Operating Systems - Review Notes

Aditya Soni

Computer Science and Engineering, NIT Silchar

Reference: Operating System Concepts (10th ed.)
By Silberschatz A, Galvin PB, Gagne G

Contents

Chapter 1

Introduction

Operating System is a system software that manages computer hardware, software resources and provides common services to computer programs.

1.1 Types of Operating Systems

(1) **Batch OS**: Users do not interact with the computer directly. Each user submits job to the computer operator, then jobs with similar needs are batched together and run as a group. The objective is to maximise processor use. Major drawbacks include no interaction between user and computer, and no way to set priority for a particular request.

(2) **Time-Sharing OS**: Works by allocating time to a particular task and switching between tasks frequently. The system provide access to large number of users simultaneously. The objective is to minimize response time. Major drawbacks include reliability issues, and security and integrity of programs and data.

(3) **Distributed OS**: It is based on autonomous but interconnected computers communicating via lines or network. Each autonomous system has its own processor. This kind of OS serves multiple applications and multiple users, provides remote working, faster exchange of data, and minimizes load on host computers. Major drawbacks include cost to install and maintain, and primary network fail results in entire system shutdown.

(4) **Real-Time OS**: provides support to real-time systems that require observance of strict time requirements. The response time between input, processing and response is tiny, which is beneficial for process that are highly sensitive and need high precision. RTOS can be categorized into:

   (a) Hard RTOS guarentee that critical tasks complete on time. Virtual memory is almost never found in these systems.

   (b) Soft RTOS, where a critical real-time task takes priority over other tasks and retains the priority until completion.

(5) **Mobile OS**: These systems include PDA (personal digital assisstant) with connectivity to cellular network. Memory needs to be manages effectively due to small storage capacity. This OS does not support swapping so *memory compression* is used where several frames are compressed into single frame and then decomposed as and when needed, thus reducing memory usage of system.

Chapter 2

Process Management

2.0.1 Process

Process is a program in execution. As process executes, it changes states which are defined by current activity of the process. A process may be in one of the following states:

- New: the process is being created.
- Running: instructions are being executed.
- Waiting: process is waiting for some event to occur.
- Ready: process is waiting to be assigned to a processor.
- Terminated: finished execution

Each process is represented in the operating system by a **process control block** (PCB), which contains information about specific process including:

- Process state
- Program counter (indicating address of next instruction)
- CPU registers (state information of the registers is stored)
- CPU scheduling information
- Memory management information
- I/O states (list of devices allocated to the process)

2.0.2 Threads

Thread is a basic unit of CPU utilization, and comprises of thread id, program counter, register set and a stack. Threads belonging to the same process share code section, data section and other OS resources such as files and signals.

Multithreading means concurrently executing multiple threads of a process. It has several benefits:

a) Responsiveness: allows program to continue running even if part of it is blocked or performing lengthy operation (like I/O can be done simultaneously)

b) Resource-sharing: Threads share memory and resources of the process to which they belong, so multiple activities can be performed within the same address space.

c) Economy: Allocating memory and resources is costly, so its more efficient to create and context-switch threads.

d) Scalability: even better performance of multiprocessor architectures where threads can run in parallel.

2.1 CPU Scheduling

CPU-I/O Burst Cycle Processes alternate between CPU execution and I/O wait. An **I/O bound** program typically has many short CPU bursts whereas a **CPU-bound** program might have few long CPU bursts.

CPU Scheduler Whenever the CPU becomes idle, OS must select one of the processes in the ready queue to be executed, which is done by CPU scheduler.

- **Preemptive Scheduling**: CPU is given to a process based on certain priority, and resources are taken away by context-switch once the process goes down in priority or when the process terminates. Preemptive scheduling can result in race conditions when data is shared among several processes. A preemptive kernel requires mechanisms such as mutex locks to prevent race conditioins when accessing shared kernel data structures.

- **Non-Preemptive Scheduling**: CPU is given to a process, and the resources are released voluntarily by the process on termination or during I/O wait.

Dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. The time it takes to stop one process and start another running is known as *dispatch latency*.

2.1.1 Scheduling Criteria

The criteria for comparison and efficiency of scheduling algorithms are:

(1) **CPU Utilization**: keeping CPU as busy as possible

(2) **Throughput**: number of processes that are completed in certain time duration.

(3) **Turnaround Time**: sum of time periods spent waiting in the ready queue, execution time and I/O bursts.

(4) **Waiting Time**: sum of time periods spent waiting in the ready queue.

(5) **Response Time**: time taken by the process to start responding i.e. time duration from submission of request until the first response is produced.

Note: It is desirable to maximise CPU utilization and throughput and to minimize turnaround time, waiting time and response time.

2.2 Scheduling Algorithms

2.2.1 First-Come-First-Serve (FCFS)

The process that requests the CPU first is allocated the CPU first. This is easily implemented using FIFO queue. FCFS is a non-preemptive algorithm so the average waiting time varies substancially if CPU bursts vary. All processes must wait for their turn to use the CPU.

2.2.2 Shortest Job First (SJF)

Here, length of the process's next CPU burst is assosiated with the process. When the CPU is available, it is assigned to process that has smallest next CPU burst. If its same for two processes, the FCFS is used to break the tie. SJF decreases average scheduling time as compared to FCFS. Clearly, there is no real way to predict the length of next CPU burst, but exponential average can be used for the purpose: $\tau_{n+1} = \alpha\tau_n + (1 - \alpha)\tau_n$

2.2.3 Round-Robin Scheduling

The ready queue is treated as a circular queue and a small unit of time, called *time quantum* is defined. CPU scheduler goes around allocating the CPU to each process for a time interval of upto 1 time quantum. Either process will have CPU burst of less than 1 time quantum, so it will release the CPU voluntarily and CPU will move to next process, or else the process will be interrupted and CPU will move on to the next process in the queue through context-switching. This algorithm is *preemptive*. The performance heavily depends on the size of time quantum. If its too large, RR will be same as FCFS or if its too small, number of context switches will be large. Generally, time quantum is chosen such that 80% CPU bursts should be shorter than time quantum.

