Operating System Overview

Definition: **Operating System**

A *software system* (collection of related programs) whose objectives are

1 - to manage the *physical resources of a computer* (memory, files, processors, devices)

- *Safely* (no *trashing* of files / *crashing* of system / hacker *intrusions*)
- *Efficiently* (O/S doesn't consume excessive system resources)
- *Fairly* (resources are consumed in accordance with management objectives)

2 - provide applications a programming interface *(API)* that

- imposes useful abstractions on raw hardware creating virtual resources

  files on storage devices
  address spaces on physical memory
  network devices

- simplifies management of *serially reusable resources*
- supports communication and synchronization among *cooperating processes* (two or more processes working cooperatively to solve a particular problem)

Required hardware capabilities for *safety* and *efficiency*

- *Two state processor(s)* : (user state / supervisor state)
  - *User state:* Dangerous instructions (change processor state / initiate I/O operation) disabled..
  - *Supervisor state:* All instructions work
  - Application processes run in *user state* and the runs O/S in *supervisor state*...
  - So how does the switch between states happen
- *Interrupt mechanism*
  - *Software interrupt:* Used by application to trigger switch to supervisor state to request OS service (Read from file, allocate memory)
  - *Hardware interrupt:* Used by devices to indicate device needs OS service
- *Memory protection*
  - Protect applications from each another
  - Protect OS from applications
  - Typically done by *mapped (virtual) memory*

Operating System functionality – *depends on application requirements*

General purpose Operating Systems: *Z-OS, OS-X, Windows, Linux/Unix*

- Support for *userids* and login (password or key) *authentication*.
- Create new *processes* (an instance of an executing application program)
- Load application programs
-
- Provide a memory management scheme in which each process is assigned a protected private region of memory called an *address space*
- Support *light weight processes* or *threads* that coexist in a single shared address space
- Impose a filesystem abstraction on raw storage capability of disks.
- Associate file ownership with *userids* and provide file protections.
- Provide a set of I/O device drivers serving each type of device on the system.
- Provide a scheduling system that starts a thread running on a processor and preempts it after it has consumed its time quantum unless the thread voluntarily suspends execution.

Special purpose (embedded) Operating Systems: WiFi AP, Network router, Car ECM, uWave oven, Cell Phone, Aircraft control (MANY systems), Car driver assist, Car autodrive, Medical devices, Guided missile, Robots, etc..

- Functionality is *highly* dependent on the specific execution environment but is a subset of the above capabilities.
- Thousands of *distinct* systems are in widespread use
- Implementation strategies include a combination of
  - Written from scratch (e.g. MicroCIM at Clemson)
  - Tiny OS (Open source sensor networks)
  - Cotswold OS (IBM)
  - Derived from a general purpose (Linux) OS (e.g. Android)
  - Derived from a previous version but with extended functionality (Aircraft, Automotive, Locomotive control)

Failures in embedded medical, transportation, and military Operating Systems and their applications are human life critical!

- Therac 25
- Airbus 330 Q 72 / AF 447
- Boeing 737 Max

Functional components of an OS

Process (independent instance of a program in execution) Management

- High level scheduling (Occurs only twice in the life of a process)
  - Create/Destroy processes

- Low level scheduling (Occurs thousands of times per *second.*

  - Cause a process to start execution on a processor (dispatching)
  - Provide for communication and synchronization between processes.

Memory Management

- High Level
  - Create / destroy process address spaces

- Low Level
  - sub- allocations from within an address space.

Resource Management

- Control access to serially reusable resources; eg, printer, file, directory, memory resident data structures while simultaneously preventing/avoiding/detecting *deadlocks and livelocks*

File Management

- Manage both free and allocated disk space.
- Create/Delete files.
- Directory management services
- Map logical file addresses to physical disk addresses.

I/O Device Management

- OS must initiate all I/O on behalf of apps and itself
- Respond to interrupts that signal I/O completion.

In summary –

This particular classification serves as a reasonable outline structure of an implementation agnostic OS course.

Why is OS development challenging?

- Every model of I/O device controller (disk/video/network..) has …
  - a dozen+ programmable registers
  - Every register has
    - a dozen+ programmable bits
- There are 1000 plus I/O controllers supported by linux
- But the *main* challenge is *ensuring correct execution* when cooperating threads are *accessing shared data*
- Single thread applications always produce the same result for a given input
  - Interactive debuggers (gdb) can detect all errors for a given input (but not *all inputs*)
  - Multiple thread apps can fail every time on a given input *or* they can run correctly 999 out of 1000 times.

Classical problems in process synchronization

Read/modify/write

- 8 threads run concurrently and increment the global variable *shared*
- The value of *loopcount* is 1,000,000
- The final value of shared should be 8,000,000

```
43 /* We will run instances of this function as separate threads */
44 /* Each iteration (attempts) to increment "shared" by 1       */
45
46 void xthread(
47 int  id)       /* Logical thread id.. values are 0, 1, 2,... */
48 {
49    int i;
50    int local;
51 // printf("Made it in %d \n", id);
52
53    for (i = 0; i < loopcount; i++)
54    {
55    // get_lock(id);
56       shared++;
57    // free_lock(id);
58    }
59    pthread_exit((void *)0);
60 }
```

```
Value of shared is 170991
Value of shared is 175724
Value of shared is 171583
```

The solution is called mutual exclusion

- *mutual exclusion* is a mechanism that can ensure that only a single thread is executing in
- a *critical section* where destructive updates of a shared variable can occur.

