**Faculty of Technology and Engineering**

**U & P U. Patel Department of Computer Engineering Assignment – 2**

**CE354 OPERATING SYSTEM**

1. Write a program to implement Banker’s Algorithm for deadlock avoidance.

## ANS: Code:

#include <iostream> #include <vector> using namespace std;

// Function to check if a process can be granted resources

bool canGrantResources(vector<vector<int>>& need, vector<vector<int>>& allocation, vector<int>& available, int process) {

for (int i = 0; i < need[process].size(); ++i) { if (need[process][i] > available[i]) {

return false;

}

}

return true;

}

int main() {

int numProcesses, numResources;

cout << "Enter the number of processes: "; cin >> numProcesses;

cout << "Enter the number of resources: "; cin >> numResources;

vector<vector<int>> maxClaim(numProcesses, vector<int>(numResources)); vector<vector<int>> allocation(numProcesses, vector<int>(numResources)); vector<vector<int>> need(numProcesses, vector<int>(numResources)); vector<int> available(numResources);

// Input maximum resource claims of each process

cout << "Enter the maximum resource claims for each process:" << endl; for (int i = 0; i < numProcesses; ++i) {

cout << "Process " << i + 1 << ": ";

for (int j = 0; j < numResources; ++j) { cin >> maxClaim[i][j];

}

}

// Input currently allocated resources to each process

cout << "Enter the currently allocated resources to each process:" << endl; for (int i = 0; i < numProcesses; ++i) {

cout << "Process " << i + 1 << ": ";

for (int j = 0; j < numResources; ++j) { cin >> allocation[i][j];

need[i][j] = maxClaim[i][j] - allocation[i][j];

}

}

// Input available resources

cout << "Enter the available resources:" << endl; for (int i = 0; i < numResources; ++i) {

cin >> available[i];

}

vector<bool> isProcessFinished(numProcesses, false); vector<int> safeSequence;

int completed = 0;

while (completed < numProcesses) { bool found = false;

for (int i = 0; i < numProcesses; ++i) {

if (!isProcessFinished[i] && canGrantResources(need, allocation, available, i)) { for (int j = 0; j < numResources; ++j) {

available[j] += allocation[i][j];

}

safeSequence.push\_back(i); isProcessFinished[i] = true; completed++;

found = true;

}

}

if (!found) {

cout << "Unsafe state detected! System is in deadlock." << endl; break;

}

}

if (safeSequence.size() == numProcesses) {

cout << "Safe state detected! Safe sequence: "; for (int i = 0; i < safeSequence.size(); ++i) {

cout << "P" << safeSequence[i] + 1; if (i != safeSequence.size() - 1) {

cout << " -> ";

}

}

cout << endl;

}

return 0;

}

1. Write a program to represent the scenario of how deadlock can occur when each thread

holds one resource and waits for the other. (use the concept of IPC)

## ANS: Code:

#include <iostream> #include <thread> #include <mutex>

std::mutex mutex1, mutex2; void threadFunction1() {

mutex1.lock(); // acquire resource 1

std::this\_thread::sleep\_for(std::chrono::milliseconds(10)); // simulate some processing std::cout << "Thread 1 acquired resource 1, waiting for resource 2." << std::endl;

mutex2.lock(); // try to acquire resource 2

std::cout << "Thread 1 acquired resource 2." << std::endl;

mutex2.unlock(); mutex1.unlock();

}

void threadFunction2() { mutex2.lock(); // acquire resource 2

std::this\_thread::sleep\_for(std::chrono::milliseconds(10)); // simulate some processing std::cout << "Thread 2 acquired resource 2, waiting for resource 1." << std::endl;

mutex1.lock(); // try to acquire resource 1

std::cout << "Thread 2 acquired resource 1." << std::endl;

mutex1.unlock(); mutex2.unlock();

}

int main() {

std::thread t1(threadFunction1); std::thread t2(threadFunction2);

t1.join();

t2.join();

return 0;

}

1. Implement a scenario for Inter-process communication through message passing method.

**ANS: Code:** #include <iostream> #include <thread> #include <queue>

#include <mutex>

#include <condition\_variable>

std::mutex mtx; std::condition\_variable cv;

std::queue<std::string> messageQueue;

void sender() {

// Simulate sending a message std::this\_thread::sleep\_for(std::chrono::seconds(2));

// Prepare the message

std::string message = "Hello from Sender!";

