

```
#include <unistd.h>
void *curr_brk, *def_brk, *new_brk;
int main()
{
	int ret;
	int *p;
	/* grab current program break address */
	curr_brk = sbrk(0);
	def_brk = curr_brk;

	printf("\n %p \n", curr_brk);
	getchar();

	/* change the location of the program break using brk */
	brk(curr_brk + 100);

	/* verify change */
	new_brk = sbrk(0);
	printf("\n %p \n", new_brk);
	getchar();

	/* restore the old location of the program break using brk */
	brk(def_brk);
	curr_brk= sbrk(0);
	printf("\n %p \n",curr_brk);
	getchar();
	return 0;
}
```

brk and sbrk are not preferred sense allocations are one and top of the other and can only be released in LIFO order. Abstract functions like malloc are preferred for allocation heap, sense the hide specific allocation in order the heap from application program

glibc (any C library)

Contains implementations of heap memory manager, which provides two group of function interfaces for operations of heap

→ Allocation and de allocations calls (malloc)
→ Configuration and tuning calls malloc-stats, mallinfo, mallopt

Policy allocations

* Default allocation policy glibc malloc

1) Default configuration of memory manager treat a request of less than 128k as small blocks and such blocks are allocated from heap segment

2) The memory used for these allocations are never released back to system such memory renames in the free poll of the allocated heap upon, free call of the application (LIFO)

3) A request of 128kb or larger is considered as huge block and these allocated from nmap segments of the process address space such block are released back to system when application invoke a free call

Allocating memory using nmap, has the significant advantage that the allocated memory blocks can always be independently released back to the system.

Mallopt :- Set allocation parameters

int mallopt(int param, int value);

Function adjust parameters that control the behavior of the memory allocation functions

M_MAP_MAX:- This parameters specifies the maximum number of allocation requests that may be simultaneously serviced using the nmap the default value is 65k. Setting this parameter to 0, disable the use of nmap for serving large allocation requests.

M_MAP_THRESHOULD:-

For allocations greater than or equal to be limit specified (for bytes) by M_ MAP_THRESHOULD, **that can't be satisfied from the free list** the memory allocation functions employ nmap instead of increases the program break using sbrk → Defailt is 128KB

DEFAULT_MMAP_THRESHOLD_MAX: 512*1024 on 32-bit systems or
4*1024*1024*sizeof(long) on 64-bit systems.

M_TOP_PAD

This parameter defines the amount of padding to employ when calling sbrk(2) to modify the program break.

Modifying M_TOP_PAD is a trade-off between increasing the number of system calls (when the parameter is set low) and wasting
unused memory at the top of the heap (when the parameter is set high).

M_TRIM_THRESHOLD

When the amount of contiguous free memory at the top of the heap grows sufficiently large, free(3) employs sbrk(2) to release
this memory back to the system. (This can be useful in programs that continue to execute for a long period after freeing a
significant amount of memory.) The M_TRIM_THRESHOLD parameter specifies the minimum size (in bytes) that this block of mem
ory must reach before sbrk(2) is used to trim the heap.

The default value for this parameter is 128*1024. Setting M_TRIM_THRESHOLD to -1 disables trimming completely.

Modifying M_TRIM_THRESHOLD is a trade-off between increasing the number of system calls (when the parameter is set low) and
wasting unused memory at the top of the heap (when the parameter is set high).

Mincore – Determine whether pages are resident in memory,

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

Natural alignment :-

An allocation is considered to be naturally aligned it, its start address is evenly divisible by size.

→ When accessing N bytes of memory the data memory address must be evenly disable by N, I.e, addr N==0.

→ un aligned memory access occurs when an attempt is made to read N bytes of data starting from an address that is not evenly divisible by N (I.e, addr/N != 0)

For instance, reading 4bytes of data address 0x10004 is file but reading 4bytes of data from 0x10005 would be and unaligned memory access

Effects of unaligned access:-

The effects of performing an unaligned memory access vary from architecture to architecture
- Some architectures are able to perform unaligned memory

Posix_memaligned() : Through which aligned buffers can be allocated dynamically

mlock(*addr len) mlockall(flags);

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.

Linux I/O architecture

Linux io architecture high designed with objective of providing with a common file apis through which they could initiate file operations of any io capable resource
Low level, High level erase blocks, FS blocks

BOOT Block | Super Block | inode Blocks

fdisk -l ← What are stroage devices are connected to system

mount -tntfs /dev/sda1 ← Media attach volume

umount /media ← Detach the volume

VFS (Virtual file system)

1) **VFS is a file system abstraction layer** which provides common file api interface for the application user mode.

