Operating Systems

Chapter - 1

3/11/21

1.1 What Operating Systems Do?

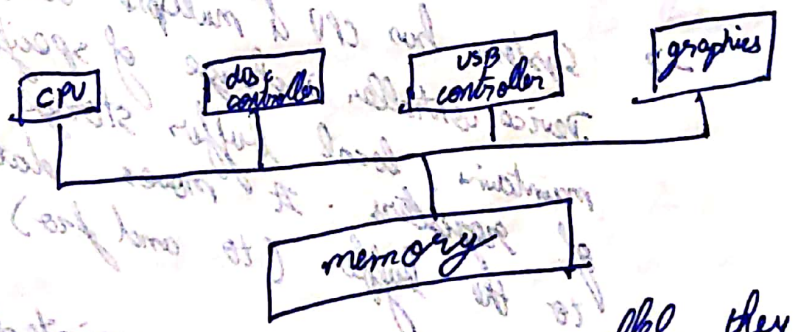
→ Hardware - Resources
Programs - How to use resource
OS - Co-ordinates resource allocation between various programs

→ Different situations use OS differently - resource conserving, performance, efficiency, etc.

→ OS can be treated as control program
Prevents improper usage.

→ OS is the one program that runs always - the kernel. Sometimes, might include system programs.

1.2 Computer Organisation



All can execute in parallel, they compete for memory cycles.

→ When computer starts, bootstrap program (stored in ROM) is initialised. It reads the OS data to load the OS and initialise all hardware. It is followed by bootloader which gives the user the power to load the OS and its applications.

→ System programs are services provided outside of kernel and are loaded at start time (also called system daemons).

→ UNIX - first system process is init (PID 0) and it initialises other daemons.

→ Waits for event - hardware or software interrupt. Then takes action for that.

→ Interrupt is processed quickly - A table of pointers to interrupt service routines, interrupt vector. Usually stored in first 100 units of memory.

→ Instruction from memory → register → decode → fetch operands → store result (optional)

→ System has CPU & multiple device controllers. Device controller in charge of specific device. It maintains local buffer storage & special set of registers. It moves data from device to the buffer (to and fro).

→ Device controller are associated with device drivers and provides the OS with uniform interface to device.

→ For operation, device driver loads register, the controller interprets, moves data to buffer, sends interrupt to the driver, and control goes back to OS with pointer to data.

→ In DMA (Direct Memory Access), the device controller transfers entire block of data from buffer to memory directly.

1.3 Computer Architecture

→ In single processor systems, special purpose process exists and relieves some of the work from the main CPU.

→ Multiple general purpose processors exist sharing bus, memory, clock, etc. They result in increased throughput and reliability

→ Assymmetric multiprocessing – each processor assigned different task with a boss processor supervising.

Symmetric multiprocessing – all processors are peers. They have private registers and cache but common physical memory

Those classification can be because of hardware or software

→ Clustered systems are when multiple systems are loosely coupled (can be connected via network)

1.4 OS Structure

→ Multiprogramming increases CPU utilization by organising jobs such that the CPU always has one to execute. A subset of job pool is kept in memory. When current process waits for I/O, another process is shifted and run.

→ Time sharing runs multiple processes simultaneously by switching among them very frequently.

1.5 OS Operations

→ OS are interrupt driven. If nothing happens, the OS is idle. Events are almost always signalled by trap or interrupt. A trap is a software driven interrupt

→ Must distinguish between OS and user code. There are hardware that allows this differentiation

→ A mode bit is used. Whenever user processes are executing, it is in user mode. When OS services are needed, it uses system call, and it transits to kernel mode

→ At system boot it is kernel mode. Hot starts user application in user mode. Whenever trap or interrupt it switches to kernel mode. The system always switches back before passing back control to user

→ Only kernel mode can execute privileged instructions.

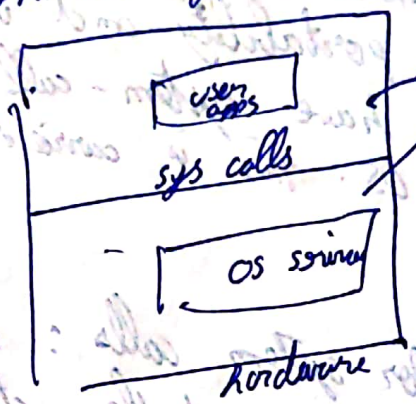
→ System calls provide the means for user to ask OS to perform tasks reserved for it. OS treats this as software interrupt, and goes to the corresponding service routine. Other info is passed via pointers or registers or stack.

→ Here are timers to prevent infinite loop programs. A specified counter is associated that is decreased every tick and once it reaches zero an interrupt occurs.

Chapter 2

2.1 OS Services

→ Environment for execution of programs.



→ User interface, program execution, I/O operations, FS manipulation, communications, error-detection.

2.2 User & OS Interface

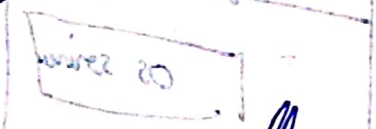
→ Users interface with OS using CLI or GUI (command interpreter). They get and execute next user command. They usually process the command themselves, or load relevant codes for the same from files.