// Lock the mutex before modifying the shared queue std::unique\_lock<std::mutex> lock(mtx); messageQueue.push(message);

// Notify the receiver that a message is ready cv.notify\_all();

}

void receiver() {

// Wait for a message from the sender std::unique\_lock<std::mutex> lock(mtx); cv.wait(lock, [] { return !messageQueue.empty(); });

// Once notified, process the message std::string message = messageQueue.front(); messageQueue.pop();

std::cout << "Receiver received message: " << message << std::endl;

}

int main() {

std::thread senderThread(sender); std::thread receiverThread(receiver);

senderThread.join(); receiverThread.join();

return 0;

}

1. Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared Boolean variables S1 and S2 are randomly assigned.

## Method Used by P1

while (S1 == S2) ;

Critica1 Section S1 = S2;

**Method Used by P2** while (S1 != S2) ; Critica1 Section

S2 = not (S1);

## Which one of the following statements describes the properties achieved? Justify your answer.

1. Mutual exclusion but not progress
2. Progress but not mutual exclusion
3. Neither mutual exclusion nor progress
4. Both mutual exclusion and progress

## ANS: The behavior of these methods can be summarized:

1. P1 enters its critical section when S1 and S2 are equal (mutual exclusion).
2. P2 enters its critical section when S1 and S2 are not equal (mutual exclusion).
3. After exiting the critical section, P1 sets S1 equal to S2, and P2 sets S2 equal to the negation of S1.

The possible scenarios:

* Initially, S1 and S2 are randomly assigned, so either P1 or P2 could start in their critical section.

1. If P1 starts in the critical section and sets S1 = S2, then P2 will not be able to enter its critical section because the condition `S1 != S2` will always be false. This ensures mutual exclusion.
2. If P2 starts in the critical section and sets S2 = not(S1), then P1 will not be able to enter its critical section because the condition `S1 == S2` will always be false. This also ensures mutual exclusion.

So, the methods guarantee mutual exclusion because only one process can access its critical section at a time.

* Progress means that if a process is not in its critical section and wants to enter it, it will eventually do so.

In this case, progress is not guaranteed because a situation could arise where both P1 and P2 are stuck in their respective while loops, waiting for the conditions to change, and neither can make progress. For example, if S1 is initially true and S2 is initially false, both P1 and P2 will be stuck in their loops, waiting for the conditions to change, but they won't.

So, the properties achieved by these methods are:

Mutual exclusion but not progress.

Processes can mutually exclude each other from accessing their critical sections, but there is no guarantee of progress because both processes can be stuck waiting for conditions that might not change.

* 1. An operating system uses the banker’s algorithm for deadlock avoidance when managing the allocation of three resource types X, Y and Z to three processes P0, P1 and P2. The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Allocation** | | | **Max** | | |
|  | **X** | **Y** | **Z** | **X** | **Y** | **Z** |
| **P0** | 0 | 0 | 1 | 8 | 4 | 3 |
| **P1** | 3 | 2 | 0 | 6 | 2 | 0 |
| **P2** | 2 | 1 | 1 | 3 | 3 | 3 |

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in safe state. Consider the following independent requests for additional resources in the current state-

REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z Which of the following is TRUE? Give the detailed solution.

* + 1. Only REQ1 can be permitted
    2. Only REQ2 can be permitted
    3. Both REQ1 and REQ2 can be permitted
    4. Neither REQ1 nor REQ2 can be permitted

## ANS: C) Both REQ1 and REQ2 can be permitted

Available Resources (Available):

Available = (3, 2, 2)

REQ1: P0 requests 0 units of X, 0 units of Y, and 2 units of Z

1. Check if REQ1 can be granted without leading to an unsafe state:

- Subtract REQ1 from the Available Resources:

Available = Available - REQ1 = (3, 2, 2) - (0, 0, 2) = (3, 2, 0)

1. Check if the resulting state is safe. To do this, perform a safety check by simulating the allocation of resources to the processes:

* Allocate resources to P0 using the available resources: Available >= Max[P0] Available >= Max[P0]

(3, 2, 0) >= (8, 4, 3) [Max for P0]

This condition is not met, so P0 cannot proceed.

* Now, allocate resources to P1 using the available resources: Available >= Max[P1] Available >= Max[P1]

(3, 2, 0) >= (6, 2, 0) [Max for P1]

This condition is met, so P1 can proceed.

