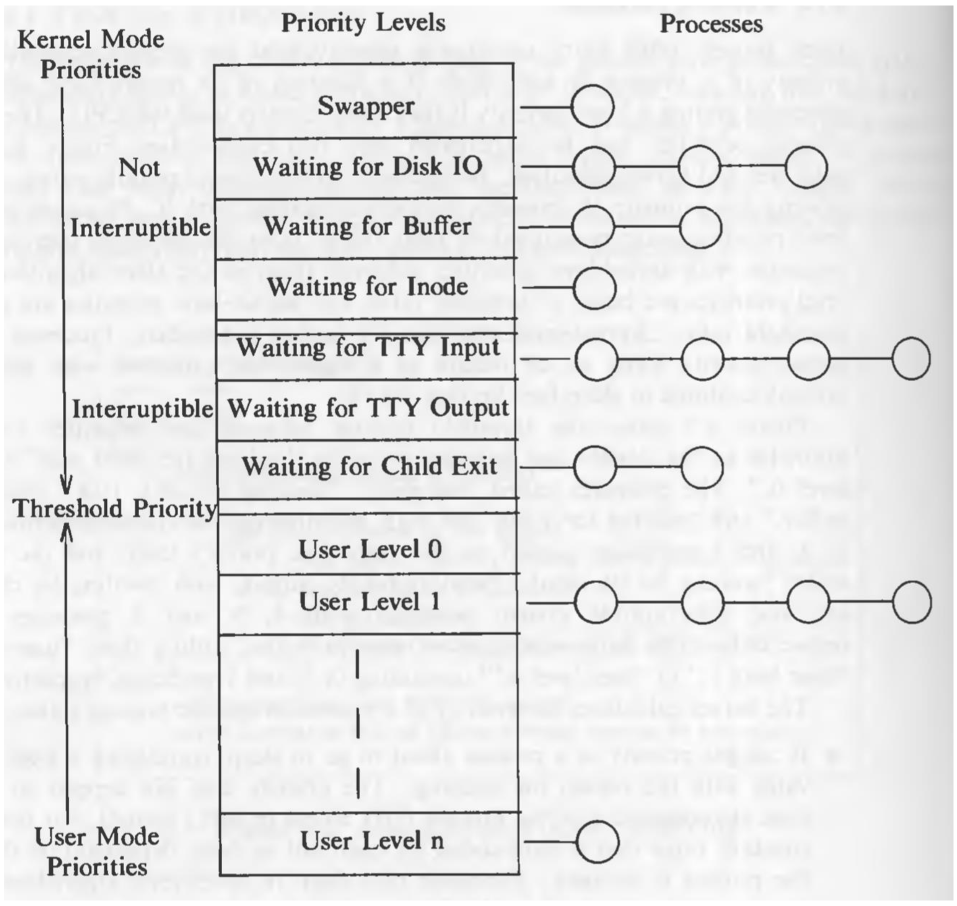
Draw and explain user level and kernel level priority

- In UNIX-like operating systems, such as Linux, the scheduler employs a round-robin with multilevel feedback algorithm.
- This scheduler organizes processes into different priority queues and determines their execution order based on their priority levels.
- Each process table entry contains a priority field. The priority is a function of recent CPU usage, where the priority is lower if a process has recently used the CPU.
- The range of priorities can be partitioned in two classes: user priorities and kernel priorities :



User-Level Priorities:

- User-level priorities are assigned to processes that are primarily user-space applications.
- These priorities are managed by the system and are below a certain threshold value. The priority range typically spans from the highest priority, which is 0, to some maximum value, which is the lowest priority.

1. **Preemption:**

- Processes with user-level priorities can be preempted when they return from the kernel to user mode.
- This means that if a user-level process is currently running and its time quantum expires, the kernel can interrupt it and move it to the appropriate priority queue.

2. **Priority Adjustment:**

- When a process returns from kernel mode to user mode, its priority is adjusted.
- This adjustment is necessary to ensure fairness among processes. The priority is lowered as a penalty for using valuable kernel resources.

3. **Priority Calculation:**

- The priority of user-level processes is recalculated periodically, typically every clock tick.
- This recalculation takes into account the recent CPU usage of the process.
- The formula for recalculating the priority involves the decay of recent CPU usage and the base level user priority.

Kernel-Level Priorities:

- Kernel-level priorities are assigned to processes that are involved in kernel operations or system-level tasks.
- These priorities are above a certain threshold value and are managed differently from user-level priorities.

1. **Preemption:**

- Processes with kernel-level priorities may not be preempted under certain conditions.
- For instance, processes with high kernel priority continue to sleep even on receipt of a signal.

2. **Priority Assignment**:

- The kernel assigns priorities to processes entering sleep states based on the reason for the sleep.

- Processes waiting for critical resources, such as disk I/O completion, are assigned higher priorities to reduce system bottlenecks.

