Submitted by:

Name: Aditi Sudhir Igade

Batch: A3

Roll no.:221081

PRN No.:22220311

Assignment 6

Aim : Case Study and Comparison of XV6 and Linux Operating system.

Theory :

What is XV6 OS?

→ xv6 is a simple Unix-like operating system developed by MIT as a teaching tool for undergraduate students to learn about the internal workings of an operating system. It is based on the Unix version 6 (v6) operating system, which was developed in the 1970s and is known for its simplicity and elegance.

→ xv6 is written in a mix of C and assembly language and provides basic operating system services such as process management, file system, and virtual memory management. It also includes a simple shell and basic user-level programs such as ls, cat, and grep. The source code of xv6 is freely available and has been used by many universities and educational institutions as a teaching tool for operating systems courses.

What is Linux OS ?

→ Linux is a popular open-source operating system that is widely used in various applications, from desktops and servers to mobile devices and embedded systems. It was first developed by Linus Torvalds in 1991 and is based on the Unix operating system.

→ Linux provides a powerful and flexible environment for running applications and services. It has a modular design, with a kernel that provides basic operating system services such as process management, memory management, and input/output (I/O) management. On top of the kernel, Linux provides a wide range of user-level services such as file systems, networking, and security.

→ Linux has become a popular choice for servers and cloud computing due to its stability, security, and scalability. It is also widely used in embedded systems and mobile devices due to its versatility and ability to run on a wide range of hardware architectures. Additionally, Linux provides a vast array of software applications and tools, many of which are open source and freely available, making it a popular choice for developers and users alike.

<u>Comparison :</u>

a) Kernel configuration :

XV6:

xv6 is a simple operating system that is designed for educational purposes. As such, its kernel is relatively small and easy to understand. It has a simple and fixed configuration, with most of the configuration options being hard-coded. This makes it easier for students to learn how an operating system works by looking at the source code.
Xv6 is implemented as a monolithic kernel, like most Unix operating systems. Thus, the xv6 kernel interface corresponds to the operating system interface, and the kernel implements the complete operating system. Since xv6 doesn't provide many services, its kernel is smaller than some microkernels, but conceptually xv6 is monolithic.
A microkernel, the kernel interface consists of a few low-level functions for starting applications, sending messages, accessing device hardware, etc. This organization allows the kernel to be relatively simple, as most of the operating system resides in user-level servers.
Monolithic kernel means the OS resides in the kernel itself. Besides there are two cpu modes the user mode and supervisor mode.

LINUX :

The Linux kernel configuration is usually found in the kernel source in the file: /usr/src/linux/.config .

1. make config - starts a character based questions and answer session
2. make menuconfig - starts a terminal-oriented configuration tool (using ncurses)
3. make xconfig - starts a X based configuration tool

The 3 states of the main selection option for the SCSI subsystem

```
CONFIG_SCSI=y
CONFIG_SCSI=m
# CONFIG_SCSI is not set
```

b) Compilation and rebooting from the newly compiled kernel :

XV6 :

1) We need to make a qemu (The K/kernel is responsible for generating the executable file of the kernel, and running Xv6 is to run the executable file)
2)Then we need to compile the kernel and get the executable file. All the source code of the kernel and the binary files generated after compilation are saved in the. / kernel / directory. OBJS variable is the collection of file names of all. o files required to compile the kernel executable.

3)Getting a linker script. The linker ` ` will link multiple. o files to generate executable files according to the instructions in the script. It mainly describes the method of processing linked files and the content layout of generating kernel executable files.
4)Then comes the giant KERNEL COMPILATION
→Compile the. c and. S source code in the. / kernel directory to obtain the. o object file.
→Link this pile of. o files according to the instructions in the kernel.ld script to get the final kernel executable file kernel.
→The Executable entry is specified as_ Entry, which is defined in the. / kernel/entry.S file.
5)Generation of file system as fs.img
6)Then we compile the user programs.

LINUX :

1)Linux kernel comes in two variants - the "vanilla" kernel, and the distribution's kernel.
2) Sources of linux kernels are available via the Internet, or on the distribution's CDs.
3) After this we need to ready the sources and configure the kernel .
4) Once configured, Compiling the kernel is easy.
→We use make bzImage command after this we should have a file
   Example -rw-r--r--   1 root   root   1064017 Jan 16 01:53 arch/i386/boot/bzIma
→Then we compile kernel modules by make modules command
5) Then we will be installing the kernel by backing up the modules
6) We will be installing modules by using make modules_install
7) Then we will be updating the bootloader (lilo or grub)
8) Once everything is set we need to reboot our machine and boom it run linux now!

c) Scheduling policies :

XV6 :

1. Round-Robin Scheduling: xv6 uses a round-robin scheduling algorithm. Each process is assigned a fixed time slice, known as a quantum or time quantum. The CPU is switched between processes in a circular order, allowing each process to run for a specific amount of time before being preempted.

