Sumedh

- **Process Concept**

An OS runs many programs :-

1. Batch System : Handles tasks
2. Time-shared system : Handles user / prog tasks

A process has diff parts :-

i) code : The actual prog instruc

ii) PC & reg : track where the process is there in execution

iii) Stack : Stores temporary data

iv) data section : stores global vars

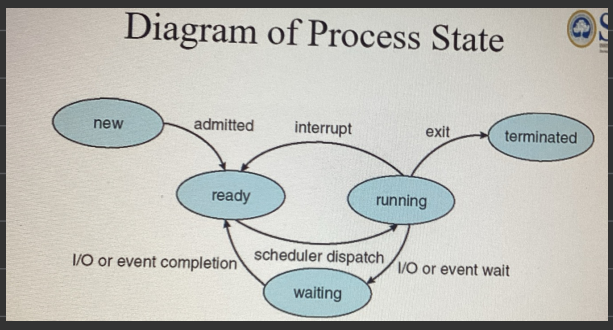
v) Heap : store dyanomic alloc memry

Also if prog is passive, it's stored on disk and if active then run in mmry

Process State :-

As prog. runs it moves through diff states

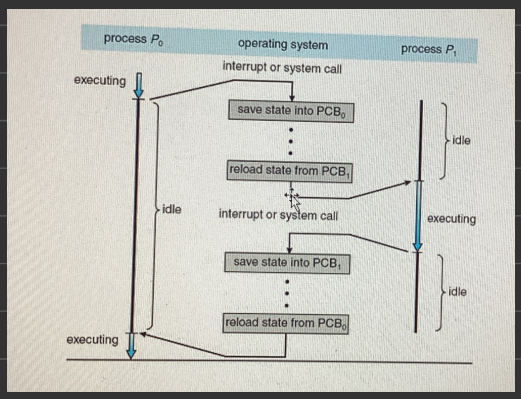
1. New : It's created
2. Running : Currently executing
3. waiting : waits for some event / input
4. Ready : ready to run but waits for its turn in CPU
5. Terminate : done



- **Process Control Block (PCB)**

every process has a pcb that stores :-

1. State : running, ready , etc
2. PC : location of instruc to execute next
3. CPU regs : have info specific to the process
4. CPU scheduling : checks priorities & queue pointers
5. Memry mgmt info : how much mmry process uses
6. Accounting info : CPU used , clock time elapsed since start
7. I/O status info : which I/O process uses

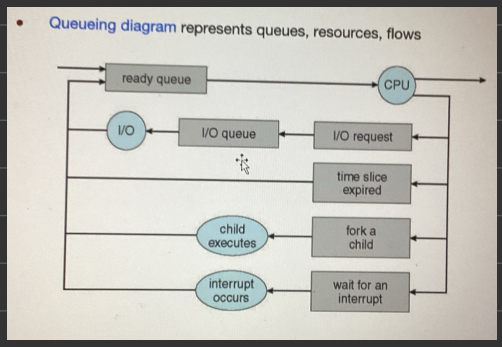


- **Process Scheduling**

OS uses scheduling to maximize CPU use by quickly switching between processes

There are diff queues :-

1. Job queue : set of all processes in the system
2. Ready queue : processes ready to run
3. Device queues : processes waiting for I/O devices



Schedulers :-

1. Short-term scheduler (CPU scheduler) :-
   - Selects which process will run next
   - It's invoked frequently

2. Long-term Scheduler (Job Scheduler) :-
   - decides which process gets added to system
   - Processes are I/O bound & CPU bound
   - Good mix of both

3. Mid-term Scheduler
   - Can pause processes if too many
   - Swap them
   - Bring again when needed

Note* Context Switch: CPU switches processes, saves the old one & loads new one, no real work done here

- Process Ops

- Process Creation

1. A parent process creates child process
2. Processes can share resources with child or their own
3. Par & child share resources (may / may not)
4. Both execute concurrently

