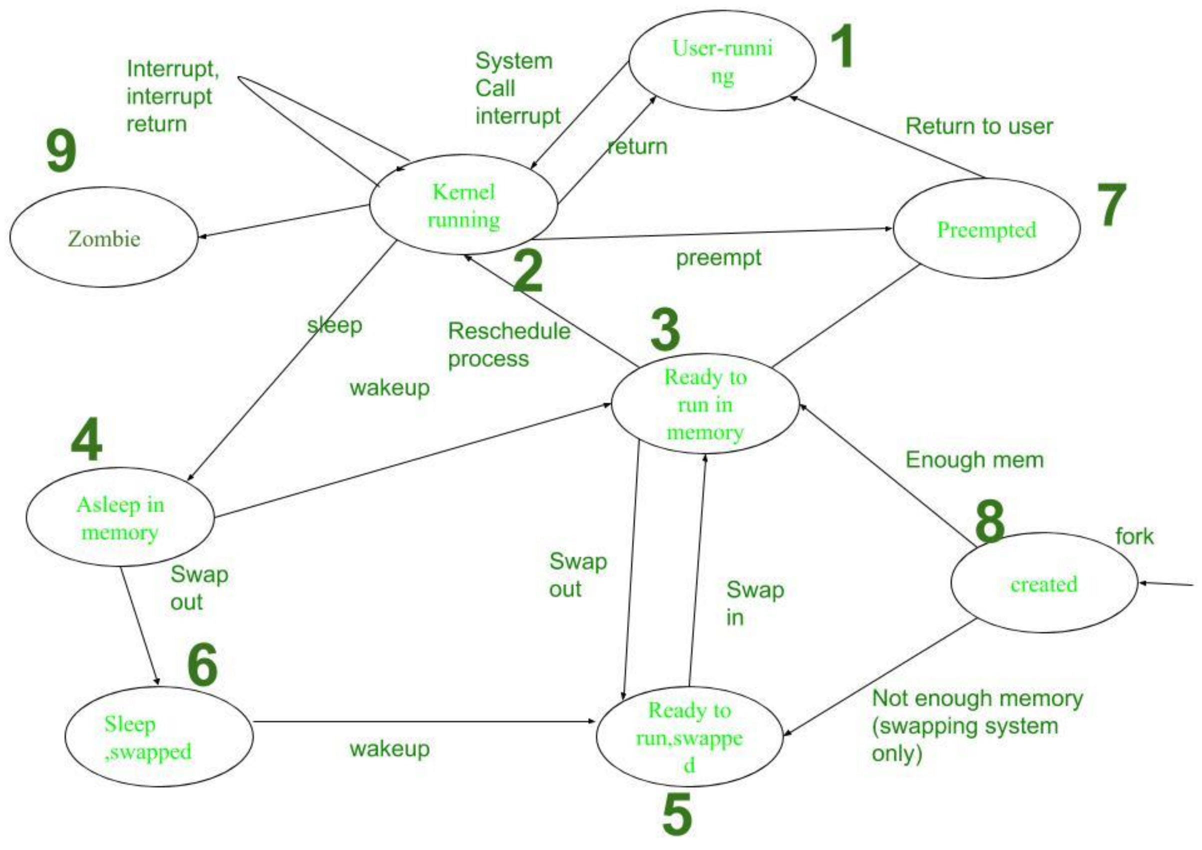
Explain in detail the process state transition in an Unix OS, and draw a neat diagram illustrating the various states a process can go through during its execution.

- Process is an instance of a program in execution. A set of processes combined together make a complete program. There are two categories of processes in Unix, namely
    1. User processes: They are operated in user mode.
    2. Kernel processes: They are operated in kernel mode.

Process States :

- The lifetime of a process can be divided into a set of states, each with certain characteristics that describe the process.



Process Transitions

The working of Process is explained in following steps:

1. **User-running:** Process is in user-running.
2. **Kernel-running:** Process is allocated to kernel and hence, is in kernel mode.
3. **Ready to run in memory:** Further, after processing in main memory process is rescheduled to the Kernel.i.e.The process is not executing but is ready to run as soon as the kernel schedules it.
4. **Asleep in memory:** Process is sleeping but resides in main memory. It is waiting for the task to begin.