2) When application initiate file operation VFS translates it to appropriate file system specific operations.

3) Applications can initiate common file api's called on any type of file irrespective of the file system which manages actual file.

*) The following are the benefits provided by VFS

1. Modifications and enhancements for existing file system will not break applications
2. New file systems can introduced without any modifications to user mode applications
3. Other kernel services like ipc, device drivers, etc.. can use virtual fs as the interface for communication with user mode applications.

Every file system as own cache in RAM

rootfs ← Constructive logical file system

Diagram

ls ← Read common list I.e, Vnode of common thing to all file systems

→ Fd is the application reference file systems

int vfs_open(const char *path);
{

Step 1: Locate specified file vnode in vfs

tree (root file system)

Step 2: find file system specific inode for the file (through vfs_node fields) and invokes open operation bound to inode
fptr = vnode → fs_node → fops → open()

int a = fptr(); // Invoking file systems open call

if (a == 0)
{
Step 3: Allocate instance of struct file

Step 4: Initialize object with attributes and address of file system operations (fops)

Step 5: Map address of file object to call process file descriptor table

Step 6: return offset number (file descriptor table) to which file descriptor structure is mapped

}

else return a;

}

open(const char *path, int flags, mode_t mode);
pts ← Terminal

* It opens a file descriptor which can be used subsequent calls
* File descriptor returned by successful all lowest numbered file not currently open for the process

read(int fd,void *buf, size_t count);

ssize_t ← Singed size size_t ← unsigned int

ssize_t write(int fd,void *buf, size_t count);

read → | sys_read --→ fs_read

fs_read()
{
Step 1: Identify data region of file on disk (inode)

Step 2: Lookup io Cache for requested data

if (true) Jmp steps

Step 3: Allocate buffer (new io cache block)

Step 4: Instruct storage driver to transfer file data to buffer

Step 5: Transfer data to caller application buffer (User space)

Step 6: Return number of bytes transferred user buffer
}

write | sys_write → fs_write

fs_write
{

Step 1: Check if request needs appending fresh data and make necessary changes to in_core inode (reserve new disk blocks)

Step 2: Identify buffer of the specified file in io cache and if needed allocate fresh buffers

Step 3: Update io cache with new data

Step 4: Schedule disk sync

Step 5: Return no of bytes transferred to cache (stdio) Synchronized io o_sync

}

Type 3: readv(int fd,iover *iov, int iovcnt);

read or write data into multiple buffers

Type 4: **Memory mapped file i/o**

* This mode allows an applications direct access to file buffer (io cache)
* This operation alters page table of the application

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

MAP_SHARED MAP_PRIVATE

mmap() creates a new mapping in the virtual address space of the call ing process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping

If addr is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new map ping.

How to map a file

filedata = (char * **)** mmap((void) 0, 100,PROT_READ_PROT_WRITE,MAP_SHARED,fd,0);

```
if ( filedata == NULL )
{
perror("mapping failed");
exit(1);
}
```

Anonymous mapping :-

filedata = (char * **)** mmap((void) 0, 100,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED,-1,0);

malloc are using

If you close file descriptor mmap not impact on file descriptor

msync(addr); // Synchronous back to mmap page table.
addr → Return from mmap

madvise:- System call advises the kernel about how to handle paging i/p o/p in the address range beginning at address addr and with size length bytes.
It allows an application to tell the kernel how it expects to use some mapped or shared memory areas, so that the kernel can choose appropriate read-head and caching techniques

MADV_RANDOM (MINIMAL) 1page

Expect page references in random order hence, read a head may be less useful then normally

MADV_SEQENTIAL (performance better)

Expect page references in sequential order (Hence pages in the give range can be aggressively read a head and may be freed soon after they are accessed

< 1GB

Direct io :- This method allow an applications to set up a buffer and configure it be used as a file cache.