How to create: fork(): new process
exec(): loads a new prog in process

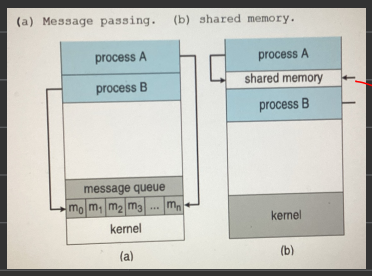
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
     /* parent will wait for the child to complete
     wait(NULL);
     printf("Child Complete");
   }

   return 0;
}
```

- Process termination

1. Process calls exit() system call
2. parent can also terminal child press by abort()
3. If parent waits for child it'll use wait()
4. If not then child becomes zombie
5. If parents terminates & child runs, he's an orphan

- **Communication (Interprocess Comm IPC)**

- Processes can be independant or cooperating
- Coorp process have benefits like info share, fast computat^n, convinience & modularity
- Coorp needs IPC
- Two models of IPC :-



1. IPC - Shared memory

- processes share a specific area for comm.
- The comm is under control of user & not OS
- It lacks sync to avoid data inconsistencies when multiple processes access shared mmry

2. IPC - Message passing

- processes comm. by sending & receiving mssg
- has 2 ops :-
  send (mssg)
  receive (mssg)
- Also processes need to estabolish a comm link to xchng mssgs
- Implementation of comm link
  ① physical : Shared mmry
  h/w bus
  Network
  ② logical : direct / indirect
  sync / Async
  Automatic

- Direct / Indirect Comm

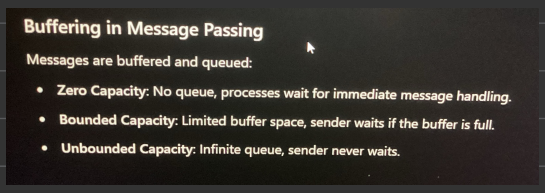
1. Direct :-

- processes comm. by naming each other
  eg Send (P, mssg)
- Some properties of comm link :-
  i) established automaticlly
  ii) bidirectional
  iii) only blw two process

2. Indirect :-

- Mssgs sent & received through mailboxes
- each mailbox has unique id
- Here processes only comm via mailbox
- props :-
  i) uni / bi-dir
  ii) each pair may have multiple comm links
- ops are :-
  Send (A, mssg) A: mailbx
  receive (A, mssg)

Note * Blocking (Sync) : waits till mssgs sent / received
non-blocking (Async) : doesn't



- Eg's of IPC system

1. POSIX

- For shared mmry, segments are created using shm_open()
- mmry size set using ftruncate()
- To print sprintf()

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
```

2. Mach

- mach uses mssg-based comm
- Sys calls r mssg
- each task get mailbox
- Sys calls are msg_send, msg_receive, msg_rpc
- It has flexible handling like wait, return, etc

3. Windows :-

- mssg passing occurs via Local procedure calls
- procedure uses ports like mailboxes
- Server creates private ports
- Client opens connection

• Client Server Systems

1. Sockets
2. Remote Procedure Calls
3. Pipes
4. Remote Method Invocation

· Socets

- They are network comm. endpoints
- It's a endpoint for com
- defined by an IP add & a port
- Alows two-way comm

* all ports <1024 are well-known

- Sockets in java :-
  - i) TCP (Connection-oriented)
  - ii) UDP (Connectⁿ less
  - iii) Multicast - send data to multiple users

· Rpc

- They abstract function calls blw processes on diff systems
- Stubs : proxy that does comm blw clint & srvr
- Windows uses MIDL (Microsft Intrfce definitⁿ lang
- data representatn in Big-endian & little-endian

· Pipes

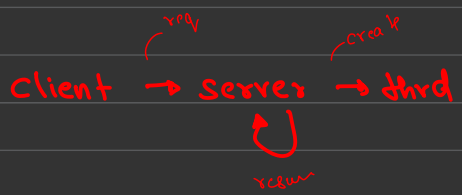
- Allows processes to comm by passing data blw them

- Ordinary pipes :-
  - i) They can't be accessed outside process
  - ii) usually blw parent child
  - iii) unidir
  - iv) producer at one end & consumer at other