2.3 System calls

→ Syscalls provide interface to services. These are generally available as routines written in C/C++ (sometimes assembly) as well.

→ Copying one file to another :- Getting file names involves bunch of system calls (I/O operations) opening and creating files and related error detection requires separate system calls. Reading and writing also requires syscalls. Closing files and termination also needs them

→ Programmers use APIs like POSIX to interact with the OS, which actually calls the system calls. This helps in portability and usability. Programming languages also have system-call interfaces that links to the OS. This is carried out by a table.



→ Parameters for system calls : registers, address to a memory, and the stack

2.4 Types of Sys Calls

- Process Control :- Ending a process, loading and executing, process attributes, file management, device management, information management, communications

→ Process control :- Loading, ending, wait-events, locks, fork, exec.

→ File Management :- open, read, write, delete, close, file attributes

→ Device Management :- Resources are devices. Request, release, read, write, etc.

→ I/o maintenance:- Transfer of info from user to OS. Time, date, dump, process attribute, etc.

→ Communication:- Shared memory & Message passing. Odt id, open connection, shared memory create, attach, etc.

→ Protection:- permissions, allow users, etc.

2.5 System Programs

→ They are shipped with OS usually. They provide convenient environment. Some are interfaces to sys calls, while others are complex.

→ Filemanagement (manipulates files and directories), status info (retrieve time, disk space), file modification (editors), programming languages (compilers, assemblers, etc.), communications (browser, emails, ssh), background services (network daemon, process schedulers, device drivers).

→ Daemons can be used by OS to run important activities in user context

→ A layered approach is usually used to build OS, with layer 0 being the hardware. Layer above can invoke operations from below layers but not vice versa.

Chapter-3

3.1 Process

→ Process is a program in execution. It consists of the instructions, counter, register contents, process stack (temporary data) and data section (global variables), Heaps (dynamically allocated).

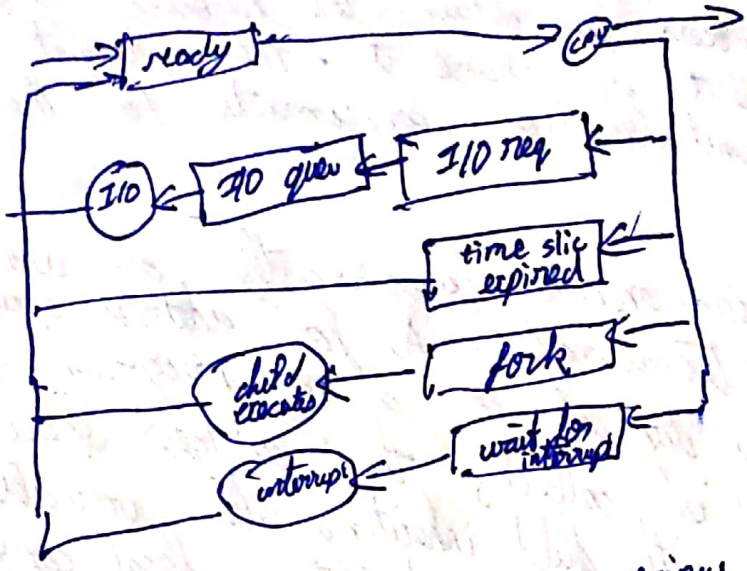
→ New (process created), Running (Instructions executed), waiting (for I/O, signal, message, etc.), Ready (waiting to execute), Terminated (Done).

→ Represented by PCB. Has Process state, prog. counter, CPU registers, scheduling info, memory info (page tables), accounting info., I/O info. Whenever interrupts occur, the current state is stored in the PCB and when it comes back it is loaded from it. Information about threads are also included.

3.2 Scheduling

→ A process scheduler selects an available process for execution on the CPU.

→ Whenever a process enters system, they enter job queue (all processes in system). The processes that are in main memory and ready to execute are in ready que (a linked list of PCBs). When process is interrupted to access a shared I/O device, the process enters the waiting queue for that device called device queue (each device has their own).



→ The selection process among various queues is done by schedulers. The job scheduler (long-term) selects process to load into memory and CPU scheduler (short term) chooses process from ready queue for execution.

→ Long term schedulers can be more costly as frequency of execution is less. It maintains the degree of multiprogramming. It selects a proper mix of I/O & CPU bound processes.

→ There are medium term schedulers which swaps processes out of memory and put them back in later on (swapping). Necessary to accommodate memory changes.

→ When process is interrupted the current context is stored and later when resumed is restored. This is known as context switch (the overhead to switch processes).

3.3 Operations

→ A process (parent) can create new processes (children). Recursively they form a Tree. Init (pid 1) serves as the root process for everything (eg:- init starts login, kthread, sshd, etc.)

→ The child process can obtain the resource from OS or the parent or share it. The parent may pass on initialisation arguments to the child as well.