2.2.4 Priority Scheduling

A priority is assosiated with each process, and the CPU is allocated to the process with the highest priority. SJF is special case of priority scheduling where priority is inverse of the CPU burst time. This algorithm can be either preemptive or non-preemptive. A major problem with priority scheduling is *infinite blocking* or *starvation* i.e. a process that is ready to run but never get hold of CPU due to low priority. A solution to starvation is *aging*, where priority of a process is gradually increased with the duration it spends waiting.

2.2.5 Multilevel-Queue Scheduling

If all processes are placed in a single queue, an $O(n)$ search may be necessary to find the highest priority process. However, in multilevel queue, separate queues are used for each distinct priority and scheduler simply schedules the highest priority queue. Processes can be classified into queues based on process types, for example, (1)Real-time processes, (2)System processes, (3)Interactive processes, and (4)Batch processes (here, each queue has absolute priority over low-priority queues i.e. if an interactive process enters while batch process is executing, the batch process will be preempted). Also, processes can be scheduled inside each queue using different algorithms.

2.2.6 Multilevel-Feedback Queue Scheduling

This algorithm allows a process to move between queues. The idea is the separate processes according to their CPU bursts. If a process using too much CPU time, it will be moved to a low priority queue. This scheme leaves I/O bound and interactive processes which have short CPU bursts in high priority queue. Also, a process that waits too long in a low priority queue may be moved to high priority queue

to prevent starvation. For example, an entering process is put in queue 0, where its given a time of 8ms. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at head of queue 1 is given 16ms. If it doesn't complete, it is preempted and put in queue 2, where FCFS is used when queue 0 and queue 1 are empty. Here, processes less than 24ms of burst time are executed quickly. However, starvation must be prevented for low-priority queue.

2.3 Multiprocessor Scheduling

If multiple CPUs are available, load sharing, where multiple threads may run in parallel, becomes possible. Multiprocessor term can be used for following architecture:

- Multicore CPUs
- Multithreaded cores
- NUMA systems (Non-uniform memory access)
- Heterogenous multiprocessing

2.3.1 Asymmetric multiprocessing

Here, all scheduling decisions, I/O processing, and other system activities are handled by a single processor, the master server. The other processors only execute user code. This is simple as only one core accesses system data structures, thus reducing the need for data sharing. The downfall is that the master server becomes a potential bottleneck where overall system performance may be reduced.

2.3.2 Symmetric multiprocessing

This is the standard approach where each process is self-scheduling. Scheduling proceeds by having scheduler for each processor examine the ready queue and select a thread to run. There are two possible strategies: (i)All threads may be in a common ready queue, and (ii)Each processor have its own ready queue. The second option permits each processor to schedule threads from its private run queue so its the most common approach on SMP systems. Also, having private per-processor run queues in fact may lead to more efficient use of code memory. Issue with this is workloads of varying sizes can cause more load on certain processor but balancing algorithm can be used to equalize workload among all processors.

Load Balancing attempts to keep workload evenly distributed across all processors in SMP systems. Generally, two approaches are used:

- Push Migration: a specific task periodically checks load on each processor and if it finds imbalance it evenly distributes loads from overloaded to idle or less busy processor.
- Pull Migration: when an idle processor pulls a waiting task from a busy processor.

Completely Fair Scheduler (CFS) became the default scheduling algorithm for Linux systems.

Chapter 3

Process Synchronization

System consists of several threads running either concurrently or in parallel. Threads often share user data. Meanwhile, OS continuosly updates various data structures to support multiple threads. A race condition exists when access to shared data is not controlled, possibly resulting in corrupt data values.

Race condition is the situation where several processes access and manipulate the same data concurrently and the outcome of execution depends on the particular order in which the access takes place.

3.1 Classic Problems of Synchronization

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem

Note: Further reading suggested on these.

3.2 Critical Section Problem

Consider a system of n processes $\{P_0, P_1, P_2, ..., P_n\}$. Each process has segment of code, called *critical section*, in which process may be accessing and/or updating data that is shared with at least one other process. The important feature is, when one process is executing in its critical section, no other process is allowed to execute in its critical section. A protocol is needed so that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its critical section. The section of code implementing this request is *entry section*. The critical section may be followed by an exit section. The remaining code is the *remainder section*.

A solution to critical-section problem must satisfy the following requirements:

(1) Mutual Exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

(2) Progress: It is ensured that processes will cooperatively determine what process will next enter its critical section, and this selection cannot be postponed indefinitely.

(3) Bounded Waiting: Limiting how much time a program will wait before it can enter its critical section.

Software solutions to the critical-section problem, such as Peterson's solution, do not work well on modern computer architectures, the reason being, to improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies. Hardware support includes hardware instructions, such as compare-and-swap, and atomic variables.

3.3 Peterson's Solution

This classic software-based solution is restricted to two processes that alternate between their critical sections and remainder sections. The processes are P_0 and P_1, but for convinience when presenting, P_i and P_j are used. This solution requires two data items:

```
int turn;           // indicates whose turn it
                    // is to enter critical section
boolean flag[2];    // indicates if the process is ready
                    // to enter its critical section
```

The structure of process P_i in Peterson's solution looks like:

```
do {
    flag[i] = true;
    turn = j;
    while(flag[i] && turn==j);

    /* Critical Section */

    flag[i] = false;

    /*Remainder Section*/

} while(true);
```

What's happening here: To enter the critical section, process P_i first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

To prove that this solution is correct, the points that need to be proven are:

- Mutual Exclusion is preserved: P_i can enter its critical section only if either flag[j]==false or turn==i. Even if flag[0]==flag[1]==true, both processes can't enter critical section as turn can only be either i or j and not both. flag[j]==true and turn==j will persist as long as P_j is in its critical section.