- Named pipes :-
  - i) More powerfull than ordinary
  - ii) No par-child relationship
  - iii) provided on both UNIX & windows
  - iv) Bi-dir

- **Threads**

  - Threads are basic units of CPU
  - They allow multiple task to run parallel
  - easier & light
  - simplify coding
  - Improve efficiency
  - Kernel OS is multithreaded



- **Multicore Programming**

They put pressure on programmers & challenges like :-

1. divide work
2. balancing workload
3. split data
4. mange data
5. Test & debug

parallelism : multiple task at same time
Concurrency : making progress on multiple task not at same time

data & task parallelism

data parallelism: distributes subsets of same data across multiple cores

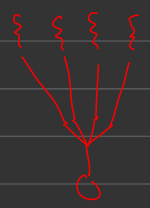
task parallelism: distributes threads across cores

Note * Amdah'l law : predicts performance by adding CPU cores as more cores add up, speed limits

$$\text{speed up} \leq \frac{1}{S + (1-S)/N}$$

- Multithreading models

1. Many-to-one



- Multiple user threads mapped to one kernel thread
- If one blocked, all blocked
- No parallelism
- eg: Solaris green threads

2. One-to-One Model

- Each user thread mapped to a kernel thrd
- More concurrency
- Has overhead limits
- eg Windows, linux, Solaris 9+

3. Many-to-Many

- Allows many user to many many K thrds
- More flexible & parallelism
- Keeps K thrds sufficient
- eg Windows

4. Two-level

- like many-to-many, but allows binding a specific user to K thrd
- eg IRIX, HP-UX, Solaris 8, Tru64 UNIX

- Threading Issues :-

1. Semantics of fork() & exec()
   - does fork() duplicate calling thrd or all?
   - exec() replaces curr thrd

2. Signal Handling

- They manage signals
- every signal has a default handler or user-defined
- For multi-thrd signals are delivered to specific or all threads

3. Thread Cancellation

async deferred

- They can be cancelled immediately or after check
- Thrd to be " is target thrd
- In deferred, they check specific points where cancellation needed

4. Thread local storage

- Allows each thread to have its own copy of data
- diff frm local vars or static data

5. Schedule activations

- Mech to manage thrd in models
- use an intermediate structur called lightweight procss (LWP)

- **Synchronisation**

  Topics (imp only) : 1. The critical section problem
  2. Peterson's sol"
  3. Mutex locks

- **Basics**

- Sync make sure multiple proc work w/o interfering
- To maintain data consistency
- To avoid interruption & data issues

eg producer code : if space is there then add & inc (++)

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

consumer : if there are items, takes one & dec (--)

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

1. Critical section problem

- It's a section of code where a process accesses shared data
- Only 1 process can be in its critical sect^n at a time
- The objective to create protocols to access only 1 process

Algorithm :

```
do {
    while ( turn == j) ;        // Basically wait till turn comes
      // critical //
    turn = j ;
      // remainder //
} while (true)
```

Solution :-

- Mutual exclusion : only 1 in crit sect^n
- progress : if no process in " " then take next process without any delay
- Bounded waiting : limit to process of no. of times in crit sect^n
- Kernel handles thin 2 ways :-
  1. pre-emptive : allows pre-emption
  2. Non-pre : exits kernel, then allows

2. Peterson's Solution

- It's a two-process sol^n
- It's a good algo to solve crit section for 2 processes
- The 2 process share 2 vars and turn up the flag of whose turn it is

Algorithm :

```
do {
    flag [i] = true ;
    turn = j ;
    while ( flag [j] && turn == j);
      // critical //
    flag [i] = false ;
      // remainder //
} while (true)
```

Solution :-

- Mutual exclusion preserved : A prcs only enter where the other one is not ready
- progress guaranteed : if no one in crit sectn , anyone can enter
- Bounded limt

3. Mutex locks

- Simplifies the crit section problem
- It requires a lock before entering & release it afterward
- Functions are acquire() & release()