Stroage driver is configured to fetch file data directly into buffer set up by application

direct_read.c fadvice

posix_fadvise (int fd, off_t offset, off_t len, int advise);

Program can be posix_fadvise to announce an intention to access file data in a specific pattern in the future, this following the kernel to perform appropriate optimizations

sequential ← Prefix maximize region

POSIX_FADV_SEQUENTIAL
the application expects to access the specified data sequentially (with lower offsets read before higher ones)

Process Management

1) Process initialization and representation
   - System loader (Process initialization)
   - Process control board (PCB) to store process information

2) Process CPU scheduler
   - Generic Scheduling algorithm
   - Scheduling priorities and priority queues
   - Scheduling Policy plugins for core algorithm

3) Job control or event management
   - Notification of events
   - Process triggered events
   - System triggered events
   - Providing default event handlers & event queues (process level)

Programming API's :-

1) Process creation and management api's
2) Scheduling api's
3) Job control or Signal api's

1) Process Creation Calls

Process creation calls are required while implementing any of the file of application.
* OS User interface (Shell)
* Debugger
* Virtualization (Virtual machine)
* Concurrent multi thread application

Multi thread or concurrent applications

1. Applications designed and implemented to spawn execution context dynamically referred to as concurrent or multi threaded

Concurrent applications

1) Event Driven
Maximum dependencies

2) Parallel processing
(Minimum dependencies) Pure concurrent

User Level threaded (User Mode)

Threading model

User level threading

1) As per this approach each threaded is a numerated as a sequence of instruction with in a process

2) **This threads are fully managed by a process level scheduler which resides as part of process code segment**

3) User level threads are not identified by kernel level scheduler and they can't content for CPU time at kernel level (system call)

Kernel Supported threads

1) As per this approach each thread is initialized as a process which is identified by kernel process scheduler

2) These threads are assigned a complete address space in a user mode and or represented by PCB in kernel mode

Diagram

User Thread	**Kernel thread**
Quicker response	Fault tolerant
Easy implement and maintain	Efficient in multi core processing
Latency go down (Just jump operations instead context switch)	Efficient use of hardware

Diagram

Unix threads fork api

→ fork() creates a new process by duplicating the calling process. The new process referred to as the child is an exact duplicate of calling process

→ Since code segment is duplicated instructions appearing after fork are executed in context of calling process and also in the context of child process

→ To branch out parent and child on unique code path a conditional construct is used

```
#define CHILD 0

child_pid = fork()

if ( childpid == CHILD )
{
work to be executed in child

}
else
{
work to be executed in parent
}
```

→ As per the fork, on success, the pid of the child process is returned in the parent and 0 is returned in the child

The condition is evaluated both child and parent process and it results in the calling context removed into appropriate code branch

```
fork() → sys_fork() → do_fork()
{
validate args
invoke do_fork()
return PID;
}
```

fork is an api which invokes appropriate kernel function responsible for duplicating the caller process

copy_process is the kernel function currently used for duplication

```
do_fork()
{
1. Allocate new address space
2. Copy segments of caller address space to new address space
3. Allocate new task_struct instance
4. Copy caller task_struct entries to new task_struct ( expect identification details )
5 return
}
```

Note: When operation in step 4 executes caller's CPU state is copied into child process PCB , resulting both processes to resume / start execution with same CPU state (ebp, eip, esp) when sys_fork returns in parent context it returns pid of the child and when it returns in child context gives back 0

Child start in kernel mode and returns to user mode

Termination of either parent or child process will not impact execution of other process reaping (reading the exit code)

parent running child exits (defunction)

→ On termination a process is moved into exit state

→ While a process is in a exit state its PCB is retained with all resource entries

→ Immediate parent will have to read exit code of the terminated child for the child process ti be fully destroyed

→ As long as immediate parent doe's not gather exit code of terminated child, it's continues to be exit state.

→ By reporting to parent we will know how child terminates (exit) that's why algorithm like parent should know exit status of child

→ Parent can be programmed to handle exit event of child using any of the following methods

1) Synchronous clean up

Suspend parent process execution until termination upon child's termination parent reaps child and continues

2) Asychronous clean up

- Process termination is notified to immediate parent by kernels process manager
- Program parent process to set up an event handler which is run in response to child's termination event.
- Implement event handler to reap child

3) Auto clean up

- Program parent to configure process manager to instantly destroy child process on termination
- This method does not allow parent to read exit status of terminated child

Method 1:-

```
pid  = fork();

if ( pid == 0 )
        code that runs in child
else
 wait(NULL); // synchronous destruction
        code that runs in parent

wait ( int * child status )
{
```

Step 1: Suspend caller until child terminates

Step 2: Gather exit value of child and return that as out parm (cause of child termination can be discovered through this value)