5. **Ready to run, swapped:** Process is ready to run and be swapped by the processor into main memory, thereby allowing kernel to schedule it for execution.
6. **Sleep, Swapped:** Process is in sleep state in secondary memory, making space for execution of other processes in main memory. It may resume once the task is fulfilled.
7. **Pre-empted:** Kernel preempts an on-going process for allocation of another process, while the first process is moving from kernel to user mode.
8. **Created:** Process is newly created but not running. This is the start state for all processes.
9. **Zombie:** Process has been executed thoroughly and exit call has been enabled. The process, thereby, no longer exists. But, it stores a statistical record for the process. This is the final state of all processes.

What is Manipulation of the process address space?

- In Unix-like operating systems, a process can transition through several states during its execution . These states represent the different stages a process goes through from creation to termination.
- The UNIX system divides its virtual address space in logically separated *regions*. The regions are contiguous area of virtual address space.
- The region table entry contains the information necessary to describe a region. In particular, it contains the following entries:
    - A pointer to the inode of the file whose contents were originally loaded into the region
    - The region type (text, shared memory, private data or stack)
    - The size of the region
    - The location of the region in physical memory.
    - The reference count.
    - The status of a region, which may be a combination of

— in the process of being loaded into memory

— valid, loaded into memory.

- Following are the opeartions that manipulate region :
    1. Lock a region.
    2. Unlock a region.
    3. Allocate a region.
    4. Attach a region.
    5. Change the size of a region.
    6. Load a region.
    7. Free a region.
    8. Detach a region.
    9. Duplicate a region.

Memory allocation –

- During certain system calls (such as fork, exec, and shmget for shared memory), the kernel allocates a new memory region for a process.
- The kernel maintains a region table, where each entry represents a memory region. These entries can be either on a free linked list or an active linked list.
- The allocreg algorithm ensures that processes receive locked, allocated memory regions for their execution. These regions can be shared among processes based on the associated inode.

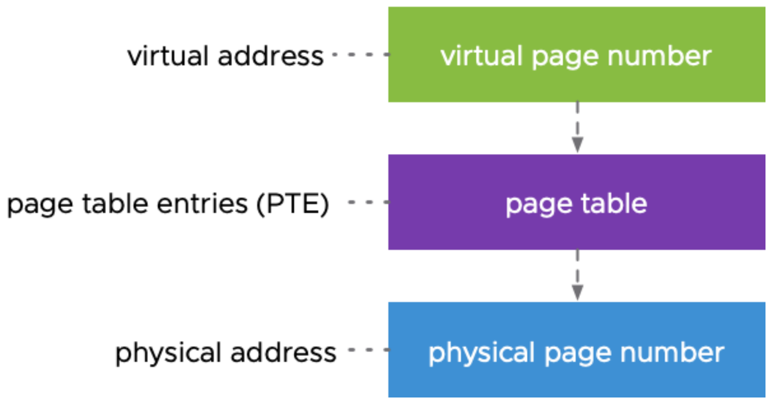
- When allocating a region, the kernel performs the following steps:
    - Removes the first available entry from the free list.
    - Places it on the active list.
    - Locks the region to prevent concurrent modifications.
    - Marks the region as either **shared** or **private**.
    - Associates the region with an **inode** (explained below).

Deallocation / Detaching a region –

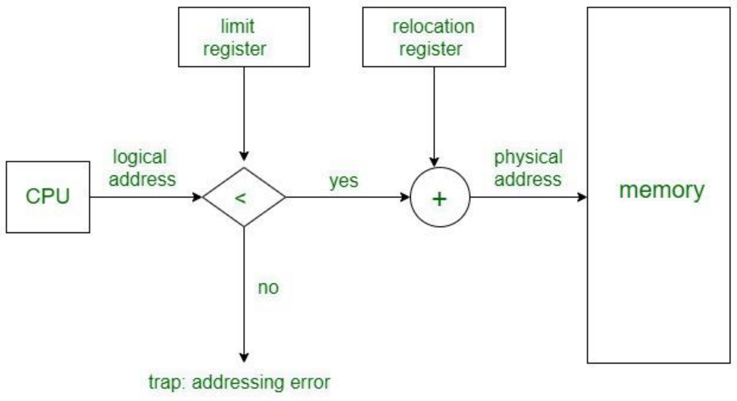
- The kernel performs region detachment during specific system calls, such as exec, exit, and shmdt.
- During this process, the kernel:
    - Updates the **pregion entry**, which contains information about the shared memory region.
    - Invalidates the associated **memory management register triple**, effectively cutting the connection to the physical memory.
    - Note that the invalidation applies specifically to the process, not to the entire shared memory region (as seen in the freereg algorithm).
    - Decrements the **region reference count**.
- If the region's reference count drops to 0 and there's no reason to keep the region in memory (as determined later), the kernel frees the region using the freereg algorithm.
- Otherwise, if the region is still needed by other processes, the kernel only releases the region and inode locks.