→ The parent can execute concurrently or wait for child to terminate. As for address space, it could be a duplicate or a new program loaded. If fork() is called, the address space is duplicated with child inheriting all variables, but modifications are carried out in local copy. Exec replaces the memory image with a new program.

→ If parent wants to wait, wait() can be called. Also exec() doesn't return unless error occurs or Terminates

---

→ A process can finish by calling exit() which returns integer value to parent to indicate status. All resources are deallocated. A parent can also issue a terminate command.

→ Cascading termination occurs when child cannot exist without its parent.

→ When process exits, its still there in process table untill parent calls wait() to collect its return status. A process without a parent calling wait() is a zombie process. All processes exist in this state briefly. If parent has already terminated, then the child is called orphan. Orphans are assigned 'init' as the parent, which periodically invokes wait

Sync :- could be blocking, nonblocking send and receive

Buffering :- The sent messages reside in a buffer before being received. Can be zero length bounded and unbounded buffer. The sender must block when queue is full.

→ shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
integer fd ↓ SHM name flags ↓ permissions

ftruncate(shm_fd, 4096); → size

ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0)

ptr for accessing it ↓ establishes memory mapped file

→ Pipes act as conduit allowing a process to communicate. Ordinary unidirectional pipes, pipe(int fd[]). fd[0] is the read end and fd[1] is the write end. Usually parent creates pipe and calls fork to communicate with child. Read & write are done similar to a file.

→ Named pipes are bidirectional and does not need parent-child process. Several processes can use named pipe (FIFOs). They are similar to regular pipes, except the kernel synchronises read and write. Data is not written to disk (only buffer). Have to agree on a name ahead of time.

Writes are atomic as long as they are less than PIPE_BUFF size.

Chapter 4

→ Basic unit of CPU utilisation. It shares code & data & other OS resources with threads of same process. PC, registers & stacks are different. Very efficient to create threads as compared to processes.

→ Very helpful in responsiveness, resource sharing, and, economy and scalability

4.2 Multicore Systems

→ When multiple CPUs - multicore system. In these systems multiple threads can be executed simultaneously instead of just being interleaved

Parallel vs Concurrent. can have concurrency without parallelism

→ Two types :- Data parallelism - subset of same data across cores and performing the same operation. Task parallelism is distributing tasks (could be same/different data).

4.3 Multithreading models

→ Two threads :- User ones are managed without kernel support whereas kernel threads are managed by the OS. There should also exist a relation between them

→ In many-to-one model many user threads are mapped to 1 kernel thread. Only 1 thread can access the kernel at a time

→ Maps each user thread to a kernel thread in one-to-one. Multiple threads can be run in parallel, but overhead in kernel thread creation.

→ Many to many model multiplexes several user threads to smaller kernel threads. This doesn't suffer from the overhead nor the parallelism issue.

4.4 Thread libraries

→ Could be user space or could involve sys calls

→ pthreads refer to POSIX standard defining an API for thread creation & synchronisation.

```
pthread_t tid;
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_create(&tid, &attr, runner, args);
pthread_join(tid, NULL);
```

→ separate threads begin executing in the same function. Each thread has some attributes. The argument passed is in the form of void *. Multiple arguments can be passed in terms of structure. Have to be careful with scope when passing arguments

4.6 Threading Issues

→ Fork() can result in forking all threads and others fork only the one we are using to call fork. Exec() replaces all threads.

→ Signals can be handled by overriding default handler and handles the signals that arrive asynchronously. A lot of option exists on which threads receive the signals

→ The cancellation of threads could be asynchronous or deferred.

Chapter 5

→ When process use shared data race condition can occur when they access same data simultaneously. Eg:- counter++ , counter--

r_1 = counter	r_1 = counter
r_1 = counter + 1	r_1 = counter - 1
counter = r_1	counter = r_1

This can lead to incorrect assignment when instructions are interleaved

5.2 Critical Section

→ when 1 process is using CS no other process is allowed to use it.

→ In the CS problem solution, we must satisfy mutual exclusion, progress (only processes ins entry, exit section can decide the entry of a process to enter CS and cannot postpone indefinitely), bounded waiting (no. of time other process can enter before this)

→ In kernels, non-preamptive kernel is free from race condition

5.3 Peterson's Solution

→ No way to guarantee if the solution will work on modern computers because instructions can be reordered.

Process i & j

```
do { flg[i] = true
     turn = j        → decides which process will enter
     while (flag[j] & turn == j);

     CS

     flg[i] = false;

     reminders

} while (true);
```

→ To prove mutual exclusion:- both flags must be 1, but we can see that turn can be only either i or j.
* "turn == j" will hold P_i until P_j exists & then is not going to change as long as P_i in CS

→ To prove progress, we can see that the process permanently never gets stuck in the while loop. And this leaves the while loop after P_j process leaves and before P_j enters again and thus enters after 1 entry by the other process, preserving bounded waiting

5.4 Synchronisation Hardware

→ Software solutions are not guaranteed to work because of instruction reordering. Hence, we look for hardware ones.