Step 3: instruct process manager to destroy defunct child

}

Method 2:-

Asynchronous

```
void handler( int signum)
{
wait(NULL);
}
int main()
{

pid  = fork();

if ( pid == 0 )
        code that runs in child
else
{
        signal(SIGCHLD, handler);
        code that runs in parent
}

}
```

(Cursor is not change) open close are change

Under linux, fork() is implemented using copy_on_write pages, so the only penalty that it incurs us the time and memory required to duplicate the parent's page tables and to create a unique task_struct for the child

vfork()

Current generation not using it old generation have it

→ vfork() differs from fork() in that the calling thread is suspended until the child terminates

→ Until that point the child shares all memory with it's parent including the stack

→ vfork() call differs only in the treatment of the virtual address space, as described above (but marked OBSOLETE)

Signals:-

1) Signals are asynchronous messages delivered by kernel process manager to a valid process or group of process

2) Each signal is represented by integer constants which is used to identify type of message

3) Linux kernel supports 64 signals grouped into two categories

* General Purpose signals (32 group) 1-31 signals
* Real time signals (32 group) 34 – 64

Signals

Event notification	Process communication
(General Purpose signals)	(Real time signals)

Diagram

→ When source entity is raise a signal it is refereed to as signal generation

→ When signal sub system makes receiver process aware of the signal by initializing it's response routine it is called signal delivery

→ **There is a possibility of delay in delivery of signal after it is generate**

→ A signal queued not delivery any of following true

1) Receiver process is found to be uninterruptible wait state

2) Receiver process found to be previous occurrence of same signal

3) Receiver has explicitly disable delivery of signal

Undelivered signals are queued in the receiver process PCB's pending signals list

1) When a signal is delivered appropriate signal dis-potion configured would be initialized

2) Applications can choose any of the following has default disposition at least one

*) Execution of kernel defined signal handler
*) Execution of application defined signal handler
*) Ignoring signal event

12 signal (cntrl + \ back slash)

signal(SIGINT,SIG_IGN);

signal(SIGINT,handler); // Changing one field

signal(SIGINT,SIG_DFL); // Default enabled

man sigaction (Changing entire node)

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

* This system call is used to change the action taken by a process on receipt of a specific signal

sigaction structure is defined as following

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

^c ^c ^c ^c

* sigaction provides various flags which the behavior of signal sub system

SA_NODEFER:-
Do not prevent the signal from being received from within its own signal handler

SA_RESETHAND
Restore the signal action to the default upon entry to the signal handler. This flag is meaningful only when establishing a signal handler

SA_NOCLDSTOP
If signum is SIGCHLD, do not receive notification when child processes stop (i.e., when they receive one of SIGSTOP,
SIGTSTP, SIGTTIN, or SIGTTOU) or resume (i.e., they receive SIGCONT) (see wait(2)). This flag is meaningful only when
establishing a handler for SIGCHLD.

SA_NOCLDWAIT (since Linux 2.6)
If signum is SIGCHLD, do not transform children into zombies when they terminate. See also waitpid(2). This flag is meaningful

```
struct sigaction act;
act.sa_flags = SA_NOCLDWAIT;
act.sa_handler=SIG_DFL;

if( sigaction(SIGCHILD,&act,NULL) == -1 ) )
        perror("sigaction");
```

SA_RESTART : - Restart interruptable signals
act.sa_flags = SA_RESTART

* system calls interrupted by the signal handler would be restarted
* This flag causes instruction pointer to jump to the location of the api call in the return path of the signal handler
* By default system calls interrupted by signal events do not resume on completion of signal handler
* Process scheduler suppose to types of wait state

1) Un interruptable wait – While in the state process PCB is in a wait queue and any signals generated will not be delivered

2) Interruptable wait – While in this state all generated signals for the process are delivered

→ Sigaction allows a process to mask (Block) choosen set of signals while executing specified signal handler

Ex:- strcut sigaction actl
sigset_t sigmask;

sigemptyset(&sigmask);
sigaddset(&sigmask,SIGQUIT);
sigaddset(&sigmask, SIGTERM);

act.sa_handler = hander;
act.sa_mask=sigmask;

<u>sigprocmask</u>

* A process is allowed to block signals from with in it's main thread this can be configured through an api sigprocmask

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

* Is used to fetch and / or change the signal mask of the calling thread

PCB having 3lists about signal

1) Blocking signals list (sigmask)
2) Handler signals
3) Queued signal list (Pending queue)

Process Space | PCB (Masking in)

SIG_BLOCK
The set of blocked signals is the union of the current set and the set argument.