2. Ready Queue: The scheduler maintains a ready queue that contains all the processes that are ready to run. When a process is created or becomes ready, it is added to the end of the ready queue.

3. Blocked Processes: If a process is blocked, typically due to waiting for I/O or synchronization, it is removed from the ready queue and placed on a sleep queue. The scheduler does not consider blocked processes for execution until the blocking condition is satisfied.

4. Preemption: When a process exhausts its quantum, it is preempted, meaning it is suspended and placed back in the ready queue. The scheduler then selects the next process from the ready queue for execution.

5. Priority Scheduling: xv6 also supports priority scheduling. Each process has a priority value assigned to it, indicating its relative importance. The scheduler maintains multiple priority queues, and it selects the process with the highest priority that is ready to run.

6. Priority Adjustment: The priority of a process can be adjusted dynamically based on factors such as its behavior or resource requirements. This allows for better resource allocation and fairness among processes.

7. Simple and Understandable: The scheduling algorithm in xv6 is relatively simple and straightforward. It is designed this way to make it easier for students and learners to understand the basics of process scheduling in an operating system.

It's important to note that xv6 is primarily an educational operating system, and its scheduling algorithm may not be as sophisticated or optimized as those used in production-grade operating systems like Linux or Windows.

LINUX :

1. Completely Fair Scheduler (CFS): The default scheduler in Linux since version 2.6.23 is the Completely Fair Scheduler (CFS). CFS is a process scheduler that attempts to allocate CPU resources to processes in a "completely fair" manner, meaning that each process is given a fair share of CPU time based on its priority level.

2. Process Priority: Linux assigns each process a priority value, ranging from -20 (highest priority) to 19 (lowest priority). The scheduler uses these priority values to determine which process should be executed next.

3. Time Slicing: The Linux scheduler uses time slicing to ensure that each process gets a fair share of the CPU. Time slicing involves dividing CPU time into equal time slices and allocating each process a slice of CPU time based on its priority level.

4. Preemption: Linux supports both preemptive and non-preemptive scheduling. In preemptive scheduling, a process can be preempted at any time, even if it is not finished executing. Non-preemptive scheduling allows a process to continue executing until it voluntarily relinquishes the CPU.

5. Priority Inheritance: Linux supports priority inheritance, which means that if a high-priority process is blocked waiting for a resource that is being held by a lower-priority process, the lower-priority process is temporarily boosted to the higher priority until it releases the resource.

6. Real-Time Scheduling: Linux also supports real-time scheduling, which is used in applications that require deterministic response times. Real-time scheduling assigns fixed time intervals to processes and ensures that they execute within those intervals.

7. Multiple Queues: The Linux scheduler uses multiple queues to manage processes. Each queue is associated with a specific priority level, and the scheduler selects the process with the highest priority that is ready to run.

Overall, the Linux scheduler is highly configurable and can be customized to suit the needs of different applications. The CFS algorithm is designed to ensure that each process gets a fair share of CPU time, but other scheduling algorithms are available for use if necessary.

d) Memory Management Policies

XV6:

1. Paging: xv6 uses a paging system to manage memory. The system is divided into fixed-size pages, typically 4KB in size. Each process has its own page table, which maps virtual addresses to physical addresses.

2. Demand Paging: xv6 uses demand paging, which means that pages are only loaded into memory when they are actually needed. This helps to conserve memory resources.

3. Kernel Memory: The xv6 kernel has its own page table, which maps kernel virtual addresses to physical addresses. Kernel memory is separate from user memory to protect the kernel from user processes.

4. Memory Allocation: xv6 uses a simple first-fit algorithm to allocate memory to user processes. When a process requests memory, the allocator searches for the first block of free memory that is large enough to satisfy the request.

5. Copy-on-Write: xv6 uses a copy-on-write technique to conserve memory. When a process forks, the child process shares the same physical memory as the parent process. However, when either process attempts to modify the memory, a copy of the page is made to ensure that the other process is not affected.

6. Page Replacement: xv6 uses a simple page replacement algorithm called the "clock" algorithm. When a page needs to be evicted from memory, the algorithm searches for the oldest page that is not currently being used, marks it as "unreferenced," and evicts it from memory.