→ CS problem could be solved if in a single processor allows no interrupt when shared varibles are being modified. Not possible in multi processor environment.

→ Hardware instruction that allow (test & set mem) or (swap content) atomically. We can then use these instructions for the CS problem. Executes atomically even in Multicore env by locking the bus.

```
→ test_set(bool *target) {
      bool rv = *target;
      *target = true;
      return rv
  }
```

```
while (test_and_set(&lock));
    // CS
lock = false;
```

```
→ compare_swap(int *val, int exp, int new) {
      int temp = *val;
      if (*val == exp) {
          *val = new;
      }
      return temp;
  }
```

```
while (comp_swap(&lock, 0, 1));
    // CS
lock = 0;
```

→ The above two only satisfy mutual exclusion but not other criteria.

→
```
waiting[i] = true;
key = true;
while (waiting[i] && key) key = test_set(&lock);

waiting[i] = false;

    // CS

j = i+1 mod n
while (j != i && !waiting[j]) j = j+1 mod n;
if (j == i) lock = false;
else waiting[j] = false;
```

The first process will have key=false (local variable) but others must wait. The waiting[] can become false only if another process leaves CS. Thus Progress is also met by making lock = false or waiting[j] = false.

within n-1 turns it will enter

5.5 Mutex Locks

→ Software solutions are much better as they are easier to access. Mutex locks are one such solution. A process must acquire a lock before entering CS and release it once it is done. The lock is a boolean variable.

→
```
acquire() {
    while (!lock);
    lock = false;
}
```

```
release() {
    lock = false;
}
```

Must be performed atomically

→ Disadvantage is the busy waiting (spinlock). It wastes CPU cycles.

Spinlocks are useful when time is short because of a lack of context switch.

→ To execute them atomically, enabling or disabling interrupts will work. Could also use enque in acquire (pushing them in queue & waiting for it to be popped) and suspend, and then deque in release. This stops busy waiting. Another is using test_set in acquire

5.6 Semaphores

→ Semaphore is an integer variable only accessed through atomic operations.

wait() or P

```
wait(S){
    while(S<=0);
    S--;
}
```

signal() or V

```
signal(S){
    S++;
}
```

→ could be binary or counting semaphores.

→ Counting semaphores control access to a resource consisting finite number of times.

Also, solve synchronisation problems.

```
S1                  | wait(synch);
signal(synch);      | S2
```

will make sure S_2 is executed after S_1.

→ Similar to mutex, instead of using busy wait, we can make the process block itself after putting itself in queue in wait(). When some process does signal(), the process in queue is given wakeup signal. But under this definition, semaphores can be negative

---

→ Could also cause deadlock - where a set of process is waiting for an event from another process in the set.

→ 3 processes L < M < H (priority). Process H requires a resource R being used by L. Now if M starts running & prempts L, it affects how long H must wait for R. This is known as priority inversion

→ Solving this needs priority inheritance protocol. All process that are accessing resources needed by a high priority process inherit the higher priority until they are done with the resource.

5.7 Classic Problems

① Bounded buffer (Producer consumer)

```
int n
semaphore mutex = 1, empty = n, full = 0

Producer                          Consumer
wait(empty);                      wait(full);
wait(mutex);                      wait(mutex);
// producer                       // consumer
signal(mutex);                    signal(mutex);
signal(full);                     signal(empty);
```

② Readers-Writers (Database)

→ First readers-writers (no reader be kept waiting unless a writer is writing)

```
semaphore rw_mutex = 1, mutex = 1;
int rc = 0;

mutex - updating rc
rw_mutex - 1st, last readers & all writers

Writer                            Reader
wait(rw_mutex);                   wait(mutex);
// writing                        rc++;
signal(rw_mutex);                 if(rc==1) wait(rw_mutex);
                                  signal(mutex);
                                  // reading
                                  wait(mutex);
                                  rc--;
                                  if(rc==0) signal(rw_mutex);
                                  signal(mutex);
```

Starvation of writers

→ Useful in situations where we know what is each write and more readers than writers. read-write locks require more overhead which compensated by multiple readers.

→ Second readers-writers, priority to writers. But starvation for readers. Can be solved by having wc (count for writers) and allowing readers when wc == 0.

~~5.13 Dining~~

③ Dining Philosopher's problem

→ Need to allocate resources in a deadlock-free & starvation-free manner. A simple solution of 5 semaphores (5 chopsticks) will result in deadlock when all of them grabs 1 chopstick.

→ For that we can allow a philosopher to pick chopsticks if only both are available which is locked in a CS

Chapter 6

→ CPU scheduling relies on the cycle of CPU & I/O burst. The distribution of CPU & I/O bound process is important when we schedule.

→ The short-term (CPU) scheduler selects a process from memory for the CPU to execute. The units in the ready queue are PCBs. The scheduling happens if:

i) If process goes from running → waiting (some event)

ii) when process goes from running → ready (when interrupt occurs)

iii) goes from waiting → ready (when event completion)

iv) when process terminates.