- Progress requirement is satisfied and Bounded-waiting requirement is met: P_i can only be prevented from entering its critical section if its stuck in the while loop i.e. flag[j]==true and turn==j. If P_j is not ready, then flag[j]==false and P_i can enter critical section. When P_j resets flag[j] to true, it must also set turn to i, and since P_i does not change value of turn in 'while' statement, P_i will enter critical section (progress) after at most one entry by P_j (bounded waiting).

3.4 Hardware Support for Synchronization

Hardware Instructions of certain kinds, allows testing and modifying the content of word or to swap contents of two words *atomically* i.e. as one uninterruptible unit.

3.4.1 test_and_set()

This instruction is defined atomically (i.e. if two of these instructions are executed simultaneously, they will be executed sequentially in some arbitrary order) as:

```
boolean test_and_set (boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

Mutual exclusion is implemented as follows (structure of process P_i):

```
do {
    while (test_and_set(&lock))
        ;       /* do nothing */

    /* Critical Section */

    lock = false;

    /* Remainder Section */

} while (true);
```

3.4.2 compare_and_swap()

This instruction uses swapping content of two words, and is defined atomically as:

```
int compare_and_swap (int *value, int expected,
        int new_value) {

    int temp = *value;
    if(*value == expected)
        *value = new_value;

    return temp;
}
```

Mutual exclusion is implemented as follows (structure of process P_i):

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ;       /* do nothing */

    /* Critical Section */
```

```
        lock = 0;

        /* Remainder Section */

    } while (true);
```

Although the above algorithm satisfies mutual exclusion requirement, bounded-waiting requirement is not fulfilled. Here is an algorithm which satisfies all critical section conditions:

```
do {
        waiting[i] = true;
        key = 1;
        while (waiting[i] && key == 1)
            key = compare_and_swap (&lock, 0, 1);
        waiting[i] = false;

        /* Critical Section */

        j = (i+1) % n;
        while ((j!=i) && !waiting[j])
            j = (j + 1) % n;

        if (j == i)
            lock = 0;
        else
            waiting[j] = false;

        /* Remainder Section */

    } while (true);
```

Common data structures are:

```
boolean waiting[n]; // array initialized to false
int lock;           // initialized to 0
```

Here,

- The key can become 0 only through compare_and_swap() when lock==0. waiting[i] is set to false only if another process leaves its critical section, and only one waiting[i] is set to false, ensuring mutual exclusion.
- Progress requirement is met as process exiting the critical section can either set lock to 0 or set waiting[j] to false allowing other process to enter critical section.
- Bounded-waiting requirement is met as process leaving critical section checks array in cyclic order to choose process which has waiting[j] set to true. So any process waiting to enter its critical section will do so in (n-1) turns.

3.5 Mutex Locks

One of the simplest higher-level software tool is mutex lock. A process must acquire the lock before entering critical section, and releases the lock when it exits critical section. It has a boolean variable *available* whose value indicates if the lock is available or not. A process that attempts to acquire an unavailable lock is blocked until the lock is released. Calls to *acquire()* and *release()* are performed atomically.

acquire() is defined as:

```
acquire() {
    while (!available)
        ;    /* busy wait */
    available = false;
}
```

release() is defined as:

```
release() {
    available = true;
}
```

Busy Waiting While a process is in its critical section, other processes must loop continuosly to call acquire lock, thus wasting CPU cycles. Hence, these types of locks are called spinlocks. Busy waiting can be prevented by using sleep for waiting processes or using semaphores to block processes and place them on waiting queue.

3.6 Semaphores

A semaphore S, is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

wait() is defined as:

```
wait (S) {
    while (S <= 0);       // busy wait
    S--;
}
```

signal() is defined as:

```
signal (S) {
    S++;
}
```

Types of Semaphores

- Counting Semaphore: can be used to control access to a given resource consisting of finite number of instances. The semaphore is initialized to the number of resource instances available.
- Binary Semaphore: can range only between 0 and 1, thus behaving like mutex locks.

Note: When count of semaphore reduces to 0, implying that all resource instances are being used, process that wish to use the resource will block until another process calls signal() and count becomes greater that 0.

3.7 Monitors

A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses *condition variables* that allow processes to wait for certain conditions to be true and to signal one another when condition has been set true. The only operations that can be invoked on condition variables are *wait()* and *signal()*. The monitor ensures that only one process at a time is active within the monitor. Pseudo-code syntax of a monitor is as follows:

```
monitor monitor_name
{
    /* Shared variable declaration */

    function P1 (. . .) {
        . . .
    }

    function P2 (. . .) {
        . . .
    }
        .
        .
        .

    function Pn (. . .) {
        . . .
    }

    initialization_code (. . .) {
        . . .
    }
}
```

Say, condition variable x exists. Now, *x.wait()* means that the process invoking this operation is suspended until another process invokes *x.signal()*. The *x.signal()* operation resumes exactly one suspended process. If no process is suspended, the *signal()* operation has no effect i.e. the state of x is the same as if the operation had never been executed.

Chapter 4

Deadlocks

A set of processes is in deadlocked state when every process in the set is waiting for an event (mainly, acquire and release of resources) that can be caused only by another process in the set.

4.1 Necessary conditions for deadlock

(1) Mutual Exclusion: A resource exists that can only be held by one process at a time. If P_1 is using resource R, P_2 cannot use it at that instant. Alternatively, for this condition, at least one resource must be non-shareable.

(2) Hold and Wait: When a process request more resource from other processes while still holding resources of its own.

(3) No Preemption: There is no preemption of resources that have already been allocated i.e. a process can only release a resource voluntarily.