3. **Priority Recalculation**:

- The priorities of kernel-level processes do not change as frequently as user-level priorities.
- Recalculation occurs when a process exits a critical region of the kernel.

Explain the algorithm for exit() system call.

- Processes on the UNIX system exit by executing the *exit* system call. When a process *exit*s, it enters the zombie state, relinquishes all of its resources, and dismantles its context except for its process table entry.

exit (status);

- where **status** is the exit code returned to the parent.
- The process may call exit explicitly, but the startup routine in C calls exit after the main function returns.
- The kernel may call exit on receiving an uncaught signal. In such cases, the value of status is the signal number.

The algorithm for exit is given below:

```
/*  Algorithm: exit
 *  Input: return code for parent process
 *  Output: none
 */

{
        ignore all signals;
        if (process group leader with associated control terminal)
        {
                send hangup signal to all members of the process group;
                reset process group for all members to 0;
        }
        close all open files (internal version of algorithm close);
        release current directory (algorithm: iput);
        release current (changed) root, if exists (algorithm: iput);
        free regions, memory associated with process (algorithm: freereg);
        write accounting record;
        make process state zombie;
        assign parent process ID for all child processes to be init process (1);
                if any children were zombie, send death of child signal to init;
        send death of child signal to parent process;
        context switch;
}
```

Steps –

1. Ignore all signals to prevent interruptions.
2. If process is a group leader with a control terminal, send hangup signal to all members and reset their group ID.
3. Close all open files to release resources.

4. Release current working directory.
5. Release current root directory if it was changed during execution.
6. Free allocated memory regions to prevent memory leaks.
7. Write runtime statistics and accounting data to records.
8. Transition process to zombie state, marking it as terminated but still in process table.
9. Assign parent process ID to init for child processes.
10. Send death of child signal to init if any children were zombies.
11. Send death of child signal to parent process.
12. Perform a context switch to select the next process for execution.

Explain the different functions of the clock interrupt handler

- The clock interrupt handler is a fundamental component of an operating system responsible for managing time-related functions and coordinating activities based on the system clock.
- When the system clock generates an interrupt at regular intervals, the clock interrupt handler is invoked to handle these interrupts.
- The functions of the clock interrupt handler are to:

1. **Restart the Clock:** The handler ensures that the system clock is consistently operational, managing interruptions effectively to maintain accurate timing across the system.

2. **Schedule Invocation of Internal Kernel Functions Based on Internal Timers:** It utilizes internal system timers to schedule kernel operations, ensuring they occur at designated times or intervals, which is crucial for system consistency and efficiency.

3. **Provide Execution Profiling Capability:** By collecting data on execution times for kernel and user operations, the handler facilitates the optimization and analysis of overall system performance.

4. **Gather System and Process Accounting Statistics:** It compiles important metrics related to resource usage and process performance, aiding in system monitoring and resource management.

5. **Keep Track of Time:** The clock interrupt handler is essential for maintaining an accurate and reliable account of time, supporting time-dependent operations and scheduling within the operating system.

6. **Send Alarm Signals to Processes on Request:** It manages the timing and dispatch of alarm signals to various processes according to predefined schedules or intervals, ensuring processes act or respond promptly.

7. **Periodically Wake Up the Swapper Process:** The handler ensures the regular activation of the swapper process, which is vital for managing memory efficiently by swapping tasks between active memory and storage.

8. **Control Process Scheduling:** Influences the process scheduling decisions in the operating system by controlling when context switches occur, based on established scheduling policies and priority settings

Explain the System Boot and the Init process.

- The system boot process is a critical sequence of events that initializes an operating system and prepares it for user interaction.
- It involves several stages, starting from the execution of firmware instructions stored in the computer's nonvolatile memory to the initialization of essential system processes.
- **Firmware Execution:** The boot process begins with the execution of firmware instructions stored in the computer's nonvolatile memory, such as BIOS or ROM. These instructions are automatically executed when the power is turned on or the system is reset.
- **Boot Program Execution**: The firmware instructions locate and execute the system's boot program, which is typically stored in a standard location on a bootable device, such as block 0 of the root disk or a special partition. The boot program is responsible for loading the Unix kernel into memory and passing control of the system to it.
- **Kernel Initialization:** Once the kernel is loaded into memory, it initializes various system components, including device drivers, file systems, and process management. At this stage, the kernel sets up essential data structures and prepares the system for user interaction.
- **Init Process Start:** The kernel starts the Init process, also known as initialization. Init is the first user-space process started during the booting process. It continues running until the system is shut down and is responsible for managing other processes.

Roll of init process :

• Init is the parent of all processes, executed by the kernel during the booting of a system. Its principle role is to create processes from a script stored in the file /etc/inittab.

• Init is a daemon process that continues running until the system is shut down. It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes.