SIG_UNBLOCK
The signals in set are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal
which is not blocked.

SIG_SETMASK
The set of blocked signals is set to the argument set.

All events of real time signals are queued and delivered to process

* Real time signals have higher precedence over general purpose

Pending real time signals are always delivered first

9,19 (Can't be blocked)

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

suspends execution of the calling thread until one of the signals in set is pending (If one of the signals in set is already pending for the calling thread, sigwaitinfo() will return immediately.)

removes the signal from the set of pending signals and returns the signal number as its function result.

Sigqueue – queue a signal and data to a process

int sigqueue(pid_t pid, int sig, const union sigval value);

The value argument is used to specify an accompanying item of data (either an integer or a pointer value) to be sent with the sig
nal, and has the following type:

```
union sigval {
   int   sival_int;
   void *sival_ptr;
};
```

POSIX Threads

Pthreads library organization

1. Thread creation and Management calls

    pthread_t, pthread_attr_t, pthread_once_t

2. Thread synchronization calls

    pthread_cond_t, pthread_barrier_t, pthread_sigmask

3. Shared data access sync calls
    pthread_mutex_t, rwlock_t, sem_t, pthread_spinlock_t

4. Timer events
    timer_t

5. Hardware access calls

Function naming conventions

1. Standard posix threads calls

libname_dataobject_operationname();
libname_operation();

Thread Creation and Management

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

The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

```
pthread_create(&t1,NULL,threadfunc,"Hello world\n");
```

```
pthread_exit(NULL); //blocking call
```

The pthread_exit() function terminates the calling thread and returns a value via retval

To allow other threads to continue execution, the main thread should terminate by calling pthread_exit()

Thread attributes :-

* A set of attributes can be assigned to a thread while creating it

The following are valid attributes

1. Detach state

These attributes specifies destruction mode of the thread possible values

* pthread_create_joinable

These will create a thread in synchronous mode (on termination thread object is retained until any other thread reaps exit value)

* pthread_create_detached
This value create a thread in detached mode thread is auto destructed on termination

2. Scope

These attribute specifies type of the thread to be created possible values

PTHREAD_SCOPE_SYSTEM

If this value is specified threads is as a light weight process

pthread_scope_process

These value causes a thread to be created a pure user level thread

These attributes not change on linux system

3. Inherits Schedulers

These attribute is to enable or disable inheritance of scheduling attributes from parent

pthread_inherit_sched (Default) - Scheduler inheritance is enabled

pthread_explicit_sched - Schedule inheritance disable

Scheduling Policy

Policy to be applied for current thread

1. Sched_OTHER:- Priority preempted fail share scheduler

2. Sched_FIFO:- Starting priority preemptive real time scheduler

3. Sched_RR:- Timer based scheduler

Scheduling priorities :-

Specifies priority of current threads (0-99) possible values

Default is zero

Stack Attributes :-

These attributes are used to a assigned stack and specified overflow guard size

Default stack is 2MB for thread

Maximum 8MB and minimum 16k

Linux Process Scheduler

1) Kernel process scheduler is composed of two layers

* Core Scheduler

* Scheduling Policy

Core scheduler carrying out all scheduling operations it's maintain 99 priorities queues for CPU and it is priority is preemptive

Scheduling policies are used by core scheduler to determine which among equal priority thread to run first and how long

Scheduling Policy layer contains implementation of two categories of policies

1. General Purpose Scheduling policies

SCHD_OTHER
SCHED_BACH
SCHED_IDLE

20

10

0

Category 2: Real time policy (1- 99)

SCHED_FIFO - First in first out

SCHED_RR - Round robin

General Purpose

SCHED_OTHER (Default)

Can be used at only static priority 0

The thread to run is choosen from the static priority 0 list. based on a dynamic priority that is determined only inside this list

The dynamic priority is based on the nice value and increased for each time quantum the thread is ready to run, but denied to run by the scheduler

SCHED_BATCH :-
→ Similar to sched_other with a difference that this policy is called cause scheduler (CPU intention)

→ This policy is useful for workloads that are non interactive

SCHED_IDL :-

This policy is intended for running jobs at extremely low priority

Group 2 :-

SCHED_FIFO :- First in first out scheduler

→ Can be used only with static priorities higher than 0

→ FIFO is simple scheduling algorithm without no time algorithm

→ A FIFO process or thread when preempted by scheduler will remain at the head of it's priority list