→ 1 & 4 (nonpreemptive) & other wise it is preemptive. Preemptive are difficult to deal with. Could be preempted during system calls. The OSes usually wait for their completion before doing context switch. They disable & reenable interrupts

→ Dispatcher gives control of CPU to process selected by scheduler. Switching context, switch to user mode, jump to proper location. It should be very fast.

6.2 Scheduling Criteria

→ CPU utilisation - The amount of time the CPU is kept busy in percent (0-100). Should be high.

→ Throughput - No. of processes that are completed per unit time.

→ Turnaround time - The interval from time of submission to completion.

→ Waiting time - Waiting time is the amount of time spent waiting in the ready queue.

→ Response time - Time between submission & first response.

6.3 Algorithms

→ FCFS (First come, first serve) - The process that requests CPU first is allotted it. Implemented by a FIFO queue. Average waiting time is high. This might also result in convoy effect when 1 big process holds the CPU and makes the other processes wait for it. It results in lesser CPU & device utilisation. Moreover, its non-preempti-

→ SJFS (Shortest Job First) – This process associates with it the length of next CPU burst. When CPU is available, allotted to the smallest next CPU burst. This algo gives the minimum average waiting time.

Primary difficulty is the problem of estimating next CPU burst. That's why its common in long term scheduling using the user estimated time.

Next CPU burst is predicted as an exponential average of previous CPU bursts measures.

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n$$

can be preemptive or non-preemptive

→ Priority Scheduling – A priority is allocated to each process and CPU is given to highest priority. Equal priority are served in FCFS manner. Usually low numbers represent high priority. Could be implicit or explicit priorities. Again, could be preemptive or non-preemptive.

In this scenario, some low priority process will be starved. We can use aging to counter this (gradually increasing priority with time).

→ Round Robin – Designed for time sharing systems. Similar to FCFS, but preemption is added. A circular ready queue is traversed with each process being allotted 1 time slice. The average waiting is often high. Very low time quanta will result in lot of context switch overhead. 80% of CPU bursts should be shorter than the time quanta.

→ Multilevel Queue - For processes that could be easily classified into different groups. This algo partitions the ready queue into several separate queues. The processes are permanently assigned to 1 queue based on some property and each queue has its own scheduling algo. There is scheduling among the queues as well.

→ Multilevel Feedback Queue - Allows a process to move between queues. It separates process according to CPU bursts (if process uses too much CPU, put into low priority queue)

- no. of queues
- scheduling algo for each queue
- upgrade & degrade a process

6.4 Thread Scheduling

→ It is the kernel-level threads - not processes - that are being scheduled by OS. To run on CPU, user level threads has to be mapped to an associated kernel thread.

→ On many to one model, the user library schedules user-level threads to run on available LWP. This is known as process contention scope. To decide which kernel level thread to run on CPU, the kernel uses system contention scope.

→ Using SCS on in many-to-one model results in mapping threads on a one to-one basis.

→ LWPs runs in user space on top of a single kernel thread and shares its memory address space and system resources with other LWPs of the same process.

Chapter 7

→ A process must request a resource before using it and release it after using it. If the request can't be granted immediately, then the process must wait.

→ A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. Locks are potential source of deadlocks

7.2 Deadlock Characterisation

→ A deadlock can arise if following hold simultaneously
i) Mutual exclusion - At least 1 resource is present in non-shareable mode
ii) A process must be holding 1 resource & waiting for additional ones.
iii) Resources cannot be prempted
iv) There should be a circular wait for resources

→ In a resource allocation graph, we represent deadlocks. $V = \{P_i, R_j\}$ (Process & Resource). $P_i \to R_j$ implies P_i is waiting for R_j. $R_j \to P_i$ implies R_j has been allocated to P_i. The dots inside R_j denotes number of such resource. When requests are satisfied, request edges become assignment edges. Releasing the resource deletes that edge.

→ In such a graph, if graph contains no cycle, no deadlocks. Otherwise, they may exist. If only 1 resource of each type exists, then cycle imply a deadlock. If more than 1 resource of each type exists, then cycle may not imply a deadlock. If process outside the cycle may break it.

7.3 Handling Deadlocks

→ Deadlock prevention - Provides method to stop one of the necessary condition

→ Deadlock avoidance - Using future requests info about processes, OS can avoid deadlocks.

→ Deadlock detection & recovery

→ Ignore deadlocks and let programmers handle it. Use manual methods to recover from deadlocks. This is cheaper.

7.4 Deadlock prevention

Stops 1 necessary condition

→ Mutual exclusion - Not possible as some resources are intrinsically not shareable

→ Hold-wait - i) Each process must request and be allocated all its resource before execution. (Resource utilisation problem)
ii) A process must release all its resource before it can request others. (Starvation)

→ No Preemption - i) If a process requests and is not allotted resource, all its current resource are released and added in waiting queue for set of resources being waited upon by the process.
ii) Preempt a resource that is being currently held by a process waiting for another resources if that is requested.

→ Circular wait - Put a total ordering on the resource and the constraint that resources can only be requested in increasing order or after releasing the higher order resources.