* + P1 is the only process that can proceed. Let's allocate resources to P1 and update the available resources:

Allocated = A[P1] = (3, 2, 0)

Available = Available + Allocated = (3, 2, 0) + (3, 2, 0) = (6, 4, 0)

* Now, allocate resources to P2 using the available resources: Available >= Max[P2] Available >= Max[P2]

(6, 4, 0) >= (3, 3, 3) [Max for P2]

This condition is met, so P2 can proceed.

* + P2 is the only process that can proceed now. Allocate resources to P2 and update the available resources:

Allocated = A[P2] = (2, 1, 1)

Available = Available + Allocated = (6, 4, 0) + (2, 1, 1) = (8, 5, 1)

Now, all processes have completed, and the system is in a safe state. Therefore, REQ1 can be permitted. REQ2: P1 requests 2 units of X, 0 units of Y, and 0 units of Z

1. Check if REQ2 can be granted without leading to an unsafe state:

- Subtract REQ2 from the Available Resources:

Available = Available - REQ2 = (8, 5, 1) - (2, 0, 0) = (6, 5, 1)

1. Now, check if the resulting state is safe. To do this, perform a safety check by simulating the allocation of resources to the processes:

* Try to allocate resources to P0 using the available resources: Available >= Max[P0] Available >= Max[P0]

(6, 5, 1) >= (8, 4, 3) [Max for P0]

This condition is not met, so P0 cannot proceed.

* Try to allocate resources to P1 using the available resources: Available >= Max[P1] Available >= Max[P1]

(6, 5, 1) >= (6, 2, 0) [Max for P1]

This condition is met, so P1 can proceed.

* Allocate resources to P1 and update the available resources:

Allocated = A[P1] = (3, 2, 0)

Available = Available + Allocated = (6, 5, 1) + (3, 2, 0) = (9, 7, 1)

* Now, allocate resources to P2 using the available resources: Available >= Max[P2] Available >= Max[P2]

(9, 7, 1) >= (3, 3, 3) [Max for P2]

This condition is met, so P2 can proceed.

* Allocate resources to P2 and update the available resources:

Allocated = A[P2] = (2, 1, 1)

Available = Available + Allocated = (9, 7, 1) + (2, 1, 1) = (11, 8, 2)

Now, all processes have completed, and the system is in a safe state. Therefore, REQ2 can also be permitted.

So, the correct answer is:

Both REQ1 and REQ2 can be permitted.

* 1. Write a menu driven program to implement following disk scheduling algorithm:
     1. FCFS
     2. SSTF
     3. SCAN
     4. C-SCAN
     5. LOOK
     6. C-LOOK

## ANS: Code:

#include <iostream> #include <vector> #include <algorithm> #include <cmath>

using namespace std;

void FCFS(vector<int> requests, int start) { int headMovement = 0;

int size = requests.size();

cout << "FCFS Order: " << start; for (int i = 0; i < size; i++) {

cout << " -> " << requests[i]; headMovement += abs(start - requests[i]); start = requests[i];

}

cout << "\nTotal head movements: " << headMovement << endl;

}

void SSTF(vector<int> requests, int start) { int size = requests.size();

int headMovement = 0; vector<bool> visited(size, false);

cout << "SSTF Order: " << start; for (int i = 0; i < size; i++) {

int closest = -1;

for (int j = 0; j < size; j++) { if (!visited[j]) {

if (closest == -1 || abs(start - requests[j]) < abs(start - requests[closest])) { closest = j;

}

}

}

visited[closest] = true;

cout << " -> " << requests[closest]; headMovement += abs(start - requests[closest]); start = requests[closest];

}

cout << "\nTotal head movements: " << headMovement << endl;

}

void SCAN(vector<int> requests, int start, int maxVal) { int size = requests.size();

int headMovement = 0; sort(requests.begin(), requests.end());

vector<int>::iterator it = lower\_bound(requests.begin(), requests.end(), start); int idx = it - requests.begin();

cout << "SCAN Order: " << start; for (int i = idx; i < size; i++) {

cout << " -> " << requests[i]; headMovement += abs(requests[i] - start); start = requests[i];

}

if (idx < size - 1) {

cout << " -> " << maxVal; headMovement += abs(maxVal - start); start = maxVal;