(4) Circular Wait: A process is waiting for resource held by second process, which is waiting for resource held by third process and so on, till last process is waiting for a resource held by first process.

Note: Deadlock can be **prevented** if any one of the four conditions is prevented from occuring. However, it can cause side-effects like low device utilization and system throughput.

4.1.1 Methods for Handling Deadlock

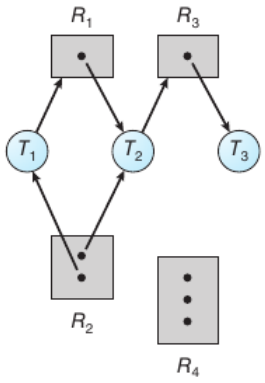
- Ignore the problem and pretend that deadlocks never occur in a system.
- Use algorithms to prevent or avoid deadlocks, ensuring that system will never enter the deadlock state.
- Allowing system to enter deadlock state, detect it and recover.

4.1.2 Resource-Allocation Graph

Deadlocks can be described more precisely using directed graph, which consists of:

- V set of vertices which are of two kinds:
  - $T = \{T_1, T_2, \ldots, T_n\}$, consisting of all active threads, and
  - $R = \{R_1, R_2, \ldots, R_m\}$, consisting of all resource types in system.

- E set of edges. Edge $T_i \rightarrow R_j$ denotes that T_i has requested an instance of resource R_j and is currently waiting for the resource (request edge), while edge $R_j \rightarrow T_i$ denotes that an instance of R_j is allocated to T_i (assignment edge).



Resource-allocation graph

If the graph contains a cycle, deadlock *may* exist. If each resource type has exactly one instance (or, a set of resources where each resource type has only one instance), a cycle is necessary and sufficient condition for existence of a deadlock. Whereas, if each resource type has several instances, a cycle is necessary but not sufficient condition for existence of a deadlock.

4.2 Deadlock Avoidance

4.2.1 Safe State

A system is in safe state only if there exists a *safe sequence* for current allocation $< T_1, T_2, ..., T_n >$, such that for each T_i, the resource request that T_i makes can be granted by currently available resources plus resources held by all T_j with $j < i$. If no such safe sequence exists, system is in unsafe state. A safe state is **not** deadlocked state, and an unsafe state **may** lead to a deadlock.

The idea is to simply ensure that system will always remain in safe state. Initially system is in safe state, whenever a thread requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or the thread must wait. The request is granted only if system after allocation remains in safe state.

4.2.2 Banker's Algorithm

Following data structures are used for implementation:

- Available: vector of length m, indicates number of available resources. $Available[j] = k$ means k instances of resource type R_j are available.
- Max: $n \times m$ matrix, $Max[i][j] = k$ means thread T_i may request at most k instances of R_j.
- Allocation: $n \times m$ matrix, $Allocation[j][j] = k$ means thread T_i is currently allocated k instances of R_j.
- Need: $n \times m$ matrix, $Need[i][j] = Max[i][j] - Allocation[i][j]$.

– **Safety Algorithm** Checks whether system is in safe state. The algorithm is as follows (require an order of $m \times n^2$ operations):

1. Let Work and Finish be vectors of length m and n. Initialize Work = Available and $Finish[i] = false$, $\forall i$.
2. Find an index i such that both
   (a) $Finish[i] == false$
   (b) $Need_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   Go to step 2.
4. If $Finish[i] = true\ \forall i$, then the system is in safe state.

– **Resource-Request Algorithm** Determines whether requests can be safely granted. Let $Request_i$ be the request vector of thread T_i. $Request_i[j] = k$ means thread T_i wants k instances of resource type R_j. When a request of resources is made by thread T_i, following steps are taken:

1. If $Request_i \leq Need_i$, go to step 2. Else, raise an error as thread has exceeded its maximum claim.
2. If $Request_i \leq Available_i$, go to step 3. Else, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated requested resources to T_i by modifying:

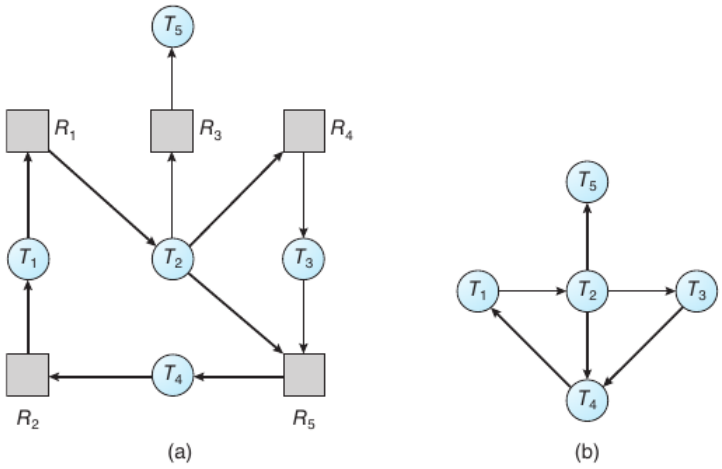
   $$Available = Available - Request_i$$
   $$Allocation_i = Allocation_i + Request_i$$
   $$Need_i = Need_i - Request_i$$

   If the resulting state is safe, the transaction is completed and T_i is allocated its resources. Else, T_i must wait for $Request_i$ and the old resource-allocation state is restored.

4.3 Deadlock Detection

4.3.1 Single Instance of Each Resource Type

A wait-for graph, which is a variant of resource-allocation graph, is used. An edge from T_i to T_j in wait for graph implies that thread T_i is waiting for T_j to release a resource that T_i needs. An edge $T_i \rightarrow T_j$ exists in wait-for graph if and only if the corresponding resource-allocation graph contains two edges $T_i \rightarrow R_q$ and $R_q \rightarrow T_j$ for some resource R_q. Cleary, deadlock exists in the system if and only if the wait-for graph contains a cycle. Here, order of n^2 operations are required.



