OS Topics:-

* Intro & Arch (Kernel, System Calls)
* Process Management
* Threads
* Scheduling
* Signals
* IPC
* Memory Management
* File System

Need for learning OS

Practical Examples / Assignments

Commands - Analysis

PBL / Micro Proj / Prob Statement

https://github.com/caia-techblr/linux-os-sys-prog/

FAQs

* What's on OS

* What are popular OS?

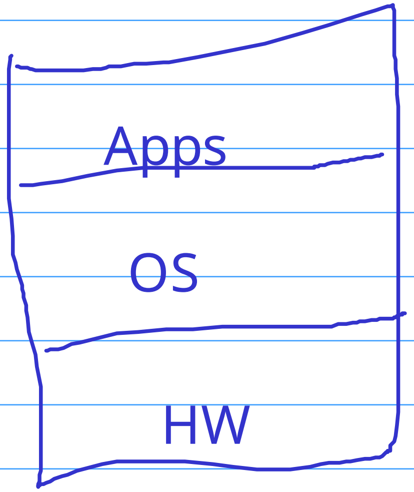
* Classification -- usage

* What are components of an OS?

- Kernel
- Device Drivers
- Libraries
- Utilities

Bridge/Glue between HW & Apps

Resource Manager



* User mode and Kernel mode

* User space and Kernel space

* Comp HW

* Normal Registers

* Program Counter/Instruction Pointer

* Flags/PSR --- mode bit(s) -- 1 or more

* Stack Pointer/Frame Pointer

Dual Mode

- Privileged/supervisor mode
- Normal/restricted mode

* HW Access

* Memory Access

* Specialized instr

```
printf --> write
scanf --> read

printf("Hello World\n");

char *buf="Hello World\n";
int len=12
write(1, buf, len);
```

stdin	(0)
stdout	(1)
stderr	(2)

a.out (hello.c)