}

for (int i = idx - 1; i >= 0; i--) { cout << " -> " << requests[i];

headMovement += abs(requests[i] - start); start = requests[i];

}

cout << "\nTotal head movements: " << headMovement << endl;

}

void CSCAN(vector<int> requests, int start, int maxVal) { int size = requests.size();

int headMovement = 0; sort(requests.begin(), requests.end());

vector<int>::iterator it = lower\_bound(requests.begin(), requests.end(), start); int idx = it - requests.begin();

cout << "C-SCAN Order: " << start; for (int i = idx; i < size; i++) {

cout << " -> " << requests[i]; headMovement += abs(requests[i] - start); start = requests[i];

}

cout << " -> " << maxVal; headMovement += abs(maxVal - start); start = 0; // back to 0

cout << " -> " << start; for (int i = 0; i < idx; i++) {

cout << " -> " << requests[i]; headMovement += abs(requests[i] - start); start = requests[i];

}

cout << "\nTotal head movements: " << headMovement << endl;

}

void LOOK(vector<int> requests, int start) { int size = requests.size();

int headMovement = 0; sort(requests.begin(), requests.end());

vector<int>::iterator it = lower\_bound(requests.begin(), requests.end(), start); int idx = it - requests.begin();

cout << "LOOK Order: " << start; for (int i = idx; i < size; i++) {

cout << " -> " << requests[i]; headMovement += abs(requests[i] - start); start = requests[i];

}

for (int i = idx - 1; i >= 0; i--) { cout << " -> " << requests[i];

headMovement += abs(requests[i] - start); start = requests[i];

}

cout << "\nTotal head movements: " << headMovement << endl;

}

void CLOOK(vector<int> requests, int start) { int size = requests.size();

int headMovement = 0; sort(requests.begin(), requests.end());

vector<int>::iterator it = lower\_bound(requests.begin(), requests.end(), start); int idx = it - requests.begin();

cout << "C-LOOK Order: " << start; for (int i = idx; i < size; i++) {

cout << " -> " << requests[i]; headMovement += abs(requests[i] - start); start = requests[i];

}

for (int i = 0; i < idx; i++) {

cout << " -> " << requests[i]; headMovement += abs(requests[i] - start); start = requests[i];

}

cout << "\nTotal head movements: " << headMovement << endl;

}

int main() { int choice; int start;

int maxVal = 200; // Assuming disk size vector<int> requests;

cout << "Enter starting position of disk head: "; cin >> start;

cout << "Enter number of disk locations: "; int n;

cin >> n;

cout << "Enter requests: "; for (int i = 0; i < n; i++) {

int x; cin >> x;

requests.push\_back(x);

}

do {

cout << "\nDisk Scheduling Algorithms: "; cout << "\n1. FCFS";

cout << "\n2. SSTF"; cout << "\n3. SCAN"; cout << "\n4. C-SCAN"; cout << "\n5. LOOK"; cout << "\n6. C-LOOK"; cout << "\n7. Exit";

cout << "\n\nEnter your choice: "; cin >> choice;

switch (choice) { case 1:

FCFS(requests, start); break;

case 2:

SSTF(requests, start); break;

case 3:

SCAN(requests, start, maxVal); break;

case 4:

CSCAN(requests, start, maxVal); break;

case 5:

LOOK(requests, start); break;

case 6:

CLOOK(requests, start); break;

case 7:

cout << "Exiting..." << endl; break;

default:

cout << "Invalid choice!" << endl; break;

}

} while (choice != 7);

return 0;

}

* 1. Explain EXT file system in Linux.

**ANS:** The EXT file system, short for Extended File System, is one of the oldest and most widely used file systems in the Linux operating system and its variants (like Android). There have been several versions of the EXT file system over the years, including EXT2, EXT3, and EXT4, each offering improvements and

additional features. As of my last knowledge update in September 2021, EXT4 is the most commonly used version.

1. EXT2 (Second Extended File System): This was the first file system in the EXT series and was introduced in 1993. It provided basic file system functionality with features like hierarchical directory structure, file permissions, and support for large file sizes and partitions. However, it lacked features such as journaling, which meant that in case of a system crash, file system consistency had to be checked and repaired manually.
2. EXT3 (Third Extended File System): EXT3, introduced in 2001, was an evolution of EXT2 with the addition of journaling. Journaling helps maintain file system integrity by logging changes before they are committed, making it easier and faster to recover from crashes. EXT3 is backward-compatible with EXT2, meaning you can mount an EXT2 file system as an EXT3 file system.
3. EXT4 (Fourth Extended File System): EXT4, introduced in 2008, is the most commonly used version of the EXT file system in modern Linux distributions. It includes enhancements over EXT3, such as support for larger file sizes and partitions, better performance, and improved storage allocation. EXT4 also supports delayed allocation, which improves write performance, and extends the maximum file system size significantly.