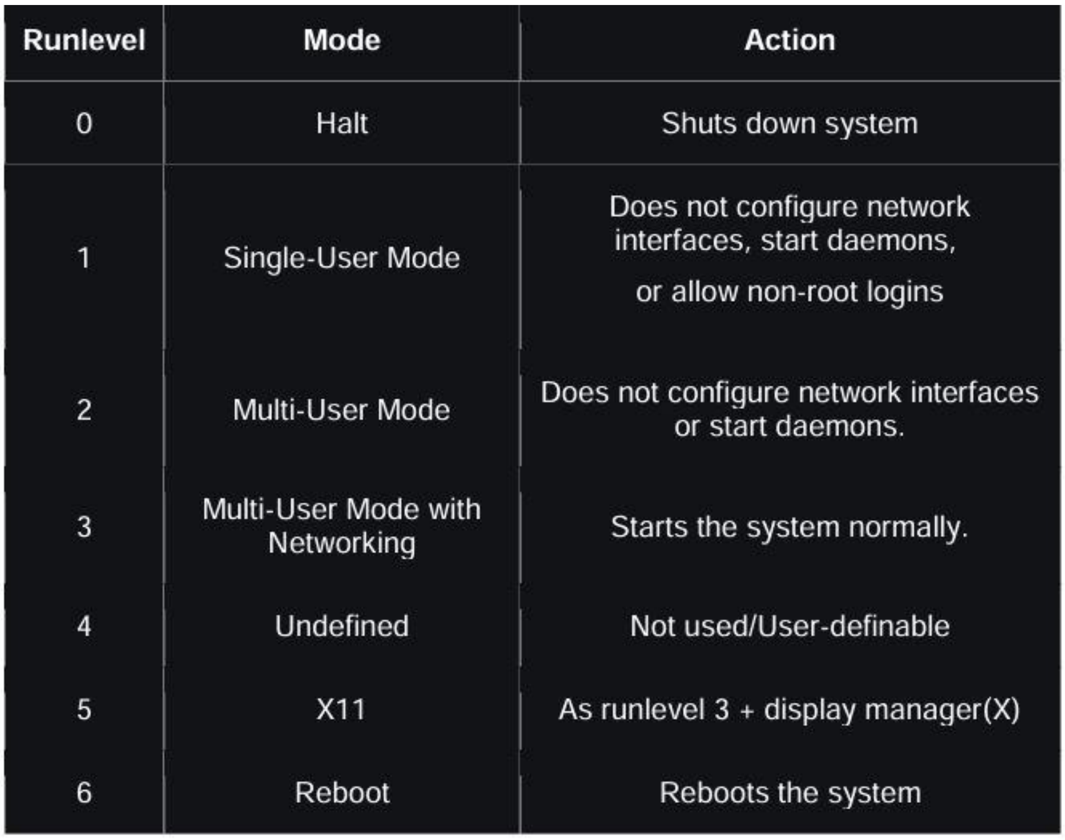
• A kernel panic will occur if the kernel is unable to start it, or it should die for any reason. Init is typically assigned process identifier 1.

• The init process also sets the runlevel, which defines the system’s mode of operation (e.g., single-user mode, multi-user mode, or graphical mode).

Runlevels:

o The runlevel determines which services are started during boot.

- The init process transitions the system to the appropriate runlevel.
- Common runlevels include:



Runlevel	Mode	Action
0	Halt	Shuts down system
1	Single-User Mode	Does not configure network interfaces, start daemons, or allow non-root logins
2	Multi-User Mode	Does not configure network interfaces or start daemons.
3	Multi-User Mode with Networking	Starts the system normally.
4	Undefined	Not used/User-definable
5	X11	As runlevel 3 + display manager(X)
6	Reboot	Reboots the system

What is the use of signal? Explain the types of signals

- In Unix, signals are a mechanism for informing processes of the occurrence of synchronous events.
- These signals serve as notifications sent to a process to perform predefined actions, typically in response to certain events or conditions
- They are essential for efficient process management, resource utilization, and error handling within an operating system.

Types of signals –

1. **Termination Signals**: These are sent when a process exits or terminates, either voluntarily or forced by another process through a system call, such as SIGTERM (soft termination) or SIGKILL (immediate termination).
2. **Exception Signals**: These occur when a process performs an illegal action like accessing unauthorized memory spaces (SIGSEGV) or executing an illegal instruction (SIGILL). These signals help in maintaining system stability by handling exceptions gracefully.
3. **Resource Management Signals**: Sent during instances where system environments reach critical states, such as running out of memory during a process execution (SIGXCPU or SIGXFSZ for exceeding CPU time or file size limits, respectively).
4. **Error Condition Signals**: These signals are generated during anomalous conditions in system calls, often used to abort processes that execute illegal operations that could otherwise lead to undefined states or system security risks.
5. **User-defined or Arbitrary Signals**: Abstract signals like SIGUSR1 and SIGUSR2 are designed for inter-process communication (IPC), allowing processes to communicate or synchronize their states in user-defined ways.
6. **Terminal Interaction Signals**: Signals like SIGHUP (hang-up signal) or SIGINT (interrupt signal from keyboard, like pressing Ctrl+C) directly relate to user interactions with terminal devices, facilitating immediate response to user commands and system status updates.
7. **Tracing and Debugging Signals**: Used primarily in debugging scenarios, signals like SIGTRAP allow debugging tools to intercept and diagnose

process execution flows, which is crucial for developing and maintaining stable software.