Describe Mapping virtual addresses to Physical Addresses.

- Mapping virtual addresses to physical addresses is crucial for efficient memory management in computer systems. In Contiguous memory allocation mapping from virtual addresses to physical addresses is relatively straightforward.
- To map virtual memory addresses to physical memory addresses, page tables are used. A page table consists of numerous page table entries (PTE).



- If we take a process from secondary memory and copy it to the main memory, the addresses will be stored in a contiguous manner, so if we know the base address of the process, we can find out the next addresses.
- The Memory Management Unit is a combination of 2 registers –
  1. Base Register (Relocation Register)
  2. Limit Register.
- **Base Register** – contains the starting physical address of the process.
- **Limit Register** -mentions the limit relative to the base address on the region occupied by the process.
- The logical address generated by the CPU is first checked by the Limit Register.
- If the value of the logical address generated is less than the value of the Limit Register, the base address stored in the Relocation Register is added to the logical address to get the physical address of the memory location.
- If the logical address value is greater than the Limit Register, then the CPU traps to the OS, and the OS terminates the program by giving a fatal error.



- In Non Contiguous Memory allocation, processes can be allocated anywhere in available space. The address translation in non-contiguous memory allocation is difficult.
- There are several techniques used for address translation in non contiguous memory allocation like Paging, Multilevel paging, Inverted paging, Segmentation, Segmented paging.
- Different data structures and hardware support like TLB are required in these techniques.

Explain in brief The Context of a Process.

- The context of a process refers to the collection of data structures and information that the operating system uses to manage and control the execution of that process.
- The context of a process made up of three areas are given below:
  - A user-level context that has contents of its (user) address space.
  - A register context that has contents of hardware registers.
  - A system context that has contents of Kernel data structures that related to the process.

The User Level Context :

- User level context consists of the process text, data, user stack and shared memory that is in the virtual address space of the process.
- The part which resides on swap space is also part of the user level context.

The Register Context :

The register context consists of the following components:

- Program counter stores the address of the next instruction to be executed. It is an address in the kernel or in user address space.
- The processor status register (PS) denotes hardware status related to the process. It has sub-fields which specify if last instruction overflowed, or resulted in 0, a positive or negative value, etc. It also specifies the current processor execution level and current and most recent modes of execution (such as the kernel, user).
- The stack pointer points to the current address of the next entry in the kernel or user stack. If it will point to the next free entry or last used entry is dependent on the machine architecture. The direction of the growth of stack (toward numerically higher or lower addresses) also depend on machine architecture.
- The general purpose registers contain data generated by the process during its execution.

The System Context :

- The system level context has two parts a "static part" and a "dynamic part". A process has one static part throughout its lifetime. But it can have a variable number of dynamic parts.
- The static part consists of the following parts:
  - The process table entry
  - The u-area
  - P-region entries, region tables, and page tables.

Dynamic Part :

- Kernel Stack: A separate stack used by the kernel to execute kernel-level functions on behalf of the process.
- System Call Context: Information related to the process's interaction with the operating system through system calls.
- Interrupt Context: Data saved by the system when an interrupt occurs, allowing the system to resume execution after handling the interrupt.