Objective

proc 1	proc 2
while(1)	while(1)
{	{
magic bullet	other processing
c.s.	magic bullet
other processing	c.s.
	other processing
}	}

--> The magic bullet
ensures only 1 process in the critical section at any time

- Desirable properties of a *mutex* mechanisms
-
  - Safety: A maximum of one thread from executes in the critical section at one
  - Deadlock and livelock free
  - Does not require strict round robin access
  - Does not employ busy waiting

Mutex in the pthread system

The *pthread* library provides the *pthread_mutex_lock()* and *pthread_mutex_unlock()* mechanisms

```
27 void get_lock(
28 int tid)
29 {
30    pthread_mutex_lock(&t_mtx);
32 }
```

```
Value of shared is 8000000
Value of shared is 8000000
Value of shared is 8000000
```

Failure to ensure that mutex is provided when required can have literally *fatal* consequences in transportation, medical and military systems

Deadlock even with correct Operating system support

```
P1:
while (1)
{
        pthread_mutex_lock(&mtx1);
        c.s.1();
        other-stuff;
        pthread_mutex_lock(&mtx2);
        c.s.2(();
        pthread_mutex_unlock(&mtx1);
        pthread_mutex_unlock(&mtx2);


}
```

```
P2:
while (1)
{
    pthread_mutex_lock(&mtx2);
    c.s.3();
    other-stuff;
    pthread_mutex_lock(&mtx1);
    c.s.4();
    pthread_mutex_unlock(&mtx1);
    pthread_mutex_unlock(&mtx2);


}
```

Using *ad hack* application level approaches

Use 2 variables to control access with *test then set*:

```
int p1using, p2using;     /* must be in shared memory */
p1:
while(1)
{
     while( p2using == 1 );   /*  loop...busy-wait */
     /* IF PREEMPTED HERE--BOTH  PROCESSES CAN GET INTO CS */
     p1using = 1;
          c.s.;
     p1using = 0 ;
     otherstuff ;
}

p2:
while(1)
{
     while( p1using == 1 );   /*  loop...busy-wait */
     /* IF PREEMPTED HERE--BOTH  PROCESSES CAN GET INTO CS */
     p2using = 1;
          c.s. ;
     p2using = 0 ;
     otherstuff ;
}
```

Use 2 variables to control access with *set then test*:

```
     int p1using, p2using;    /* must be in shared memory */

     p1:
     while(1)
     {
          p1using = 1;
/* IF PREEMPTED HERE--BOTH  PROCESSES CAN DEADLOCK */
          while( p2using == 1 );   /*  loop...busy-wait */
               c.s.;
          p1using = 0 ;
          otherstuff ;
     }

     p2:
     while(1)
     {
          p2using = 1 ;
/* IF PREEMPTED HERE--BOTH  PROCESSES CAN DEADLOCK */
          while( p1using == 1 );   /* loop...busy-wait */
               c.s. ;
          p2using = 0 ;
          otherstuff ;
     }
```

Peterson's Algorithm

Set then test with a turn variable

```
P1:
while (1)
{
        p1using = 1 ;
        turn = 1;
        while( ( p2using == 1 ) && ( turn == 1 ) ) ;

        c.s;
        p1using = 0 ;
        other-stuff;
}
```

```
P2:
while (1)
{
        p2using = 1;
        turn = 2;
        while( ( p1using == 1 ) &&
                (turn == 2 ) ) ;
        c.s
        p2using = 0;
        other-stuff
}
```

Notes:

- Works modulo memory hardware write buffering
- But still uses *busy waiting.* One process loops wasting CPU time instead of yielding.

Classical Cooperating Process Problems

The Producer / Consumer Problem

a.k.a (Bounded buffer || Circular buffer)
Problem involves both
mutex and
general synchronization

The buffer is a circular array of “slots”

Two indices/pointers are used to manage buffer access

in_slot	identifies the next slot in the buffer in which a new item will be placed
out_slot	identifies the next slot in the buffer from which an item will be taken

When implemented in hardware such buffers are commonly called FIFOs

Synchronization problems

When the buffer is empty consumers *must be blocked until an item is produced*

When the buffer is full producers mu*st be blocked until space is available in the buffer*.

Counting semaphores that count both free slots and available items can accomplish both of these missions.

Mutex problems

If there are multiple competing producers, two or more of them *must not be allowed to produce into the same slot.*

If there are multiple competing consumers, two or more of them *must not be allowed to consume the same item.*

If there is only a single producer and a single consumer, the mutex problems don't exist but the syncrhonization problem remains critical.

Readers and Writes problem

Framework of the Problem:

A resource exists that can be read or can be written to.
Multiple processes that can safely read concurrently
Only one process can write at a time and only if nobody is reading.

There are 3 possible solution objectives:

Reader priority - Arriving readers may pass waiting writers to join existing readers. This may lead to *writer starvation.*

Writer priority – Arriving writers by pass waiting readers in the queue. This maximizes concurrent reading but may lead to *reader starvation.*

Strict FIFO – No passing by either readers or writers. This minimizes the level of concurrent reading, but *eliminates starvation... but with a potential nasty side effects if readers read for a VERY LONG TIME.*