(a)Resource-allocation graph. (b)Corresponding wait-for graph.

4.3.2 Several Instances of Resource Type

Following time-varying data structures are used:

- Available: vector of length m indicating number of available resources of each type.
- Allocation: $n \times m$ matrix, defines the number of resources of each type currently allocated to each thread.
- Request: $n \times m$ matrix, indicates the current request of each thread.

The algorithm here simply checks every possible allocation sequence for the threads that remain to be completed, and requires order of $m \times n^2$ operations, is defined as follows:

1. Let Work and Finish be vectors of length m and n. Initialize Work = Available and $\forall i$ if $Allocation_i \neq 0$, then $Finish[i] = false$. Else, $Finish[i] = true$.
2. Find an index i such that both
   (a) $Finish[i] == false$
   (b) $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   Go to step 2.
4. If $Finish[i] = false$ for some i, then the system is in deadlocked state. Moreover, if $Finish[i] == false$, then thread T_i is deadlocked.

Here, resources of thread T_i are reclaimed as soon as it is determined that $Request_i \leq Work$, because it is known that T_i is currently *not* involved in deadlock (since $Request_i \leq Work$). Thus, an optimistic attitude is kept and it is assumed that T_i will require no more resources to complete its task, and soon will return all currently allocated resources to the system. If this assumption is incorrect, a deadlock may occur late, which will be detected the next time the algorithm is invoked.

4.4 Deadlock Recovery

4.4.1 Process and Thread Termination

- Aborting all deadlocked processes will clearly break deadlock but at great expense as aborting process may lead to loss of computed data up until now.

- Aborting one process at a time until deadlock is eliminated, incurs incredible overhead as after each process is aborted, deadlock-detection cycle needs to be run again.

4.4.2 Resource Preemption

Preempt resources allocated to the deadlocked process until deadlock cycle is broken. The following needs to be taken care of: (i)Selection of victim, (ii)Rollback (rollback process to some safe state and restrat it from there after preemption), and (iii)Starvation (ensuring it doesn't happen).

Chapter 5

Memory Management

Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address. One way to allocate an address space to each process is through the use of base register (holds smallest legal physical memory address) and limit register (specifies the size of the range). For a program to run, it must be brought into main memory from the disk. Addresses in source program are generally symbolic (example, variables). A compiler **binds** the symbolic addresses to relocatable addresses. The linker or loader in turn binds the relocatable addresses to absolute addresses. Each binding is a mapping from one address space to another.

The binding of instructions and data to memory addresses can be:

- **Compile Time**: If its known at compile time where the process will reside in memory, absolute code can be generated. If starting address changes, entire code needs to be recompiled.
- **Load Time**: Compiler must generate relocatable code and final binding is done at load time. If starting address changes, only user code needs to be reloaded.
- **Execution Time**: If process can move from one memory segment to another, then binding must be delayed until runtime.

Memory Management Unit (MMU) does the runtime mapping from virtual to physical addresses. Address generated by the CPU is referred to as logical (or virtual) address, whereas the address seen by memory unit i.e. the one loaded into memory address register is referred to as physical address. Binding addresses at compile time or load time generates identical logical and physical addresses, whereas, they differ in execution time binding.

5.1 Contiguous Memory Allocation

Here, each process is contained in a single section of memory that is contiguous to the section containing the next process. Dynamic storage allocation problem concerns with how to satisfy a request of certain size from a list of free holes. The strategies used for the problem are first-fit, best-fit, and worst-fit.

5.1.1 Fragmentation

External fragmention exists when there is enough total memory space to satisfy a request but available space is not contiguous. As a solution to external fragmentation, physical memory is split into fixed size blocks (usually sizes are powers of 2) and memory is allocated based on block size. *Internal fragmentation* is unused memory that is internal to a partition.

5.2 Paging

Memory management scheme that permits a processes' physical address space to be non-contiguous. The method involves breaking physical memory into fixed sized blocks called *frames* and breaking logical memory into blocks of same size called *pages*. When a process is to be executed, its pages are loaded into any available memory frames from their source.

Every address generated by CPU has two parts: *page number (p)* and *page offset (d)*. The *page table* contains the base address of each frame in physical memory, and the offset is location in frame being referenced. The steps used by MMU to translate logical address to physical address are:

(1) Extract page number (p) and use it as index in page table.

(2) Extract corresponding frame number f from page table.

(3) Replace page number (p) in logical address with frame number f, so $f + d$ makes up the physical address.

Note: One additional bit is generally attatched to each entry in page table: **valid-invalid bit**. Valid means assosiated page is in process's logical address space and is thus legal page. Invalid means page is not in process's logical address space, so the page is trapped and OS decides to allow or disallow access to the page.

5.2.1 TLB

Memory access is slow using the standard page table since two memory accesses are needed to access data. The solution is using a small, fast-lookup hardware cache called **transition look-aside buffer (TLB)**. It contains key-value entries with certain page numbers and their frame numbers. If the page number exists in the TLB, the corresponding frame number is directly returned (called TLB hit), else if page number is not in TLB (called TLB miss), then its done the usual way. In TLB, effective memory access time is defined by,

$$EAT = TLB\_miss\_time * (1 - hit\_ratio) + TLB\_hit\_time * hit\_ratio$$

5.2.2 Structure of Page Table

- Hierarchical Paging, involves dividing a logical address into multiple parts, each referring to different levels of page tables.
- Hashed Page Tables, where each entry has linked list and proper hash function to lookup entries.
- Inverted Page Tables, which has one entry for each real page (or frame) in main memory. Each entry consists of virtual address of page stored in real memory location.