→ A FIFO process or thread preempted by an io call would enter it's priority list will enter the priority list from runnable again

→ A sched FIFO thread of process is puts to head of is list on valentary preemption

→ A call to sched_set scheduler will put the SCHED_FIFO thread identified by pid at the start of the list if it was runnable as a consequence it may preempt the currently running thread if it has the same priority

SCHED_RR:-

Round robin scheduling

Every thing described above for SCHD_FIFO also applies to SCHED_RR expect that each thread is allowed to run only for maximum time quantum

If a SCHED_RR threads has been running a time period

It will be put at end of list, list priority

Process

Struct sched_param param;

param sched_priority = 60;

sched_set_scheduler(0,SCHED_RR,&param);

SCHED_SET scheduler ((0, SCHD_RR| SCHED_RESET_ON_FORK, &param);

This will avoid copy priorities child copy, default to 0

/Linuxpro/sched

CPU set

sched_setaffinity(pid_t pid, size_t cpusetsize, const cpu_set_t *mask);

→ A threads CPU affinity mask determines the set of CPU's on which it is eligible to run

→ On a multiprocessor system, setting the cpu affinity mask can be used to obtain performance benefits

→ Restricting a thread to run a single CPU also avoids the performance cost caused by the cache invalidation that occurs when a thread cases to execute on the CPU and the recommency execution on different CPU.

cpu_set_t cset;

sysconf(_SC_NC_PROCESSORS_CONF);

CPU_ZERO(&cset);
CPU_SET(0,&cset);
CPU_SET(1,&cset);

sched_staffinity(0,sizeofset,&cset);

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

        pthread_attr_init(&attr);

        /* switch off sched inheritence from parent */
        pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

        /* Assign Sched policy and priority */
        policy = SCHED_FIFO;
```

```
pthread_attr_setschedpolicy(&attr, policy);

param.sched_priority = 10;
pthread_attr_setschedparam(&attr, &param);

/* create thread with choosen attribs */
pthread_create(&tcb, &attr, t_routine, NULL);

/* destroy attribute object */
pthread_attr_destroy(&attr);

sleep(2);

param.sched_priority = 20;
policy = SCHED_RR;
pthread_setschedparam(tcb, policy, &param);
```

The following routine is by two initialized with integer argument 1000

```
static void * thread_func (void *arg)
{
int local,j;
int loops=(int*) arg;

for(j=0;j<;loops ; j++)
{
local=glob;
local++;                                    access race instructions
glob=local;
}
return NULL;
}
```

To fix this we must do the load_add_structure in a single step

→ Both threads programmed to increment clk counter by fix number

Code segment one 1 & 2 in loop construct are possible data access race instructions concurrent execution of those instructions will corrupt state of clock counter.

→ To ensure that access to clock counter is dynamically synchronized between both the threads with atomic instructions must be applied while performed on glob

→ Each processor architecture supports atomic instructions for performing arthematic and bit wise operations

→ Atomic instructions execute specify operation in a single uninterrupted step with execute control signal asserted (even a hardware interrupt can’t preempt on atomic operation)

For synchronizing access to customize sized shared data mutual exclusion protocol can be implemented through an api

The following is a sample design of a simple mutual exclusion api

```
typedef struct __lock_t {
int flag;
}lock_t;

void lock_init(lock_t *lock)
{
lck→flag =0;
}
void lock(lock_t *lck)
{
while( lck→flag == 1 ); busy loop
lck→  flag = 1;

}

unlock(lock_t *lck);
{
lck→flag =0;

}
```

sample use code

Threads can be synchronized through lock unlock api's

```
typedef struct __shdata_t {
int a;
int b;
int c;
}shdata_t;

shdata_t glob;
lock_t glck;


static write(void *arg)
{
lock(&glck); //Exclusive lock
/*
code to modify shared data
 */
unlock(&glck);// Exclusion unlock

}

static read( void *arg)
```

```
{
lock(&glck); //Exclusive lock
/*
code to modify shared data
 */
unlock(&glck);// Exclusion unlock
}

main()
{
lock_init(&glck);
pthread_create(&t1,NULL,writer,NULL);

pthread_create(&t2,NULL,reader,NULL);

pthread_exit(NULL);

}
```

→ Lock functions current implementation of exclusion API programmed lock functions to initialize lock flag using regular CPU instructions which are not concurrent state

→ The race conditions on lock functions will successes resulting in all contented threads acquiring lock at same point in time

→ To ensure exclusion is strictly ensures lock functions must be programmed to use atomic instructions at serialize lock operations