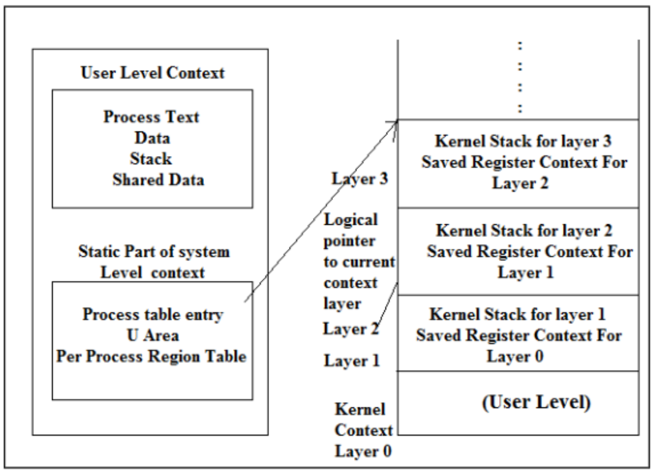


Fig. Formation of context of the process

- The right side of the figure shows the dynamic part of the context. It consists of several stack frames where each stack frame contains saved register context of the previous layer and the kernel stack as it executes in that layer.

Write short note on Context switch.

- Context switching in an operating system involves saving the context or state of a running process so that it can be restored later, and then loading the context or state of another. process and run it.
- As seen previously, the kernel permits a context switch under 4 situations:
  - When a process sleeps
  - When a process exits
  - When a process returns from a system call to user mode but is not the most eligible process to run.
  - When a process returns from an interrupt handler to user mode but is not the most eligible process to run.
- The procedure for a context switch is similar to handling interrupts and system calls.
- Following are the steps involved in context switching –

Decide whether to do a context switch, and whether a context switch is permissible now:

- Before initiating a context switch, the operating system evaluates whether it's necessary to switch to another process.
- Various factors influence this decision, including time quantum expiration, I/O completion, or a higher-priority process becoming ready to execute.
- Additionally, the system must ensure that a context switch is permissible at the current moment, considering system constraints, priorities, and any critical sections that should not be interrupted.

Save the context of the "old" process:

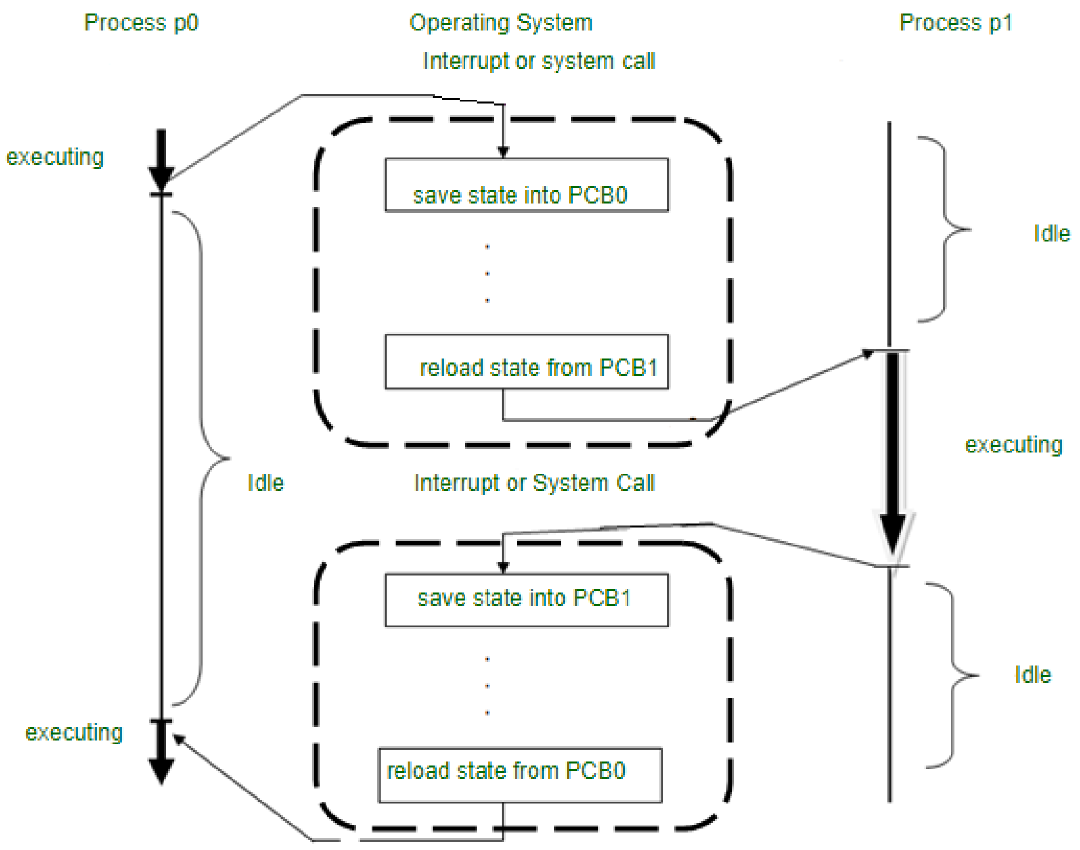
- Once the decision to switch processes is made, the operating system saves the context of the currently executing or "old" process.
- This involves preserving the state of the process by saving critical data such as the contents of the processor's registers, program counter, stack pointer, and any other relevant execution state.
- Furthermore, system-level context, such as memory mappings, file descriptors, and open sockets, may also need to be saved to maintain process integrity.