↓

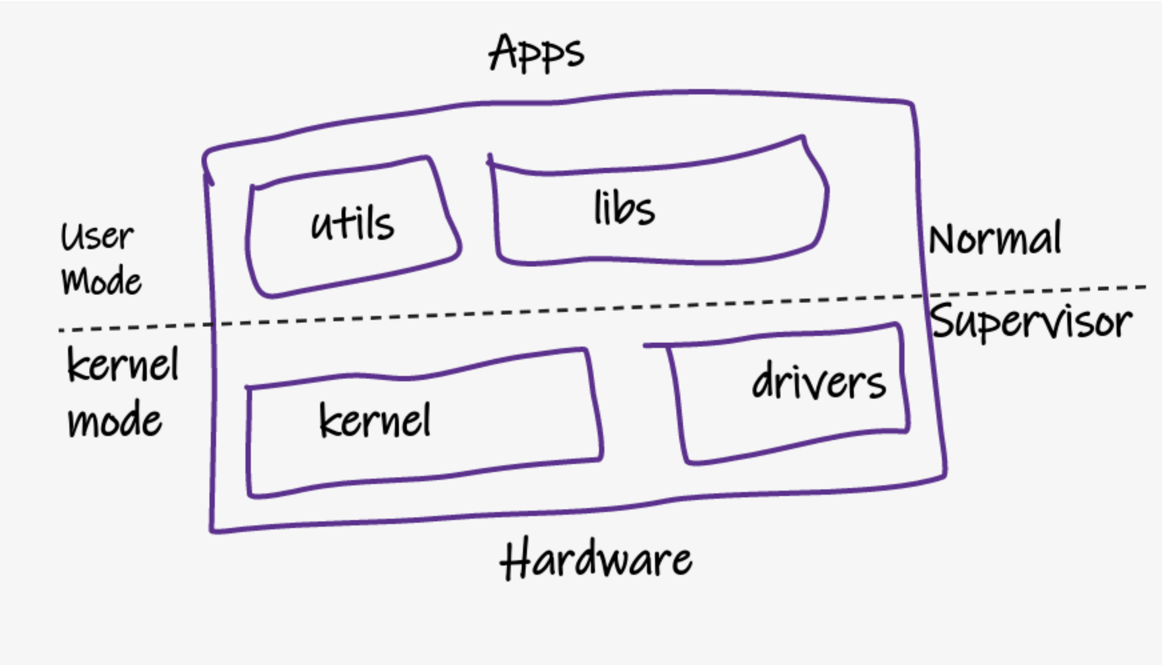
printf

↓

write

↓

HW



Trap Instruction

int 80h

sysenter

ARM:-

swi/svc

examples on file handing

- open,read,write,close,lseek

```
syscall(SYS_write, fd, buf, len);
write(fd, buf, len);
```

Assignment Ideas

- Copy files (cp file1 file2)
- Display file contents (cat file1)
- How to read large file

```
    file contents :- "ABCEDFGHIJKLMNOPQRSTUVWXYZ0123456789"
    char buf[11];
    int buflen=10;
    while(1)
    {
        nbytes = read(fd, buf, len);
        if(nbytes==0) break;
        //print buf
    }
```

```
sudo apt update
sudo apt install build-essential
```

```
gcc wrsample.c -o wrsample
./wrsample
gcc rdsample.c -o rdsample
./rdsample
----------
int main() {
  int i;
  for (i = 1; i <= 5; i++)
    printf("hello:%d\n", i);
  return 0;
}
```

```
gcc hello.c -o hello
strace ./hello
```

```
strace echo "Hello Linux"
```

Process

- program loaded in memory for execution
- program on disk (passive), process in memory(active)
- memory sections a process
    - stack
    - code
    - idata (.data)
    - udata (.bss)
    - heap
- process table / process list
- process control block / process descriptor
- process states, process life cycle
- context switching - context saving + context loading

process hierarchy : parent & child

PCB

-----

pid

ppid

name

context (??)

scheduling info

memory info

file info

cred (uid, gid)

exit status

Boot Sequence

```
ps
ps -el
ps aux
ps -e -o pid,ppid,stat,cmd

pstree -np
```

```
ret = waitpid(-1, &status, 0);
if(WIFEXITED(status))
    printf("normal,child exit status = %d\n", WEXITSTATUS(status));
else
    printf("abnormal termination\n");
```