7.5 Deadlock Avoidance

→ OS gets to know about all further sequence of requests and releases and thus makes decisions to prevent deadlocks.

— (Safe state algo)

→ A state is safe if the system can allocate resources to each process (upto its maximum) in some order and still avoid a deadlock.
A sequence of $<P_1, \ldots P_n>$ is a safe sequence for each P_i, the resources can be satisfied by currently available and resources held by P_j $(j < i)$. Only, if safe sequence exists, system is safe.

→ If resource is not available immediately it can wait until previous processes are done.
An unsafe state may lead to deadlocks. So, OS tries to avoid them. The system will allocate resource to a process only if it remains in a safe state.

— (Resource allocation)

→ If there is only 1 instance of each resource, we can use this to avoid deadlocks. We use claim edge $P_i \to R_j$ for potential resource request. Before process starts executing, all its claim edges must be present.

→ The request can only be granted, if after converting the request edge to assignment edge, does not result in a cycle (including all edges). Otherwise it has to wait.

— (Bankers Algo)

→ When new process enters the system, it must declare the maximum number of resources. When resources are requested, allocation takes place only if state is safe after doing so.

→ Data structures - n processes, m resources
- available [m] - no. of free resources of each type
- Max [n][m] - max demand of each process
- Alloc [n][m] - allotted for each process
- Need [n][m] - remaining resource needed by each process

→ i) Work = Available (m), Finish = {False} (n)

ii) Find i such that Finish[i] == false & Need$_i$ ≤ Work.

(Safety detection)

If no i exists go to iv)

iii) Work += Allocation$_i$;
Finish[i] = true
Go to ii)

iv) If Finish[i] == true ∀i, system is safe.

→ Request$_i$ request vector of process P_i.

i) If Request$_i$ ≤ Need$_i$ go to ii). Otherwise raise error

(Resource allocation)

ii) If Request$_i$ ≤ Available$_i$. otherwise wait

iii) Available = Available - Request$_i$
Allocation$_i$ = Allocation$_i$ + Request$_i$
Need$_i$ = Need$_i$ - Request$_i$

If after allocation, state is safe, we allocate. otherwise the process must wait & iii) is undone

7.6 Deadlock detection

→ Have to consider overhead for the detection as well as loss from recovery.

→ In a system with only 1 resource of each type we can use a wait-for graph. Here only vertices are processes $P_i \rightarrow P_j$ implies i is waiting for j. Then, a presence of cycle in the graph detects deadlock.

→ For more than 1 resource,

i) Work = Available
   Finish[i] = (Allocation == 0)

ii) Find i s.t
   Finish[i] = False
   Request[i] ≤ work , If no i, go to 4

iii) Work += Allocation
   Finish[i] = True
   Go to step 2

iv) If there is a Finish[i] = false, then that process P_i is deadlocked

→ Either Invoke the algo at fixed intervals (depending on frequency of deadlocks) or when CPU utilisation drops.

7.7 Recovery

→ Aborting all processes or processes one by one until no deadlock. The chosen process or based on some policies.

→ Could use resource preemption · selecting victim (choose the order for minimum loss), Rollback (move to a previous state or restart process), Starvation (ensure resources are not always preempted from same process).

Chapter 8

→ The main memory only sees a stream of memory address. The main memory & registers are the only general purpose storage that CPU can access directly.

→ We must protect OS data from users and this protection must be done by the hardware as OS doesn't intervene between CPU & its memory access. This is done by allotting a separate memory space for each process. Determined by base & limit register

→ A user program goes through several steps before being executed and the addresses are represented differently in each step. The addresses are bound (map from 1 space to another). Instruction & data to memory address can happen any time

- Compile time - Absolute code are generated only if you know where in main memory this is going to reside
- Load time - compile time addresses are bound to relocatable address. Final binding is delayed until load time. If starting address changes, reload code
- Execution time - If process can be moved during execution, then binding must be delayed until run time

→ Address generated by CPU is known as logical address, and address seen by memory unit is physical address. Compile time & load time address generate same, whereas execution time is not

→ The set of all logical address is the logical address space. Ily for physical. The run time mapping of virtual to physical is done by MMU. A simple scheme is to use the base address to generate physical address from logical ones (just add them)

→ The user program deals with logical address. The final location of a referenced memory address is not determined until the reference is made & only when the memory is involved (a load/store).

→ In dynamic loading, a routine is not loaded until it is called. All routines are in disk in a relocatable load format. When a routine needs to call another routine and if the routine is not loaded, the relocatable linking loader is called to load the desired routine in memory and update the program's address tables

→ Dynamically linked libraries are system libraries linked to user programs when they are run. In static liking these libraries are linked and combined into the binary by the loader but dynamic linking holds of the linking until execution time. In dynamic linking, a stub is included on how to locate the appropriate routine (whether in memory or not).

One version update occurs, the old program won't be affected as it is detected by the version number.

8.2 Swapping

→ A process can be swapped in and out of memory temporarily into (from a backing store (fast disk). If CPU decides to execute a process, the dispatcher

swaps out a process if the chosen process can't fit in memory.