Find the "best" process to schedule for execution, using the process scheduling algorithm:

- With the context of the old process saved, the operating system selects the next process to execute.
- This selection is based on the process scheduling algorithm, which considers factors such as process priority, CPU burst time, and scheduling policy.
- The goal is to identify the most suitable process for execution, balancing system performance, responsiveness, and fairness.

Restore the context of the selected process:

- Once the new process is chosen, the operating system restores its context from the previously saved state.
- This involves loading the saved data, including the contents of the processor's registers, program counter, stack pointer, and any other relevant execution state.
- Additionally, system-level context, such as memory mappings, file descriptors, and open sockets, are restored to ensure the process can continue its execution seamlessly.


Process p0
Operating System
Process p1
Interrupt or system call
executing
save state into PCB0
Idle
reload state from PCB1
executing
Idle
Interrupt or System Call
save state into PCB1
Idle
executing
reload state from PCB0

Explain with diagram Loading of Region.

- The UNIX system divides its virtual address space in logically separated regions. The regions are contiguous area of virtual address space.
- In a system supporting demand paging, the kernel can "map" a file into process address space during the exec system call, arranging to read individual physical pages later on demand.
- Following is the process of loading a region :

Mapping a File into Process Address Space:

- In systems with demand paging, during the exec system call, the kernel establishes a connection between a file and the process's virtual memory space.
- This allows the process to access the file's contents later when specific parts are needed, rather than loading everything into memory at once.

Loading an Executable File into Memory:

- If demand paging is not supported, the kernel copies the entire executable file into the process's memory space during the exec system call.
- It loads the different regions of the process, such as code and data segments, into specific virtual addresses defined in the executable file.

Setting Up Virtual Memory Regions:

- "Loading a region" refers to the process of setting up a part of the process's virtual memory space with the contents of an executable file.
- This ensures that the process can access the data and instructions in the file when it runs, without needing to fetch them from disk each time.

Detecting Illegal Memory Accesses:

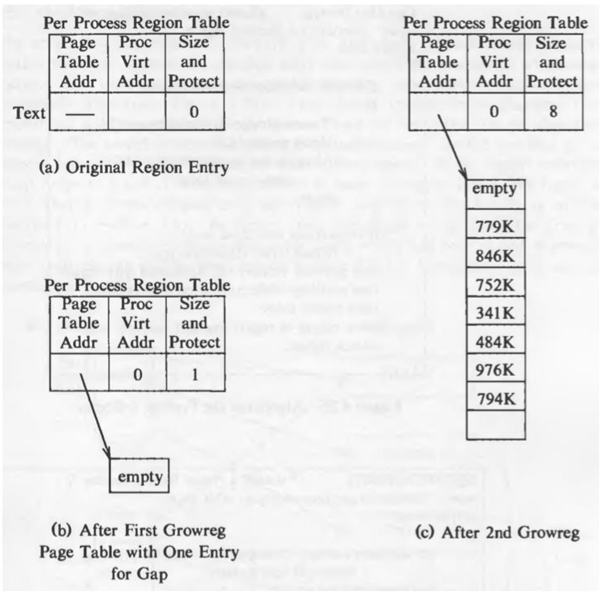
- Sometimes, there may be gaps between the loaded regions in the process's memory space.
- The kernel may use these gaps to catch illegal memory accesses, such as attempting to access address 0, which is often a sign of programming errors.