Threads

- Concurrency
- Resource Sharing

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

void *task_body(void *pv) {
  char *plabel = pv;
  int i;
  printf("%s--welcome\n",plabel);
  for (i = 1; i <= 100; i++)
    printf("%s--%d\n", plabel, i);
  // pthread_exit(NULL);
}
int main() {
  pthread_t pt1, pt2, pt3; // thread handle
  pthread_create(&pt1, NULL, task_body, "A1");
  pthread_create(&pt2, NULL, task_body, "B2");
  pthread_create(&pt3, NULL, task_body, "C3");
  pthread_join(pt1,NULL);
  pthread_join(pt2,NULL);
  pthread_join(pt3,NULL);
  printf("main--thank you\n");
```

IPC
- Race conditions
- Critical Section
- Mutual Exclusion
- Semaphores, Mutex
- Deadlock

- Synchronization

Semaphores, Mutex

Semaphore
- integral value/counter
- Q to hold processes/threads

wait/lock/block
1A. if value>0, value--, go ahead
1B. if value==0, block and add to Q

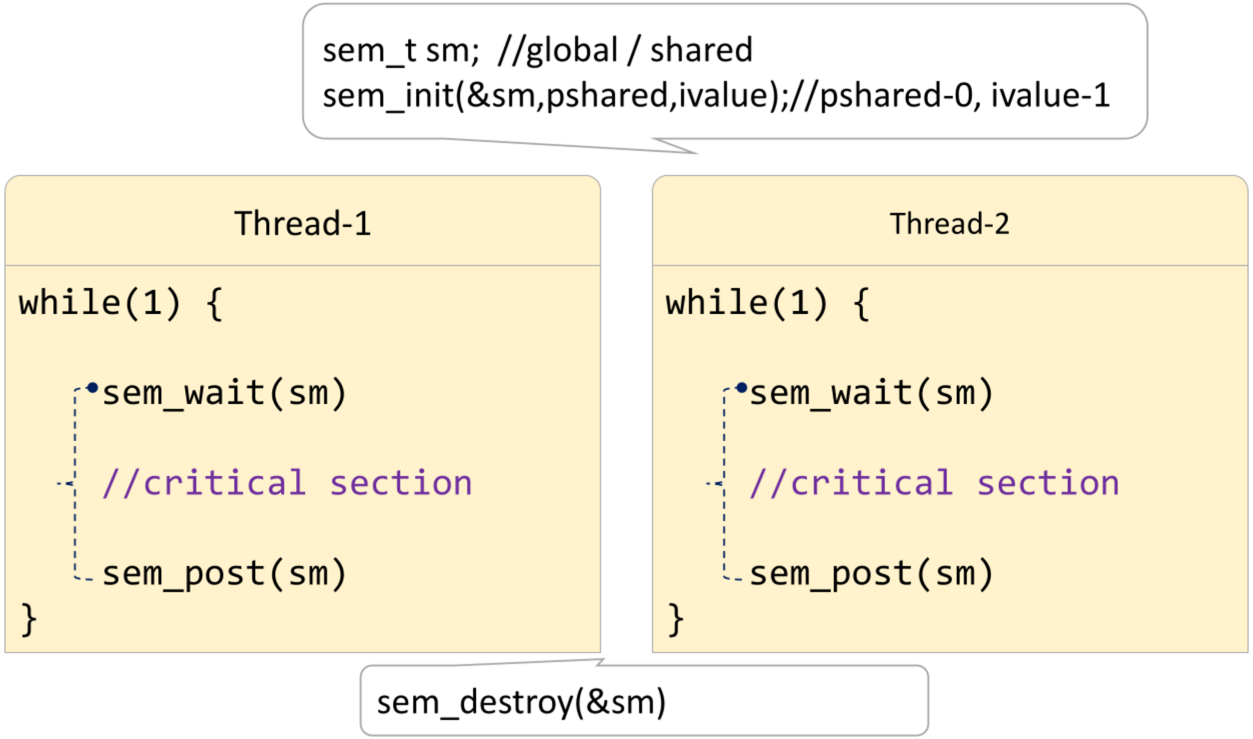
release/unlock/unblock
2A. if Q is not empty, remove one Q and resume
2B. if Q is empty, value++

Mutex API

```
pthread_mutex_t m1;   //global decl

pthread_mutex_init(&m1, NULL);       //before pthread_create
pthread_mutex_destroy(&m1);          //after join

pthread_mutex_lock(&m1);             // before val++ or val--
pthread_mutex_unlock(&m1);   // after val++ or val--
```



sem_t sm; //global / shared
sem_init(&sm,pshared,ivalue);//pshared-0, ivalue-1

Thread-1

```
while(1) {

   sem_wait(sm)

   //critical section

   sem_post(sm)
}
```

Thread-2

```
while(1) {

   sem_wait(sm)

   //critical section

   sem_post(sm)
}
```

sem_destroy(&sm)

Semaphores for synchronization (Dependency/sequencing)

//create a semaphore, initial val = 0