Key features of EXT4 include:

- Journaling: Like EXT3, EXT4 supports journaling, which helps maintain file system consistency in the event of crashes or power failures

8. Explain NTFS file system.

**ANS:** NTFS, which stands for New Technology File System, is a file system primarily used in Microsoft Windows operating systems, including Windows NT, 2000, XP, Vista, 7, 8, 10, and newer versions as of my last knowledge update in September 2021. It is one of the most common file systems for

Windows-based computers and offers several advanced features compared to older file systems like FAT (File Allocation Table).

1. File and Disk Management:

* File Security: NTFS provides a robust security model with file and folder permissions, allowing administrators to control who can access, modify, or delete files and directories. This is essential for maintaining the security and integrity of data on Windows-based systems.
* Compression: NTFS supports file and folder compression, which can help save disk space by reducing the size of stored data. However, compressed files require more CPU resources to read and write.
* Encryption: NTFS includes built-in encryption capabilities, such as Encrypting File System (EFS), which allows users to encrypt specific files and folders to protect sensitive data.
* Quotas: Administrators can set disk space quotas for users or groups to limit how much data they can store on a particular volume. This helps manage disk space more effectively in shared environments.

1. Reliability and Fault Tolerance:

* Journaling: NTFS uses a journaling system that logs changes before they are committed to the file system. This helps ensure file system consistency in case of system crashes or unexpected power losses.
* File System Recovery: NTFS includes tools for checking and repairing the file system in the event of corruption or other issues. Chkdsk (Check Disk) is one such utility that can scan and repair NTFS volumes.
* Redundant Copies: NTFS can store multiple copies of important system data structures on the disk to provide fault tolerance. This helps recover critical file system metadata in case of disk errors.

1. File System Features:

* Long Filenames: NTFS supports long filenames (up to 255 characters), which is an improvement over the 8.3 filename limit in older file systems like FAT.
* Sparse Files: NTFS allows the creation of sparse files, which are files that can have large sections of empty data. This is useful for applications that work with large datasets.
* Symbolic Links: NTFS supports symbolic links (symlinks), which are references to files or directories that can be used to create shortcuts or references to other locations in the file system.

1. Large Volume and File Support:

- NTFS can handle very large volumes and files, making it suitable for modern storage needs. It supports volumes up to 256 terabytes in size and individual files up to 16 exabytes.

1. Compatibility: While NTFS is primarily associated with Windows, it is not natively supported by other operating systems like Linux and macOS. However, there are third-party drivers and utilities that allow limited NTFS support on these platforms.

Overall, NTFS is a powerful and feature-rich file system designed to meet the demands of modern computing environments, particularly in Windows-based systems.

# 9. Explain Memory Management in XV6 operating system.

**ANS:** Xv6 is a lightweight, educational operating system developed at MIT as a reimplementation of the Unix Version 6 (V6) operating system. Memory management in Xv6 is an essential aspect of the operating system, responsible for managing the allocation and deallocation of memory resources in a way that ensures efficient and secure operation.

1. Physical Memory Organization:

- Xv6 operates in a simple memory model where physical memory is divided into different regions, including the kernel memory and user memory. The kernel memory is reserved for the operating system, while the user memory is for user processes.

1. Page Tables:

- Xv6 uses a two-level page table structure to manage virtual memory. The top-level page table is called the Page Directory, and the second-level page tables are called Page Tables. This two-level hierarchy helps in efficiently mapping virtual addresses to physical addresses.

1. Page Faults:

- When a program running in user mode accesses a memory location that is not in physical memory (i.e., a page fault occurs), the Xv6 operating system handles the page fault. It may involve loading the required page from secondary storage (such as a disk) into physical memory.

1. Kernel Memory Mapping:

- The Xv6 kernel is mapped into the address space of each user process. This allows user processes to invoke system calls and interact with the kernel. However, only a portion of the kernel memory is mapped into each process's address space, ensuring that user processes cannot directly access kernel data structures.