When a signal is sent:

- Sets a bit in the process table entry of the receiving process based on the signal type.

If the process is asleep at an interruptible priority:

- The kernel wakes it up.

Processes can remember:

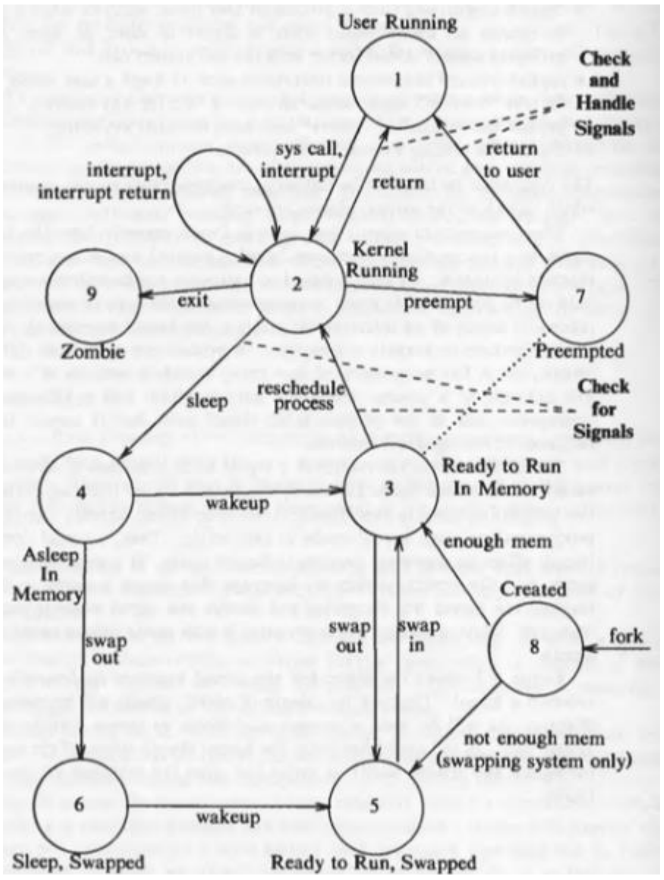
- Different types of signals received.
- But not the number of times a particular type of signal was received.

The kernel checks for signal receipt:

- When a process is about to return from kernel mode to user mode.
- When it enters or leaves the sleep state at a low scheduling priority.

The kernel handles signals:

- Only when a process returns from kernel mode to user mode.



Explain profiling in detail

- Kernel profiling gives a measure of how much time the system is executing in user mode versus kernel mode, and how much time it spends executing individual routines in the kernel.

Kernel Profile Driver:

- This component is responsible for monitoring the performance of various kernel modules by periodically sampling system activity whenever a clock interrupt occurs.
- It maintains a predefined list of kernel addresses that it needs to sample, which typically includes addresses corresponding to kernel functions. These addresses are usually provided to the profile driver by a process in advance.

Profiling Operation:

When kernel profiling is enabled:

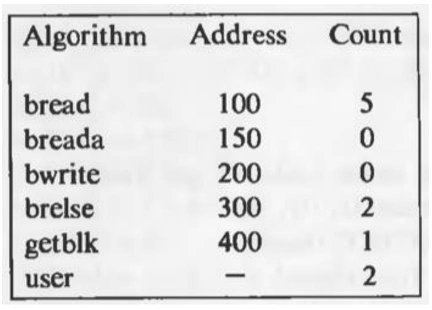
- Upon each clock interrupt, the kernel's interrupt handler triggers the profile driver's interrupt handler.
- The profile driver then checks whether the processor mode at the time of the interrupt was in user mode or kernel mode.
- If the interrupt occurred while the processor was in user mode:

The profile driver increases a count associated with user execution, indicating how much time the system spends executing user-level code.

- If the interrupt occurred while the processor was in kernel mode:

The profile driver increments an internal counter that corresponds to the program counter, providing insights into the execution of kernel code.

For example, the following figure shows hypothetical addresses of several kernel routines:



Algorithm	Address	Count
bread	100	5
breada	150	0
bwrite	200	0
brelse	300	2
getblk	400	1
user	–	2