Memory Management

- In multi-programming OS, multiple programs are loaded in memory.
- RAM memory should be divided for multiple processes running concurrently.
- Memory Mgmt scheme used by any OS depends on the MMU hardware used in the machine.
- There are three memory management schemes are available (as per MMU hardware).
    1. Contiguous Allocation
    2. Segmentation
    3. Paging

Contiguous Allocation

Fixed Partition

- RAM is divided into fixed sized partitions.
- This method is easy to implement.
- Number of processes are limited to number of partitions.
- Size of process is limited to size of partition.
- If process is not utilizing entire partition allocated to it, the remaining memory is wasted. This is called as "internal fragmentation".

Dynamic Partition

- Memory is allocated to each process as per its availability in the RAM. After allocation and deallocation of few processes, RAM will have few used slots and few free slots.
- OS keep track of free slots in form of a table.
- For any new process, OS use one of the following mechanism to allocate the free slot.
    - First Fit: Allocate first free slot which can accommodate the process.
    - Best Fit: Allocate that free slot to the process in which minimum free space will remain.
    - Worst Fit: Allocate that free slot to the process in which maximum free space will remain.
- Statistically it is proven that First fit is faster algo; while best fit provides better memory utilization.
- Memory info (physical base address and size) of each process is stored in its PCB and will be loaded into MMU registers (base & limit) during context switch.
- CPU request virtual address (address of the process) and is converted into physical address by MMU as shown in diag.
- If invalid virtual address is requested by the CPU, process will be terminated.
- If amount of memory required for a process is available but not contiguous, then it is called as "external fragmentation".
- To resolve this problem, processes in memory can be shifted/moved so that max contiguous free space will be available. This is called as "compaction".

Virtual Memory

- The portion of the hard disk which is used by OS as an extension of RAM, is called as "virtual memory".
- If sufficient RAM is not available to execute a new program or grow existing process, then some of the inactive process is shifted from main memory (RAM), so that new program can execute in RAM (or existing process can grow). It is also called as "swap area" or "swap space".

- Shifting a process from RAM to swap area is called as "swap out" and shifting a process from swap to RAM is called as "swap in".
- In few OS, swap area is created in form of a partition. E.g. UNIX, Linux, ...
- In few OS, swap area is created in form of a file E.g. Windows (pagefile.sys), ...
- Virtual memory advantages:
    - Can execute more number of programs.
    - Can execute bigger sized programs.

Segmentation

- Instead of allocating contiguous memory for the whole process, contiguous memory for each segment can be allocated. This scheme is known as "segmentation".
- Since process does not need contiguous memory for entire process, external fragmentation will be reduced.
- In this scheme, PCB is associated with a segment table which contains base and limit (size) of each segment of the process.
- During context switch these values will be loaded into MMU segment table.
- CPU request virtual address in form of segment address and offset address.
- Based on segment address appropriate base-limit pair from MMU is used to calculate physical address as shown in diag.
- MMU also contains STBR register which contains address of current process's segment table in the RAM.

Demand Segmentation

- If virtual memory concept is used along with segmentation scheme, in case low memory, OS may swap out a segment of inactive process.
- When that process again start executing and ask for same segment (swapped out), the segment will be loaded back in the RAM. This is called as "demand segmentation".
- Each entry of the segment table contains base & limit of a segment. It also contains additional bits like segment permissions, valid bit, dirty bit, etc
- If segment is present in main memory, its entry in seg table is said to be valid (v=1). If segment is swapped out, its entry in segment table is said to be invalid (v=0).

Paging

- RAM is divided into small equal sized partitions called as "frames" / "physical pages".
- Process is divided into small equal sized parts called as "pages" or "logical/virtual pages".
- page size = frame size.
- One page is allocated to one empty frame.
- OS keep track of free frames in form of a linked list.
- Each PCB is associated with a table storing mapping of page address to frame address. This table is called as "page table".
- During context switch this table is loaded into MMU.
- CPU requests a virtual address in form of page address and offset address. It will be converted into physical address as shown in diag.
- MMU also contains a PTBR, which keeps address of page table in RAM.

- If a page is not utilizing entire frame allocated to it (i.e. page contents are less than frame size), then it is called as "internal fragmentation".
- Frame size can be configured in the hardware. It can be 1KB, 2KB or 4KB, ...
- Typical Linux and Windows OS use page size = 4KB.

Page table entry

- Each PTE is of 32-bit (on x86 arch) and it contains
  - Frame address
  - Permissions (read or write)
  - Validity bit
  - Dirty bit
  - ...