→ If we are swapping, the process must be completely idle. If I/O operations are pending & I/O is accessing the corresponding memory asynchronously swapping is a problem.

→ Swapping is usually disabled until unused memory space falls below a threshold. Or sometimes only a portion of the process is swapped.

8.3 Contiguous Memory

→ The memory is partitioned into OS & user space. OS usually located in low memory. Each process must be allotted a segment of user space.

→ To protect memory, we use the base & limit registers These registers are updated accordingly during the context switch. This scheme allows for the OS size in memory to change dynamically.

→ Divide memory into fixed sized partitions. One process can use multiple partitions. But 1 partition can hold only 1 process. The OS keeps a table indicating available chunks of memory (hole)

→ Memory is usually allotted to the processes coming in until no free hole is large enough to accommodate it. OS can wait or look for other smaller process.

→ If there are too many holes that meets requirement one of them is chosen according to policy. If the chosen hole is too large, it is split - when process terminates it releases its memory, which then returns to set of holes.

→ The policies are first-fit, best fit, worst-fit
→ faster
better

→ But first & best fit suffer from fragmentation (external) Occurs when total space is enough, but they are not contiguous. Statistics show that $\frac{1}{3}$rd of memory is not usable because of this (50% percent rule). In internal fragmentation, the free space within a allotted partition is wasted.

→ Can be solved by compaction if relocation is not static. We put all free memory together. If it is dynamic, we can move the process and then change base address. Can be expensive

9.4 Segmentation

→ A memory management scheme where logical address space is a collection of segments. Each segment has a name and length. Address here is seg name & offset. Segment names are mapped to segment numbers (enumeration). The segments are created during compile time

→ The mapping from logical to physical address is carried by segment table. Each entry has a segment base & limit indexed by the segment number. This is used to generate physical addresses.

9.5 Paging

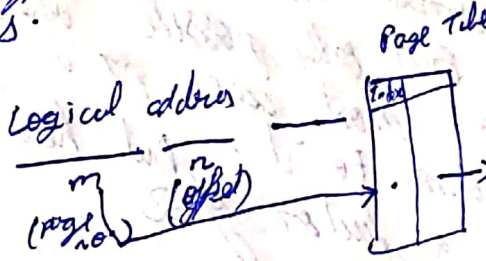
→ Paging helps in prevention of compaction & external fragmentation as well as the problem of fitting memory chunks of varying sizes in backing store.

→ Break the physical memory into fixed size blocks. called frames and logical memory into pages of same size

→ When process is to be executed, its pages are loaded into any available memory frames. This means that logical address is separate from physical one & can have varying spaces

→ Every logical address is divided into page number and offset. The page number is used as an index in page table, which contains the base address of each page in memory. Through this we can generate physical address.

→ The page size is a power of 2, as this helps in identifying offset & page number using bit operations.



Address = $F \times 2^n$ + offset

→ No external fragmentation, but internal fragmentation exists on an average of 0.5 frames per process. This means small page size are suitable, but overhead for PT entries also exist

→ Often, a PT entry is 4 bytes in a 32-bit CPU. A 32-bit entry can point to one of 2^{32} physical frames. If frame size is 4KB, ~~Logical address space is 64 bit~~ then a system can address 2^{44} bytes of physical memory. Usually some other data will also be stored in a PT Entry, making the address space smaller. Paging lets us use more physical memory than that actually exists

→ The page table is filled as soon as a process comes and is allotted its frames. This separates the programmer's view (one single entity) and actual memory (spreadout across). Each process has a separate page table and is unable to access pages outside its space.

→ The OS is aware of the frame details using frame table. Has an entry for each frame. The OS maintains a copy of the page tables which it uses in times of system calls.

→ Each OS has own methods for PTs. A pointer to the page table is stored with register values and when the process comes back the correct hardware PT values must be retrieved.

→ PTs are sometimes stored in registers for high speed efficiency. But only possible when no. of entries are small. So most cases PT is in main memory and PTBR points to it. But here memory access is slowed by a factor of 2

→ To solve this, we use TLB (Translation look-aside buffer). TLB is a cache specially used for PT entries. It consists of keys & value. When an address is looked up, it compares with all keys. If found value, is returned. Usually a part of instruction pipeline. But because of this TLB is small (32 → 1024) entries

→ First address presented to TLB. If found, we have frame address. If its a TLB miss, PT is accessed through physical memory. And we get the frame number. Then we add this to the TLB. If full, some replacement policy is used

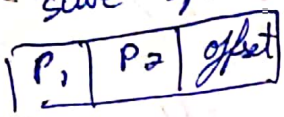
→ TLBs also store Address Space Identifiers in each entry. This helps space protection as ASID uniquely identifies process. If ASIDs do not match, it is a TLB miss. If ASIDs are not supported, TLBs must be flushed every switch.

→ Some additional bits are added to PTEs. Valid/invalid & protection (read/write) bits.

→ Paging can help sharing code (reentrant). Each user's page table maps onto the same copy of the reentrant code being used, but different data.