5.3 Swapping

A process or portion of process can be temporarily swapped out of memory to a backing store and then brought back into memory for continued execution. Swapping makes it possible for the total physical address space for all processes to exceed the real physical memory of system, thus increasing degree of multiprogramming of a system. Swapping can also be done with pages instead of processes i.e. page-in and page-out. However, mobile systems do not support swapping, instead, before terminating the process, it writes the application state to flash memory, so that it can be easily restarted.

x86-64 architectures provides a 48-bit virtual address with support for page sizes 4kB, 2MB or 1GB using four levels of paging hierarchy.

5.4 Virtual Memory

Virtual memory space of a process refers to the logical (or virtual) view of how the process is stored in memory. Advantages of using virtual memory are:

(a) a program can be larger than physical memory.

(b) a program does not need to be entirely in memory.

(c) processes can share memory.

(d) processes can be created more efficiently.

5.4.1 Demand Paging

Pages are loaded only when they are demanded during the program execution. Valid-invalid bit can be used here too. Valid means page is both legal and in memory. Invalid means page is either not valid, or is valid but currently in secondary storage. Access to a page marked invalid causes **page fault**. Basic page fault is handled as follows:

(1) Check internal table of process (PCB) whether reference was valid or invalid.

(2) If reference was invalid, terminate the process. If its valid but the page was not yet brought in, bring it in.

(3) Find a free frame.

(4) Schedule a secondary storage operation to store desired page into newly allocated frame.

(5) Modify internal table of process and page table to indicate that page is now in memory.

(6) Restart the instruction that was interrupted by trap.

5.4.2 Page Replacement

When available memory runs low, page replacement algorithm selects a page in memory to replace it with a new page.

Dirty Bit is set by hardware whenever any byte in the page in written into, indicating that the page has been modified. While selecting for replacement, the dirty bit is checked, set bit implies that the page is modified so it must be written to secondary storage. Bit not set means this copy is same as the one in secondary storage, so the page can be discarded directly.

Basic PR Algorithms

- **First-In-First-Out (FIFO)**: The page that has been in the memory for the longest time is replaced.
- **Optimal PR**: Replace the page that ***will*** not be used for the longest period of time. It's difficult to implement as knowledge about future references is required.
- **Least Recently Used (LRU)**: Replace the page that has not been used for the longest time. Two implementations are feasible to order frames by LRU:

(a) Counters: assosiate each page with a time-used field and update the value with CPU clock when the page is accessed.

(b) Stack: stack of page numbers where page is brought to the top when its accessed, so that the LRU page will always be at the bottom. Can be implemented using doubly linked list.

- **Second-Chance (or Clock) Algorithm**: Circular queue is used for implementation, where a pointer goes around checking its reference bit. If the value is 0, the page is replaced; but if the value is 1, that page is given a second chance, and the pointer clears the bit of that page and continues searching.

- **Least Frequently Used (LFU)**: Replace the page that is used the least number of times.

Belady's Anomaly For some PR algorithms, the page fault rate may increase as the number of allocated frames increases. For example, if FIFO is used and the reference string is {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}; number of page faults are 9 for 3 frames, whereas, the value is 10 for 4 frames. Stack algorithms like LRU and Optimal PR, can never exhibit Belady's anomaly.

Stack Algorithms Algorithms where it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n+1$ frames.

Page Buffering Algorithm used in addition to PR algorithm. A pool of free frames are kept, when page fault occurs, the victim page is chosen as usual but the desired page is read into one of the free frames from the pool before victim is swapped out. Thus, allowing process to restart as soon as possible from the page fault. When the victim is later written out, its frame is added to free-frame pool.

5.5 Thrashing

Thrashing refers to the high paging activity by the system. When the system doesn't have enough frames to accommodate a new page, and must replace a page in active use which'll be needed right away again, quickly resulting in another page fault.

5.5.1 Cause of Thrashing

OS monitors CPU utilization. If utilization is too low, degree of multiprogramming is increased by introducing a new process in the system. When *global page replacement* is used, pages are replaced without regard to the process to which they belong. Now, when that process needs more frames, it takes away frames from another process, which needs those frames, again resulting in page faults. The paging queue lines up and CPU utilization decreases, the scheduler sees and increases degree of multiprogramming yet again. The new process takes away frames, causing other processes to fault, resulting in even more page faults and decreasing CPU utilization. Thus, no work is being done as processes are spending all their time paging.

5.5.2 Solution to Thrashing:

1. Local page replacement algorithm can be used to minimize thrashing, where page from faulting process is selected to be replaced. **Locality Model** is one approach. A locality represents a set of pages that are actively used together. As a process executes, it moves from locality to locality. A working set is based on locality and is defined as the set of pages currently in use by a process.

2. Page-fault frequency can be monitored with an upper and lower bound.

Chapter 6

File Management

6.1 File-System Interface

File is a named collection of related information. File attributes typically consists of:

- Name
- Identifier: unique tag, usually a number that identifies the file within file system.
- Type
- Location: pointer to device and location of file on that device.
- Size: current file size (in bytes or blocks)
- Protection: access-control information
- Timestamps (create, update time) and user identification

Common file types: executable (exe, bin), object (obj, o), source code (c, cpp), batch (bat, sh), multimedia (mp3, mkv), markup (xml, tex), word processor (rtf, docx), library (lib, dll), print or view (gif, pdf), and archive (rar, zip, tar).

6.1.1 File Operations

File is an abstract data type and OS provides operations that can be performed on files. Following are the basic operations:

(1) **Creating file**: space for new file is found and a new entry is made.

(2) **Opening file**: most of the operations are performed after the file is open, hence, several pieces of information are assosiated with an open file:

   (a) File pointer: system must track last read-write location as current-file-position pointer, which is unique to each process operating of the file.

   (b) File open count: multiple processes might have opened the file so system must wait for last file to close before removing the open-file table entry.

   (c) Location of file: info needed to locate the file is kept in memory so that system doesn't have to read it from directory for each operation.