```
void lock( lock_t *lck)
{
while(__sync__lock__test_and_set(&lck→ flag,1) == 1 );

}
```

Test and set enables you to test the old value (while is what is returned) while simultaneously setting the memory location to a new value

GNU_Compiler Macro;

```
__sync_lock_test_and_set(int *p, int value);

while(__sync_val_compare_and_swap(&lock_flag,0,1) == 1 )
```

Limitation:- Current implementation of lock function enforces exclusion through atomic operation but doesn't guarantee order of locking between contender

Api can be redesigned to enforce lock equalization order between contending threads

```
void lock(tlock_t *lck)
{
int myticket ;
```

```
myticket = __sync_fetch_and_add(&plck_seq,1);
while ( lck→lock != myticket )
sleep(10);

}
```

* Unlock will increment lock counter by 1 which causes appropriate ticket token

```
unlock(tlock_t *lck)
{
lck→ lock = lck→ lock + 1);

}
```

Exclusion 1 – Unlock 0- Lock

Poll Wait
(spin locks) (Mutex) Kernel provide atomic wait queues

Semaphores :-

1. Kernel / library managed atomic counters that can never be less than 0

2. Kernel / library provide atomic increment and decrement apis/ functions

3. Each semaphore is associated with a wait queue

4. When a process attempt to decrement semaphore while it is at 0

(Value sem == 0) process is put into wait queue

5. When a semaphore is incremented from 0 to 1, waiters in semaphore queues are flushed

```
typedef __lock_t {
sem_t sem;
} lock_t;

lock_init(lock_t *lck)
{
sem_init(&lck→sem,0,1);
}
```

```
void lock(lock_t *lck)
{
```

```
sem_wait( &lck→ sem); // Decrement
}
 void unlock(lock_t *lck)
{
sem_post(&lck→ sem); // increment
}
```

→ Wait locks are implemented using semaphore

→ Linux kernel implements the semaphore through a futex

→ Futexes are atomic counters in the user mode with a wait queue in the kernel mode (Unlike traditional semaphores which combine counter and wait queue an allocating kernel mode)

In it's bare form, afutex has semaphore semantics, it is a counter that can be incremented and decremented atomically, process can wait for the value to became positive

Futex operation is entry user space for non contended case. The
kernel is involved only to arbitate the contended case as any same design will strive for non contention, futexes are also optimized for this situation

5) Preferences

- Apply poll based synchronization when lock is held for fixed / deterministic duration
- Lock protecting atomic data units
-Fully atomic (Non blocking) critical code

- Apply wait based synchronization when lock is held for variable time
Lock protecting large data units with variable access time

- Critical code contains possible blocking calls

ISSUES :-

<u>Case 1: Accidental release</u>

Occurs when a thread attempts to release a lock which it does not own

Thread 1 (lock) Thread 2 (unlock) pthread_mutex_unlock

Case 2: Recursive dead lock

Occurs when a thread attempts to acquire a lock which already owns

Thr1

```
pthread_mutex_lock(&mtx);
func();
func1();

pthread_mutex_unlock(&mtx);
```

```
fun()
{
pthread_mutex_lock(&m);
}
```

Case 3: Owner death dead lock

Owner thread terminates without releasing lock

Thr1

```
pthread_mutex_lock(&mtx);
* Critical code *
ret = func()

if ( ret < 0)
pthread_exit();

pthread_mutex_unlock(&mtx);
```

Case 4: Circular Dead Lock:
Occurs due to incorrect ordering of nested locks

Thr1 Thr2

```
pthread_mutex_lock(&A);
pthread_mutex_lock(&B);

pthread_mutex_unlock(&B);
pthread_mutex_unlock(&A);
```

```
pthread_mutex_lock(&B);
pthread_mutex_lock(&A);

pthread_mutex_unlock(&A);
pthread_mutex_unlock(&B);
```

Case 5: recursive unlock:-
When an owner thread attempts to releases the lock recursively

Result:- Will trigger accidental release which results in violation of lock protocol

Thr1

```
pthread_mutex_lock(&mtx);
* Critical code *
ret = func();
```

```
pthread_mutex_unlock(&mtx);

pthread_mutex_unlock(&mtx);
```

Case 6: Release of a live lock

Occurs when dynamically where lock resides is released while lock was owned