TLB (Translation Look-Aside Buffer) Cache

- TLB is high-speed associative cache memory used for address translation in paging MMU.
- TLB has limited entries (e.g. in P6 arch TLB has 32 entries) storing recently translated page address and frame address mappings.
- The page address given by CPU, will be compared at once with all the entries in TLB and corresponding frame address is found.
- If frame address is found (TLB hit), then it is used to calculate actual physical address in RAM (as shown in diag).
- If frame address is not found (TLB miss), then PTBR is used to access actual page table of the process in the RAM (associated with PCB). Then page-frame address mapping is copied into TLB and thus physical address is calculated.
- If CPU requests for the same page again, its address will be found in the TLB and translation will be faster

Two Level Paging

- Primary page table has number of entries and each entry point to the secondary page table page.
- Secondary page table has number of entries and each entry point to the frame allocated for the process.
- Virtual address requested by a process is 32 bits including
  - p1 (10 bits) -> Primary page table index/addr
  - p2 (10 bits) -> Secondary page table index/addr
  - d (12 bits) -> Frame offset

Demand Paging

- When virtual memory is used with paging memory management, pages can be swapped out in case of low memory.
- The pages will be loaded into main memory, when they are requested by the CPU. This is called as "demand paging".
  - Swapped out pages
  - Pages from program image (executable file on disk)
  - Dynamically allocated pages

Virtual pages vs Logical pages

- By default all pages of user space process can be swapped out/in. This may change physical address of the page. All such pages whose physical address may change are referred as "Virtual pages".
- Few kernel pages are never swapped out. So their physical address remains same forever. All such pages whose physical address will not change are referred as "Logical pages".

Thrashing

- If number of programs are running in comparatively smaller RAM, a lot of system time will be spent into page swapping (paging) activity.
- Due to this overall system performance is reduced.
- The problem can be solved by increasing RAM size in the machine.

Page Fault

- Each page table entry contains frame address, permissions, dirty bit, valid bit, etc.
- If page is present in main memory its page table entry is valid (valid bit = 1).
- If page is not present in main memory, its page table entry is not valid (valid bit = 0).
- This is possible due to one of the following reasons:
    - Page address is not valid (dangling pointer).
    - Page is on disk/swapped out.
    - Page is not yet allocated.
- If CPU requests a page that is not present in main memory (i.e. page table entry valid bit=0), then "page fault" occurs.
- Then OS's page fault exception handler is invoked, which handles page faults as follows:
    1. Check virtual address due to which page fault occured. If it is not valid (i.e. dangling pointer), terminate the process (sending SEGV signal). (Validity fault).
    2. Check if read-write operation is permitted on the address. If not, terminate the process (sending SEGV signal). (Protection fault).
    3. If virtual address is valid (i.e. page is swapped out), then locate one empty frame in the RAM.
    4. If page is on swap device or hard disk, swap in the page in that frame.
    5. Update page table entry i.e. add new frame address and valid bit = 1 into PTE.
    6. Restart the instruction for which page fault occurred.

Page Replacement Algorithms

- While handling page fault if no empty frame found (step 3), then some page of any process need to be swapped out. This page is called as "victim" page.
- The algorithm used to decide the victim page is called as "page replacement algorithm".
- There are three important page replacement algorithms.
    - FIFO
    - Optimal
    - LRU

FIFO

- The page brought in memory first, will be swapped out first.

- Sometimes in this algorithm, if number of frames are increased, number of page faults also increase.
- This abnormal behaviour is called as "Belady's Anomaly".

OPTIMAL

- The page not required in near future is swapped out.
- This algorithm gives minimum number of page faults.
- This algorithm is not practically implementable.

LRU

- The page which not used for longer duration will be swapped out.
- This algorithm is used in most OS like Linux, Windows, ...
- LRU mechanism is implemented using "stack based approach" or "counter based approach".
- This makes algorithm implementation slower.
- Approximate LRU algorithm close to LRU, however is much faster.

Dirty Bit

- Each entry in page table has a dirty bit.
- When page is swapped in, dirty bit is set to 0.
- When write operation is performed on any page, its dirty bit is set to 1. It indicate that copy of the page in RAM differ from the copy in swap area.
- When such page need to be swapped out again, OS check its dirty bit. If bit=0 (page is not modified) actual disk IO is skipped and improves performance of paging operation.
- If bit=1 (page is modified), page is physically overwritten on its older copy in the swap area.

Process Creation

- System Calls
  - Windows: CreateProcess()
  - UNIX: fork()
  - BSD UNIX: fork(), vfork()
  - Linux: clone(), fork(), vfork()

fork() syscall

- To execute certain task concurrently we can create a new process (using fork() on UNIX).
- fork() creates a new process by duplicating calling process.
- The new process is called as "child process", while calling process is called as "parent process".
- "child" process is exact duplicate of the "parent" process except few points pid, parent pid, etc.
- pid = fork();
  - On success, fork() returns pid of the child to the parent process and 0 to the child process.
  - On failure, fork() returns -1 to the parent.
- Even if child is copy of the parent process, after its creation it is independent of parent and both these processes will be scheduled sepeately by the scheduler.
- Based on CPU time given for each process, both processes will execute concurrently.

How fork() return two values i.e. in parent and in child?

- fork() creates new process by duplicating calling process.
- The child process PCB & kernel stack is also copied from parent process. So child process has copy of execution context of the parent.
- Now fork() write 0 in execution context (r0 register) of child process and child's pid into execution context (r0 register) of parent process.
- When each process is scheduled, the execution context will be restored (by dispatcher) and r0 is return value of the function.

getpid() vs getppid()

- pid1 = getpid(); // returns pid of the current process
- pid2 = getppid(); // returns pid of the parent of the current process

When fork() will fail?