   (d) Access rights: needed so that OS can allow or disallow subsequent I/O requests.

(3) **Writing file**: write pointer is kept in the file where next write is to take place.

(4) **Reading file**: read pointer is kept from where the contents are read.

(5) **Repositioning within file**: current-file-position pointer of an open file is repositioned to a given value. Also known as **file seek**.

(6) **Deleting file**: release all file space and delete the entry from file directory.

(7) **Truncating file**: erase the contents of a file but keep its attributes unchanged.

6.1.2 Access Methods

- Sequential Access Information in the file is processed in order, one record after other. Access time can be decreased if the next record is automatically loaded in memory.

- Direct Access A file is made up of fixed length *logical records* that allows programs to read and write records rapidly in no particular order. For direct access, the file is viewed as a numbered sequence of blocks or records.

- Indexed Access involves construction of an index for the file. The index contains pointers to various blocks.

6.1.3 Directory Structure

- **Single-level Directory**: All files are contained in the same directory. This has significant limitations like confusion of file names among different users.
- **Two-level Directory**: Each user has his own user file directory (UFD), and each lists files of a single user. It has limitations like, users are isolated from one another and a user won't be able to access another user's files or modify them if they wish to cooperate.
- **Tree-structured Directory**: The generalization of directory structure to tree allows users to create their own subdirectories and organize their files accordingly. The tree has *root* directory, and every file in the system has a unique path name. Path names can be of two types:

  (1) Absolute path name: begins at root and goes all the way the the file.

  (2) Relative path name: defines the path of file from current directory.

  Path names allow users to access, modify, and reference other users' files and directories.
- **Acyclic-graph Directory**: Sometimes, a common file or directory needs to be shared. A shared directory or file exists in the file system in two (or more) places at once (not the same as having two copies of file). UNIX systems implement shared directory using *link*, which is effectively a pointer to another file or subdirectory. The drawback is that maintaining links and deletion of shared files becomes complex.
- **General Graph Directory**: If cycles are allowed in graph, more complex algorithms are needed to not go through same section twice. Also, garbage collection is necessary to help determine when the last reference to a file is deleted and space can be reallocated.

6.1.4 Access Control

Most systems use a condensed version of access list and implement three classifications of users for each file:

(1) Owner: user who created the file.

(2) Group: set of users who need a similar access or work group

(3) Other: all other users in the system

For each field, access as read, write and execute is specified by *rwx* and can be expressed in binary with 3 bits and each set bit represents access granted for that field.

6.1.5 File-system Mounting

File system must be mounted before it can be available to the processes of the system. OS is given the name of the device and the *mount point* (location within the file structure where the file system is attatched).

Bootstrap Loader knows enough about file system structure to be able to find and load the kernel module. The root partition is selected by boot loader, which contains OS kernel and other system files, and is mounted at boot time. Then other file systems are mounted manually or through scripts after OS verifies them as valid file systems.

6.1.6 Virtual File System (VFS)

Most OS use object-oriented techniques to simplify, organize, and modularize the implementation. The VFS layer serves two important functions:

(1) Separates file-system generic operations from their implementation by defining clean VFS interface.

(2) Provides mechanisms for uniquely representing file throughout a network. *vnode* contains numerical designation for a network-wide unique file.

Main object types defined by Linux VFS are:

- *inode object*, which represents an individual file
- *file object*, represents an open file.
- *superblock object*, represents an entire file system.
- *dentry object*, represents individual directory entry.

6.2 File-System Implementation

File Systems provide efficient and convenient access to the storage device by allowing data to be stored, located and retrieved easily. Following on-storage data structures are used to implement file system:

- Boot Control Block: (exists per volume) contains info needed by system to boot OS from that volume. This block is empty if disk does not contain any OS.
- Volume Control Block: (exists per volume) contains volume details, such as number of blocks, size of blocks, free-block count and free block pointers, and free FCB-count and FCB pointers.

- Directory Structure: (per file system) used to organize the files. In UNIX systems, these are file names and inode.
- File Control Block: (per file) contains details about the file.

6.2.1 Directory Implementation

- Linear List: use linear list of file names with pointers to the data blocks. For every operation, we'll need to search the entire list so its time-consuming.
- Hash Table: use a linear list to store directory entries, and a hash data structure is used which takes value computed from the file name and returns a pointer to the file in the linear list. This greatly decreases search time, but drawback is that hash tables are generally of fixed size and hash function depends on it.

6.2.2 Allocation Methods

Three major methods are widely used for allocating secondary storage space:

(1) **Contiguous Allocation**: Each file occupies a set of contiguous blocks on the device. For HDDs, number of disk seeks require for accessing contiguously allocated files are minimal, as is the seek time. The directory entry of each file indicated address of starting block and length of the space allocated for file. The drawback is that external fragmentation occurs.

(2) **Linked Allocation**: Each file is a linked list of storage blocks. The directory contains pointers to first and last blocks of the file, and each block contains pointer to the next block. The drawback is that it only works efficiently for sequential access of files. Also, every block takes up extra memory to store pointers to next block. A solution is to group blocks into a *cluster* and then allocate memory for the cluster.

An important variation of linked allocation is use of *file-allocation-table (FAT)* which was used by MS-DOS.

(3) **Indexed Allocation** Each file has its own *index block*, which is an array of storage block addresses. The ith entry in the index block points to ith block in file. The directory contains address of the index block. This allocation supports *direct access*, without suffering from external fragmentation. Its performance depends on index structure, size of file and position of the block desired.