8.6 Page Table Structure

→ Hierarchical Paging :- we can use 2 level page tables to save space where pagetables are paged.



→ Logical address (forward mapped page table)

. But for 64 bits it usually needs 5 or 6 levels which slows down memory access

→ Hashed Page Tables

The virtual page number is hashed into the table. Multiple virtual numbers are maintained in linked list form to handle collision. If match exists, the frame number is returned

→ Inverted Page Tables

Page tables indexed by page numbers are large. Inverted page tables have 1 entry per frame, with virtual address & process details. If match is found in the page table for a page number P_i, the index of the location is used as physical address of the frame.

Chapter 9

→ In case of real programs, in many cases, the entire program is not needed. Even if they are needed, all of them are not needed simultaneously. The process of running a partially loaded program does not limit program size, more process are run and less swapping

→ Virtual memory involves separation of physical & logical addresses. Allows to provide large virtual memory when only smaller physical memory is available (Only required in physical, others in backing store).

→ Virtual address space, begins at 0 & is contiguous memory but actual memory need not be so. In programming, the heap grows upward & the stack downward. The inbetween space is a part of the virtual address space. These only require physical pages when they are used.

→ Shared libraries and memory might be mapped to different virtual address (according to the process) but exist in a single physical frame

9.2 Demand Paging

→ In demand-paged virtual memory, pages are loaded only when they are demanded during execution. A pager loads pages into memory as they are needed.

→ we need some form of method to distinguish between pages that are in memory & disk. The valid/invalid bit in PTE will work. If invalid & legal memory that page is currently on disk.

→ But what happens to process trying to access invalid bit page? It causes page fault. In case of page faults, we trap to the OS and
i) check for validity of address. If invalid terminate
ii) we take a free frame & page in the required page onto that frame. Mark the PTE valid.
iii) Restart process from the last instruction.

→ Demand paging performs well due to locality of reference.

→ Paging is not really simple as lot of problems occur (moving of bunch of data in single instruction across page borders).

→ An avg. fault takes 8 m.s. To reasonable access time, page faults must be of the order of 25×10^{-4} %.

→ Pages can be brought in from disk directly as long as they are from files. Others must use the swap space (anonymous memory).

9.3 Copy-on-Write

→ In case of fork(), parent & child share the same pages initially and marked copy-on-write. If either of them try to modify it, a copy is created of the page which is mapped to the child's address space & then operations occur.

→ New pages are allocated using zero-fill on demand which zeroes out the frames thus erasing previous content.

9.4 Page Replacement

→ When page fault occurs and no free frames are available, the OS decides to replace some of the existing frames. We choose one of the frames that is not being used. When we swap out such a frame, related tables are updated to indicate that page is not in memory. The new page also updates table after allotment.

→ We can simplify the process by using modify bit. If modify bit is not set, we can simply page in the required page into the chosen frame. Else, that frame must be copied to disk.

→ In replacing frames, a suitable replacement algo must be used.

→ We use a reference string and run the PR algorithm and determine no. of page faults to evaluate performance. Another factor is also the no. of frames available.

→ FIFO :- The oldest page in memory is replaced. Not always good. For some algorithm increasing available frames may result in increase of page fault. This is Belady's anomaly.

→ Optimal Page :- Lowest page fault & never suffers from Belady's anomaly. Replace the page that will not be used for the longest time. Difficult to implement because of future knowledge

→ LRU :- Replace page that hasn't been used in the longest time. It is usually good. But implementation is difficult. We can use counters or stacks to implement it. Since stack algos can't exhibit Belady's anomaly, it applies for LRU as well. Stack algos n frame ⊂ n+1 frames
pages pages

→ LRU has to be approximated. Can use reference bits. Initially, all '0's and used ones are '1's. Or could use 8 bits which shows history of use for past 8 time periods.

→ Second chance :- Similar to FIFO but with reference bit. If set to 1, the page is given 2nd chance & reference is cleared.

→ LFU & MFU (Least & Most frequently) algos are implemented with counters but are not common.

9.5 Allocation of frames

→ Allocate at least certain frames to the process to increase performance initially. Could be equal n/m frames. Or could be proportional to their original size. Could also modify to consider priority.

→ PR algorithms can be global or local. Local can only replace its own pages, whereas global can do so for all processes with lower priority. Global replacement results in a lack of control of its own page fault. But still global is better as it considers overall situation

→ In case of multiple CPUs, they can access different memory location with varying latencies. If memory is allocated close to that CPU, increased performance occurs. This is a case of Non-Uniform Memory Access.

9.6 Thrashing

→ If a process does not have enough frames to accomodate all active pages, it will frequently result in page faults. Thus, a process spends more time paging than executing and is in a situation of thrashing

→ High page fault might result in decrease in CPU use and might prompt the scheduler to bring in more processes which further aggravates the problem. This is because, beyond a certain point, increase in multiprogramming results in decreased throughput.

→ A locality model of execution is defined to predict the number of frames a process will actively use together. One example is working set model, where you use a window to determine the locality.