- When no new PCB can be allocated, then fork() will fail.
- Linux has max process limit for the system and the user. When try to create more processes, fork() fails.
- terminal> cat /proc/sys/kernel/pid_max

Orphan process

- If parent of any process is terminated, that child process is known as orphan process.
- The ownership of such orphan process will be taken by "init" process.

Zombie process

- If process is terminated before its parent process and parent process is not reading its exit status, then even if process's memory/resources is released, its PCB will be maintained. This state is known as "zombie state".
- To avoid zombie state parent process should read exit status of the child process. It can be done using wait() syscall.

wait() syscall

- ret = wait(&s);
  - arg1: out param to get exit code of the process.
  - returns: pid of the child process whose exit code is collected.
- wait() performs 3 steps:
  - Pause execution parent until child process is terminated.
  - Read exit code from PCB of child process & return to parent process (via out param).
  - Release PCB of the child process.
- The exit status returned by the wait() contains exit status, reason of termination and other details.
- Few macros are provided to access details from the exit code.
  - WEXITSTATUS()

waitpid() syscall

- This extended version of wait() in Linux.
- ret = waitpid(child_pid, &s, flags);
  - arg1: pid of the child for which parent should wait.
    - -1 means any child.
  - arg2: out param to get exit code of the process.
  - arg3: extra flags to define behaviour of waitpid().
- returns: pid of the child process whose exit code is collected.
  - -1: if error occurred.

exec() syscall

- exec() syscall "loads a new program" in the calling process's memory (address space) and replaces the older (calling) one.
- If exec() succeed, it does not return (rather new program is executed).
- There are multiple functions in the family of exec():
  - execl(), execlp(), execle(),
  - execv(), execvp(), execve(), execvpe()
- exec() family multiple functions have different syntaxes but same functionality.

Synchronization

- Multiple processes accessing same resource at the same time, is known as "race condition".
- When race condition occurs, resource may get corrupted (unexpected results).
- Peterson's problem, if two processes are trying to modify same variable at the same time, it can produce unexpected results.
- Code block to be executed by only one process at a time is referred as Critical section. If multiple processes execute the same code concurrently it may produce undesired results.
- To resolve race condition problem, one process can access resource at a time. This can be done using sync objects/primitives given by OS.
- OS Synchronization objects are:
  - Semaphore, Mutex

Semaphore

- Semaphore is a sync primitive given by OS.
- Internally semaphore is a counter. On semaphore two operations are supported:
  - wait operation: dec op: P operation:
    - semaphore count is decremented by 1.
    - if cnt < 0, then calling process is blocked.
    - typically wait operation is performed before accessing the resource.
  - signal operation: inc op: V operation:
    - semaphore count is incremented by 1.
    - if one or more processes are blocked on the semaphore, then one of the process will be resumed.
    - typically signal operation is performed after releasing the resource.
- Semaphore types
  - Counting Semaphore
    - Allow "n" number of processes to access resource at a time.

        - Or allow "n" resources to be allocated to the process.
    - Binary Semaphore
        - Allows only 1 process to access resource at a time or used as a flag/condition.

Mutex

- Mutex is used to ensure that only one process can access the resource at a time.
- Functionally it is same as "binary semaphore".
- Mutex can be unlocked by the same process/thread, which had locked it.

Semaphore vs Mutex

- S: Semaphore can be decremented by one process and incremented by same or another process.
- M: The process locking the mutex is owner of it. Only owner can unlock that mutex.
- S: Semaphore can be counting or binary.
- M: Mutex is like binary semaphore. Only two states: locked and unlocked.
- S: Semaphore can be used for counting, mututal exclusion or as a flag.
- M: Mutex can be used only for mutual exclusion.

Deadlock

- Deadlock occurs when four conditions/characteristics hold true at the same time.
    - No preemption: A resource should not be released until task is completed.
    - Mutual exclusion: Resources is not sharable.
    - Hold & Wait: Process holds a resource and wait for another resource.
    - Circular wait: Process P1 holds a resource needed for P2, P2 holds a resource needed for P3 and P3 holds a resource needed for P1.

Deadlock Prevention

- OS syscalls are designed so that at least one deadlock condition does not hold true.
- In UNIX multiple semaphore operations can be done at the same time.

Deadlock Avoidance

- Processes declare the required resources in advanced, based on which OS decides whether resource should be given to the process or not.
- Algorithms used for this are:
    - Resource allocation graph: OS maintains graph of resources and processes. A cycle in graph indicate circular wait will occur. In this case OS can deny a resource to a process.
    - Banker's algorithm: A bank always manage its cash so that they can satisfy all customers.
    - Safe state algorithm: OS maintains statistics of number of resources and number processes. Based on stats it decides whether giving resource to a process is safe or not (using a formula):
        - Max num of resources required < Num of resources + Num of processes
            - If condition is true, deadlock will never occur.
            - If condition is false, deadlock may occur