Algorithm :

```
acquire () {
    while (! available)
        // busy, plz wait //
    available = false ;
}
release () {
    available = true ;
}
do {
    acquire lock ;
    // crit sec //
    release lock ;
    // remainder //
} while (true) ;
```

- **Semaphores**

- It's a tool for process sync to help em coordinate their actions better than mutex locks
- eg Semphore S ;

two ops wait() & signal()

wait() : dec sema val (if 0 then wait)

signal() : inc sema val

- Sema usage / implementation

1. Counting sema : can have wide range valun
2. Binary sema : 0 / 1 val

- Ensure no 2 process in wait() & signal() i.e these both need to be in crit sect"

- Implementation w/o busy waiting

- each sema has waiting queue
- each entry tracks sema's val & pointer next to it
- ops are it add a procss in queue & removes & allow it to run

algo :

```
wait (semaplore *s ) {
    s -> val --;
    if (s -> val < 0) {
        add to s -> list ;
        block();
    }
}
signal ( semaphore *s) {
    s -> val ++ ;
    if (s -> val <= 0 ) {
        remov from s -> list ;
        wakeup(p);
    }
}
```

- Deadlock & Starvation

- Deadlock : When 2 prcs wait forever for events that can be done by one of them

- Starvation : A prcs may never run & just wait

- priority inversion : low-prior has lock of high-prior

- problems in sync
  1. Bounder-buffer
  2. Readers - writers
  3. Dining philo

• Bounded - Buffer

- A fix no. of buffers can hold items
- In sema :-
  - mutex : To control access to buffer
  - full : Counts items in buffer
  - empty : Counts empty slots

Algo :

For producer :-

```
do {
    wait (empty);
    wait (mutex);
    signal (mutex);
    signal (full);
} while (true) ;
```

For consumer :-

```
do {
    wait (full);
    wait (mutex);
    signal (mutex);
    signal (empty);
} while (true) ;
```

• Readers - Writer Problem

- Multiple process share data
- Reader only read , not write
- Writers do both
- Goal is to allow multiple readers & one writer

Algorithm : Writers :-

```
do {
    wait (rw_mutex);
    signal (  "  );
} while (true);
```

Readers :-

```
do {
    wait (mutex);
    read_count ++
    if (  "  == 1)
        wait (rw_mutex);
    signal (mutex);
    wait (mutex)
    read_count -- ;
    if (  "  == 0)
        signal (rw_mutex):
    signal (mutex) } while (true
```

Note : 1. no reader shld wait when writr uses procss
2. writr shld write asap

(It also leads to starvatn) ↗ both

- **Dining - Philosophers problem**

  - It leads to deadlock
  - prevention :-
    1. limit no. of philosophers
    2. Allow philo to pick a chopstick in a specific order
    3. Allow atmost 4 philo
    4. Allow pick chopstk in crit sectn

Algorithm :

```
do {
    wait ( chopstick [i] );
    wait (    "    [(i+1)% 5] )
            // eat //
    signal (      )
    signal (      )
      // think //
} while (true)
```

- **Monitors**

  - A high lvl sync tool
  - Allows only 1 procs to execute at a time
  - provides convinient way to manage shared vars

eg

```
monitor abc {
    procedure P1 ( ) {     }
              Pn ( ) {     }
        some code ( ) {    }
    }
}
```

- **Condition vars**

  - x.wait ( ) & x.signal()
  - When a process signals a condition var, it can either :-
    1. signal & wait
    2. signal & continue

- Monitor soln to dining philo

```
monitor phil {
    enum { think, hung, eat } state [5];
    condition self [5];

void pickup (int i) {
  state [i] = hung;
  test (i);
if (state [i] != eat) self [i]. wait;

void putdown (int i) {
    state [i] = think
    test ((i+4) % 5);
    test ((i+1) % 5);

void test (int i) {
  if (state [(i+4) % 5]) != eat && state [i] == hung && state [(i+1)%5]
                                                        != eat)
                    {
                    state [i] = eat;
                    self [i]. signal()
                  }
            }

      void initial() {
          for (int i=0; i<5; i++) {
              state (i) = think;
           }
        }
```

dk if imp