1. Process Memory Layout:

* In Xv6, each user process has its own virtual address space, which includes code, data, and stack segments. The layout of this address space is typically organized as follows:
  + Text Segment: Contains the executable code of the program.
  + Data Segment: Contains initialized global and static data.
  + BSS Segment: Contains uninitialized global and static data.
  + Heap: Used for dynamic memory allocation (e.g., via `malloc`).
  + Stack: Used for function call and local variable storage.

1. Memory Allocation:

- Xv6 provides memory allocation mechanisms for user processes. The `sbrk` system call is used to adjust the program's heap size by allocating or deallocating memory pages.

1. Memory Protection:

- Xv6 enforces memory protection to prevent user processes from accessing kernel memory or the memory of other processes. It uses hardware features such as segmentation (if available) and

page-level protection bits.

1. Shared Memory:

- Xv6 supports inter-process communication (IPC) through shared memory regions. Processes can create shared memory segments and communicate by reading/writing to these shared regions.

1. Memory Deallocation:

- When a process exits or explicitly releases memory, Xv6 deallocates the associated memory pages and updates the page tables accordingly.

1. Swapping:

- Xv6 may implement a basic swapping mechanism to move pages between physical memory and disk when memory becomes scarce. This helps in efficiently managing memory resources.

It's important to note that Xv6 is designed as a teaching tool and may not include all the sophisticated memory management features found in production operating systems. However, it provides a solid foundation for understanding the principles of memory management in operating systems.

# 10. Explain Process creation and Process scheduling in XV6 operating system

**ANS:** In the XV6 operating system, process creation and process scheduling are fundamental aspects of managing and executing tasks on a computer.

Process Creation in XV6:

1. Fork System Call: In XV6, the primary mechanism for creating a new process is through the `fork` system call. When a process calls `fork`, the operating system creates a new process that is a copy of the calling process. This new process is referred to as the child process.
2. Copying the Address Space: During the `fork` operation, XV6 duplicates the entire address space of the parent process, including code, data, stack, and heap segments. Both the parent and child processes share the same program, but they have separate memory spaces.
3. Setting Up the Child Process: After copying the address space, XV6 initializes the child process's state, including its registers and program counter, to make it ready for execution.
4. Return Values: In the parent process, the `fork` system call returns the child's process ID (PID), while in the child process, it returns 0. This allows the parent and child processes to distinguish themselves.

Process Scheduling in XV6:

XV6 uses a simple round-robin scheduling algorithm, where each process is given a fixed time slice (quantum) to execute before the operating system switches to the next process. Here's how process scheduling works in XV6:

1. Scheduler Initialization: When XV6 boots, it initializes the scheduler, setting up data structures to manage processes. The process table stores information about each active process, including its state, program counter, stack pointer, and more.
2. Process States: XV6 processes can be in several states, including:

* UNUSED: The process table entry is not in use.
* EMBRYO: The process is in the process of being created.
* SLEEPING: The process is waiting for an event, such as I/O completion or a timer.
* RUNNABLE: The process is ready to execute but is not currently running.
* RUNNING: The process is currently executing on the CPU.

1. Context Switching: The XV6 scheduler performs context switching to switch between processes. When a process's time slice expires, or it voluntarily yields the CPU, the scheduler saves the current process's state (registers, program counter, etc.) and restores the state of the next process to run.
2. Time Sharing: Each process is allocated a fixed time quantum to run on the CPU. Once a process's quantum expires, it is moved to the end of the run queue, and the next process in the queue is scheduled to run. This round-robin scheduling ensures fairness in CPU allocation among active processes.
3. Blocking and Waking: When a process needs to wait for some event (e.g., I/O operation), it is marked as "SLEEPING," and the scheduler selects the next runnable process. When the event occurs, the process is marked as "RUNNABLE" and placed back in the run queue.
4. Process Termination: When a process completes its execution or is terminated (e.g., by calling `exit`), it is marked as "ZOMBIE" and remains in the process table until its parent retrieves its exit status using the `wait` system call. Afterward, the process entry is marked as "UNUSED" and can be reused.
5. Idle Process: XV6 also includes an "idle" process, which runs when there are no other runnable processes. This ensures that the CPU is never left idle.

In summary, XV6 employs a straightforward round-robin scheduling algorithm to manage process execution, and it uses the `fork` system call to create new processes.