7. Memory Protection: xv6 uses memory protection mechanisms to ensure that processes cannot access memory that they should not be able to access. This is achieved by setting permissions on page table entries and using hardware support for memory protection.

It's important to note that xv6 is primarily an educational operating system, and its memory management policies may not be as sophisticated or optimized as those used in production-grade operating systems like Linux or Windows.

LINUX:

1. Virtual Memory: Linux uses virtual memory to manage memory. The system is divided into fixed-size pages, typically 4KB in size. Each process has its own virtual address space, which is mapped to physical memory using page tables.

2. Demand Paging: Linux uses demand paging, which means that pages are only loaded into memory when they are actually needed. This helps to conserve memory resources.

3. Kernel Memory: The Linux kernel has its own virtual address space, which is separate from user processes. Kernel memory is protected from user processes to prevent them from accessing or modifying kernel data.

4. Memory Allocation: Linux uses several memory allocation algorithms, including the buddy allocator and the slab allocator. The buddy allocator divides memory into blocks of different sizes and allocates them to processes as needed. The slab allocator is used to allocate objects of a specific size, such as file descriptors or network buffers.

5. Page Replacement: Linux uses a page replacement algorithm called the "page cache" to evict pages from memory. When a page needs to be evicted, the page cache looks for pages that are least recently used and evicts them.

6. Copy-on-Write: Linux uses copy-on-write to conserve memory. When a process forks, the child process initially shares the same memory as the parent process. However, when either process attempts to modify the memory, a copy of the page is made to ensure that the other process is not affected.

7. Memory Protection: Linux uses memory protection mechanisms to prevent processes from accessing memory that they should not be able to access. This is achieved by setting permissions on page table entries and using hardware support for memory protection.

Overall, Linux has a sophisticated memory management system that is designed to optimize performance while conserving memory resources. The system is highly configurable and can be customized to suit the needs of different applications.

e) Interposes communication

Interprocess communication (IPC) is an important feature of any operating system, allowing different processes to exchange data and synchronize their activities.

XV6:

1. Pipes: xv6 supports interprocess communication through pipes, which allow two processes to communicate with each other through a shared buffer. A pipe can be

created using the pipe() system call, which returns two file descriptors: one for the read end of the pipe, and one for the write end.

2. Signals: xv6 supports signals, which are used to notify a process of an event or condition. A signal can be sent to a process using the kill() system call, and a process can register a signal handler function using the signal() system call.

3. Shared Memory: xv6 supports shared memory, which allows two or more processes to share a region of memory. This can be useful for passing large amounts of data between processes without the overhead of copying the data.

4. Message Passing: xv6 does not have built-in support for message passing, but it is possible to implement message passing using a combination of shared memory and semaphores. Semaphores can be used to synchronize access to the shared memory region and ensure that messages are not overwritten or lost.

Overall, interprocess communication in xv6 is relatively simple and limited compared to more advanced operating systems like Linux or Windows. However, it provides enough functionality to support basic IPC needs, and can be extended with additional libraries or system calls if necessary.

LINUX:

1. Pipes: Linux supports interprocess communication through pipes, which allow two processes to communicate with each other through a shared buffer. A pipe can be created using the pipe() system call, which returns two file descriptors: one for the read end of the pipe, and one for the write end.

2. Signals: Linux supports signals, which are used to notify a process of an event or condition. A signal can be sent to a process using the kill() system call, and a process can register a signal handler function using the signal() system call.

3. Shared Memory: Linux supports shared memory, which allows two or more processes to share a region of memory. This can be useful for passing large amounts of data between processes without the overhead of copying the data. Shared memory can be created using the shmget() system call.

4. Message Queues: Linux supports message queues, which allow processes to send and receive messages of a fixed size. Message queues can be created using the msgget() system call.

5. Sockets: Linux supports sockets, which are a mechanism for interprocess communication over a network. Sockets can be used for both interprocess communication on the same machine and for communication over a network.

6. Semaphores: Linux supports semaphores, which can be used to synchronize access to shared resources. Semaphores can be created using the semget() system call.

Overall, Linux provides a rich set of interprocess communication mechanisms, which can be used to implement a wide variety of IPC patterns. The mechanisms are highly configurable and can be customized to suit the needs of different applications. Additionally, Linux provides a wide variety of IPC libraries and frameworks, such as D-Bus and System V IPC, which can simplify the development of complex IPC applications.