```
Prod                        Cons


                             sem_wait(&s1);
//add                       //remove
sem_post(&s1);
```

Mutex is not just Binary Semaphore

Mutex Features

- Strictly for mutual exclusion
- Ownership applies, whichever thread locks mutex, same can unlock
- Unlocking before locking/ unlocking more than once allowed

- In RTOS, Mutex provides solutions to Priority Inversion problem
  (e.g. priority inheritance/ceiling solutions)

Producer Consumer Prob

- There is a common data source, accessible by multiple threads
- Threads known as producers add data
- Thread known as consumers remove data

R1 - If buffer is empty, consumer should block, only producer is allowed
R2 - If buffer is full, producer should block, only consumer is allowed
R3 - If both are allowed (partially filled), any one can access at a time

Prod

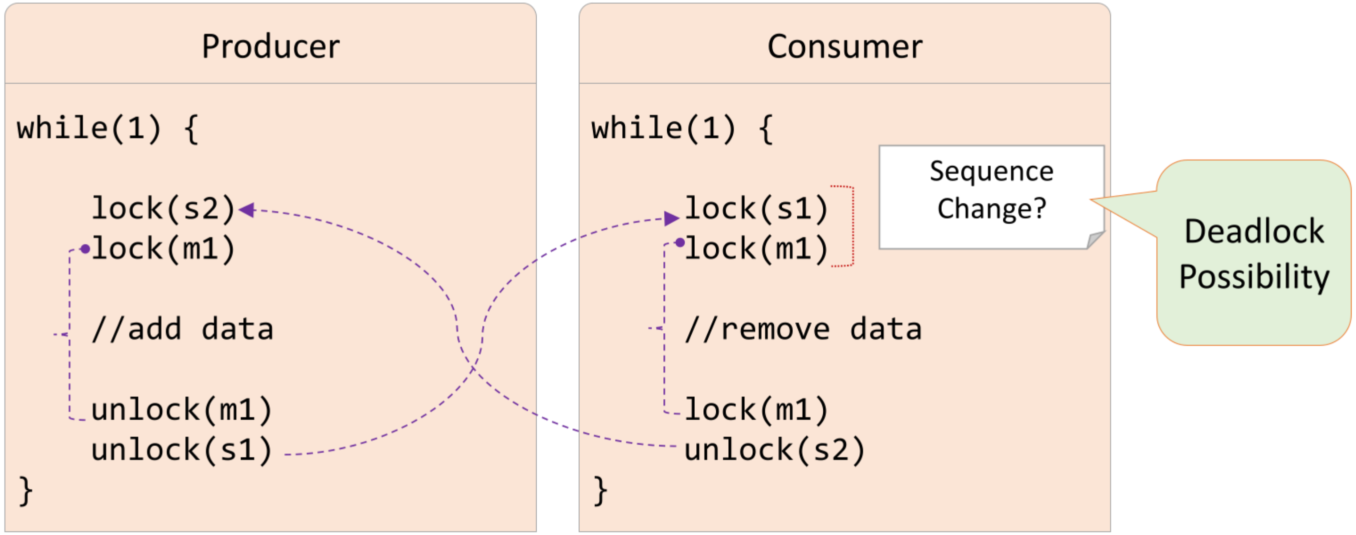
sem_wait(&s2);

//add

Cons

//remove

sem_post(&s2);



Producer

```
while(1) {

    lock(s2)
    lock(m1)

    //add data

    unlock(m1)
    unlock(s1)
}
```

Consumer

```
while(1) {

    lock(s1)
    lock(m1)

    //remove data

    lock(m1)
    unlock(s2)
}
```

Sequence Change?

Deadlock Possibility

lock(m1)

lock(s1)

```
Mutex pm;        //Semaphore pm, ival=1;   -- Printer
Mutex sm;        //Semaphore sm, ival=1;   -- Scanner

A1               A2

lock(sm)         lock(pm)
lock(pm)         lock(sm)

//copy           //copy

unlock(pm)       unlock(pm)
unlock(sm)       unlock(sm)
```

Prevention

- Atomic locking / acquire all resources in one go
- Try to lock resources in particular order
- Apply mutual exclusion after resolving dependencies

0) Examples, commands, techniques

1)Design your own shell (mini shell / tiny shell)

==> Display some string as prompt, e.g. "myshell>"
==> Take command name as user input
==> Create a child process using fork
==> Launch requested requested using execl/execlp inside child
==> Parent(shell) wait for child (command exec) using waitpid and print exit status
==> Repeat above steps, until user input is "quit"

2) Write a multithreaded application to perform parallel sum of large array

3) Modify/enhance TCP server code, to communicate to multiple clients
using threads (concurrent server)

4) Producer-Consumer problem using threads practically

5) Implement own commands like cp, cat (open,read,write,close)