The algorithm loadreg is given below:

```
/*  Algorithm: loadreg
 *  Input: pointer to per process region table entry
 *       virtual address to load region
 *       inode pointer of file for loading region
 *       byte offset in file for start of region
 *       byte count for amount of data to load
 *  Output: none
 */
 {
         increase region size according to eventual size of region (algorithm: growreg);
         mark region state: being loaded into memory;
         unlock region;
         set up u-area parameters for reading file:
                 target virtual address where data is read to,
                 start offset value for reading file,
                 count of bytes to read from file;
         read file into region (internal variant of read algorithm);
```

```
        lock region;
        mark region state: completely loaded into memory;
        awaken all processes waiting for region to be loaded;
}
```

- Example -



- Imagine the kernel needs to load a piece of text, 7 kilobytes in size, into a specific part of a process's memory.
- However, the kernel wants to leave a gap of 1 kilobyte at the beginning of this memory region.
- First, the kernel sets up a table entry and attaches the memory region at virtual address 0 of the process using certain algorithms.
- Then, it uses a function called "loadreg," which internally calls another function called "growreg" twice:
    1. The first call to "growreg" is to make space for the 1 kilobyte gap at the beginning of the region.
    2. The second call to "growreg" is to allocate storage for the actual contents of the region. This includes setting up a page table for the region.
- After setting up the memory region, the kernel prepares to read the text from a file. It specifies a byte offset in the file and reads 7 kilobytes of data from that offset.
- This data is then loaded into the memory region starting at virtual address 1 kilobyte into the process's memory.

Explain algorithm for Process termination.

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system calL
- At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).
- All the resources of the process-including physical and virtual memory, open files, and I/0 buffers-are deallocated by the operating system
- This algorithm outlines the steps involved in the **exit()** system call, from initiating process termination to performing cleanup and notification processes by the operating system :

```
/*  Algorithm: exit
 *  Input: status value (optional)
 *  Output: none
 */
exit(status):
   perform cleanup operations associated with the terminating process;
   deallocate resources such as memory, file descriptors, and I/O buffers;
   if status value is provided:
      return status value to parent process;
   notify parent process about the termination event;
   update process table to reflect termination of the process;
   execute any registered signal handlers for termination signals;
   update parent process to handle child processes, if applicable;
   perform cleanup tasks related to termination at the operating system level;
```

- Explanation of the algorithm -
  1. **Input and Output Specification:**
     - Input: The algorithm accepts an optional status value, which represents the outcome or result of the process's execution.
     - Output: There is no explicit output specified for this algorithm.
  2. **Cleanup Operations:**
     - The algorithm begins by performing cleanup operations associated with the terminating process.
     - This involves releasing or deallocating resources that were allocated to the process during its execution.
     - Resources typically include memory, file descriptors, and I/O buffers.
  3. **Status Value Handling:**
     - If a status value is provided as input to the **exit()** function, the algorithm returns this value to the parent process.
     - This status value can convey information about the termination event, such as success or failure, to the parent process.
  4. **Notification to Parent Process:**
     - The algorithm notifies the parent process about the termination event of the child process.
     - This notification ensures that the parent process is informed of the termination and can perform any necessary cleanup or handling tasks.
  5. **Update Process Table:**
     - The algorithm updates the process table maintained by the operating system to reflect the termination of the process.
     - This involves removing the entry corresponding to the terminated process from the process table to release system resources.
  6. **Signal Handler Execution:**

- Any registered signal handlers for termination signals are executed.
- Signal handlers may perform additional cleanup or execute specific actions in response to the termination event.

7. **Parent Process Handling:**
    - If the terminating process has child processes, the algorithm updates the parent process to handle these child processes appropriately.
    - This ensures that orphaned child processes are properly managed after the termination of their parent process.
8. **Operating System Cleanup:**
    - Finally, the algorithm performs any cleanup tasks related to termination at the operating system level.
    - This may include releasing system-level resources allocated to the terminated process and updating system-wide data structures accordingly.