```
typedef struct
{
pthread_mutex_t lock;
int a;
int b;
int c;
}shdata_t;
void *ptr;
main_thread()
{
ptr = ( shdata_t *) malloc ( sizeof(shdata_t));
}
```

thr1:

```
Pthread_mutex_lock(ptr→ lock);
ptr→ a = 10;
ptr→ b = 20;
ptr→ c = 20;

pthread_mutex_unclok(ptr→ lock );
```

Thr2

```
func(){
free(ptr);
}
```

Case 7: Re initialization of a live lock

Occurs when lock is initialized while it is owned. Results in failure of exclusion protocol

```
init_mutex(pthread_mutex_t *lock){
pthread_mutex_init(lock,NULL); }

thr_start()
{
init_mutex(&mylock);
pthread_mutex_lock(&mylock);

/* Same work */
pthread_mutex_unlock(&mylock);

}

int main()
{
pthread t1,t2,t3;
pthread_create(t1,NULL,thr_start,NULL);
```

```
pthread_create(t2,NULL,thr_start,NULL);
pthread_create(t3,NULL,thr_start,NULL);
}
```

Best Fixes

Case 1 Fix:-
Extended lock structure with owner field program lock() function to initialization owner field with caller PID / TID on successful acquisition of lock
Program unlock() function to validate caller PID / TID with owner and successfully unlock if caller is owner , and return failure if caller is not the owner

Case 2 Fix:- Extend lock routine to validate owner record before checking for availability of lock If caller PID / TID matches with that of current owner program the routine to return with failure status (-1).

Case 3 Best Fix:
Extend lock structure with an interval timer programmed lock() routine to start trigger timer at fixed intervals and unlock() function to disable timer have timer routine programmed to check owner status and wake up contending threads in case of owner death

Case 4 Fix : Ensure nested locking order is identical across all threads

Case 5:- Normal above approach

Case 6:- Ensure all shared memory allocations and de allocations are handling by main thread of the process

Case 7:- Fix program lock initialization routine to succeed only once subsequent calls to routine must fail to initialize lock

pthread library implements all of the above validation it's spin lock and mutex api's

→ To negate preference impact due to lock overheads (memory consume) libraries configured to turnoff all validation by default
→ It is a common practice to verify an application code lock validation on a test platform with library configured to enable all validation

Validations

PTHEAD_MUTEX_ERRORCHECK

-Recursive locking attempts return with failure

- Accidental release attempt shall fail

→ Recursive unlock attempt shall fail

PTHREAD_MUTEX_RECURSIVE

→ Recursive lock attempt is allowed to succeed (Provided owner must unlock same no of times as lock as held)

→ Accidental release attempt shall fail

→ Recursive unlock attempt is allowed to succeed on recursive lock

PTHREAD_MUTEX_ROBUST

- Check for owner death dead lock Examples/part2

```
pthread_mutexattr_t attr;

/* Set mutex type to ERROR check

pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_ERRORCHECK);

// Assign mutex attributes
```

```
pthread_mutex_init(&mutex, &attr);

pthread_mutexattr_init( &attr);

/* Enable owner death validation */

pthread_mutexattr_setrobust_np(&attr,PTHREAD_MUTEX_ROBUST_UP);

// Erase owner record from the mutex

pthread_mutex_consistant_np(&mutex);
```

Performance issues with locking :-

Lock usage designs

1) Gaint locking :-

When a single lock is applied to protect all of shared data which is accessed by all the threads of application (Gaint locking)

Example:
1) Database lock
2) Directory level locks
3) Kernel level lock protective all services

Pros:-

Easier code maintenance. Unlock debugging

Cons:-

1) Poor multi core performance
2) Highest lock condensation

2) Coarse-grained locking

This method required shared data to be organized into multiple logical modules which can be concurrently accessed at protective by a lock

Examples:-

→ Table level lock in data lock
→ File level lock in file systems
→ Kernel level service lock
→ Improve multi core performance

Limitation:
1. Increased lock over feed (memory, CPU)
2. Required active code maintenance lock level

Fine-grained locking

→ This process involves organizing shared data, it is as smallest possible units of data which can be concurrently accessed

→ Each data element unit must be protected with a lock

Pros:-

Maximum multi core performance

Cons:-

s1. Increased code maintenance
2. Complexity debugging
3. Lack of memory and CPU overhead

To verify if a specific application of lock an a shared data unit fine grained contention must be measured.