6.2.3 Free Space Management

To keep track of free disk space, the system maintains a free-space list which records all free device blocks. This free-space list is implemented as the following:

- **Bit Vector**: Each block is represented by a bit. If block is free, the bit is 1, and vice versa if block if free. Main advantage here is the relative simplicity and efficiency in finding first free block. Unfortunately, bit vectors are inefficient unless the entire bitmap is kept in main memory. For example, 1-TB disk with 4kB blocks would require $2^{40}/2^{12} = 2^{28} bits = 2^{25} bytes = 32MB$ to store its bitmap.
- **Linked List**: The approach is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory. This scheme is not efficient as, to traverse the list, each block must be read and it takes substantial I/O on HDDs. Fortunately, traversing the free list is not a frequent action.

- **Grouping**: The approach is to store addresses of n free blocks in the first free block. The first *n-1* blocks are actually free, as the last block contains addresses of next n free blocks, and so on.

- **Counting**: The approach is the keep the address of first free block and the number of free contiguous blocks that follow the first block. The entries can be stored in a balanced tree for efficiency.

Chapter 7

Storage Management

A secondary storage device is attatched to the computer by the system bus or an I/O bus. Several kinds of buses are available, including advanced technology attatchment (ATA), serial ATA (SATA), universal serial bus (USB), fibre channel (FC). Also, there is special, fast interface for NVM devices called NVM express (NVMe).

7.1 Mass-Storage Structures

7.1.1 HDD Scheduling

HDDs need to minimize access time and maximise data transfer bandwidth. Following are the methods of scheduling:

(1) **FCFS**: I/O requests to blocks are processed in FCFS order.

(2) **Shortest Seek Time First (SSTF)**: sorts the requests by amount of seek time required to accomplish the request. Request with shortest seek time has the highest priority.

(3) **SCAN**: Disk arm starts at one end of disk and moves towards the other end, servicing requests as it reaches each cylinder. At the end, the direction of head is reversed and servicing continues. This algorithm is also called *elevator algorithm*.

(4) **C-SCAN** i.e. Circular-SCAN, where, after processing requests from start to end, head comes back to start and then again processes requests. This provides a more uniform wait time.

(5) **LOOK**: an improved version of SCAN where head stops after reaching the last request on one end (i.e. doesn't go all the way to the end), then reverses direction and processes requests.

(6) **C-LOOK**: modified LOOK just like C-SCAN.

7.1.2 Drive Formatting, Partitions, and Volumes

A new storage device is a blank slate i.e. just a platter of a magnetic recording material or a set of uninitialized semiconductor storage cells. Before a storage it can store data, it must be divided into sectors that the controller can read and write. NVM pages must be initialized and the FTL (flash transition layer) created. This process is called **low-level formatting**, or physical formatting which fills the device with a special data structure for each storage location. The data structure for a sector or page typically consists of a header, a data area, and a trailer. Most drives are low-level formatted at the factory as a part of the manufacturing process.

Before OS can use a device to hold files, a set of data structures need to recorded on the drive, and the steps are:

(1) **Partition** the drive into one or more groups of blocks or pages. OS can treat each partition as though it were a separate device.

(2) **Volume Creation** and management. This step is implicit when a file system is placed directly within a partition. That *volume* is then ready to be mounted and used. The step is explicit, for example, when multiple partitions or devices will be used together as a RAID set, with one or more file systems spread across the device.

(3) **Logical Formatting** or creation of the file system. Here, OS stores file-system data structures, including maps of free and allocated space, onto the device.

7.2 I/O Systems

The *device drivers* present a uniform device-access interface to the I/O subsystem. Following are the methods of I/O transfer:

- **Programmed I/O**: The processor issues an I/O command, on behalf of a process, to an I/O module; that process then bust waits for the operation to be completed before proceeding.

  Polling: An I/O loop in which I/O thread continuously reads status information waiting for I/O to complete.

- **Interrupts**: A hardware mechanism that enables device to notify the CPU that it needs attention. When interrupt occurs, control is given to *interrupt service routine (ISR)*, which determines cause of interrupt, performs necessary processing, performs state restore, and returns the CPU to execution state prior to interrupt.

- **Direct Memory Access (DMA)**: Computers avoid burdening the main CPU with programmed I/O by offloading some work to special-purpost processor called *direct-memory-access controller*. The DMA controller operates memory bus directly, placing addresses on the bus to perform transfers **without** the help of main CPU. Handshaking between DMA controller and device controller is performed via a pair of wires called DMA-request and DMA-acknowledge. When DMA controller seized memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its cache. The **cycle stealing** i.e. DMA controller using the bus and preventing the CPU from using it temporarily, can slow down CPU computation, but offloading data transfer work to DMA controller generally improves total system performance.

7.2.1 Kernel I/O Subsystem

Kernels provide many services related to I/O which are build on hardware and device-driver infrastructure. The services are:

(1) **I/O Scheduling**: The I/O scheduler rearranges the order of requests queue to improve overall system efficiency and average response time, while also being fair, so that no application receives especially poor service.

(2) **Buffering**: A buffer is a memory area that stores data being transferred between two devices, or device and applications. Buffering is done for following reasons:

   (a) to cope up with speed mismatch between producer and consumer of a data stream.

(b) to provide adaptations for devices that have different data transfer sizes.

(c) to support *copy-semantics* for application I/O. With copy semantics, the version of data written to disk is guarenteed to be the version at the time of application system call.

(3) **Caching**: Cache is a region of fast memory that holds copies of data. Access to cached copy is more efficient than access to original. Cache is different from buffer as buffer can have the only copy of data, but cache has a copy which resides elsewhere.

(4) **Spooling and Device Reservation**: Spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams.

(5) **Error Handling**: An I/O system call will return one bit of information about the status of the call, signifying either success or failure. In UNIX systems, *errno* is returned indicating nature of failure.

(6) **I/O Protection**: It prevents users from performing illegal I/O by defining I/O instructions to be privileged instructions. Hence, a user program executes a system call to request that the operating system perform I/O on its behalf.

(7) **Power Management**: OS monitors the applications using management tools, and could analyze its load and, based of that CPU cores or external devices can be powered off.