**LALITHA MADHURI PUTCHALA**

**MY CENTRAL ID: 700657631**

**ADVANCED OPERATING SYSTEMS**

**ASSIGNMENT -3**

**FALL -2016**

***Theory:***

**5.18**

A lock that makes threads of a process enter into wait condition, while searching for availability of desired lock is known as a spinlock.

Mutual exclusion (mutex): When one process lies in its critical section, the other process wanted to enter its critical section, cannot entered.

A lock can easily be replaced by another type of lock, just by changing the lock variable. As we know, spinlocks can be used for short periods only, they typically avoid or say do not allow threads to sleep.

Whereas, a mutex lock is totally opposite of spinlock and they allow long spinning loops which means threads are allowed to sleep in the wait queue until the lock gets released by current thread.

Below given piece of code, illustrates the normal switching among spinlock and mutex lock while undergoing a program

While (LockObj.dest!- LockObj.compare)

{

if ( HasThreasholdReached ())

{

If (m\_iterations + YIELD\_ITERATION >= MAX\_SLEEP\_ITERATION)

pthread\_mutex\_init ( & mutex, NULL);

Sleep(0);

}

Else

{

pthread\_spin\_init(&spinlock, 0);

SwitchTo Thread();

}

}

Assuming the time taken for context switching will be ‘T’. The code has few variables that work under ‘if-else’ condition.

Where, if the total time taken to make a switch or locate available lock takes longer than the “max\_sleep\_iteration” time, then the thread is allocated to a mutex lock. And the reverse is allocated with the spinlock.

The upper bound for holding a spinlock must be less than or equal to time taken by the context switch (T). If a spinlock is waited for a longer period of time more than T. Then there will be wastage of CPU cycles. If a mutex is used, the maximum CPU cycle time is needed to wake-up the thread. Once a lock is available, in context switch time (T), the remaining time is used by other threads.

**6.1**

Given that there are n processes to be scheduled on one processor, and the first schedule can be done for any of the n processes, the total numbers of possible schedules in terms of ‘n’ are factorial of n => n! (n! = n \* n-1 \* n-2 \* …. \*2\*1)

For example, for 1 process, there is only one possible ordering (1), for 2 processes, there are 2 possible orderings (1,2), (2,1). For 3 processes, there are 6 possible orderings (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1). For 4 processes, there are 24 possible ordering. So, for n processes, there are n! possible orderings.

**7.5**

**Bankers algorithm:**

The given code segment in section 7.5.3 in the textbook, implements the banker’s algorithm. In the given code segment, two for loops are executed as the outer for () loops for n number of times which makes the run time to n^2.

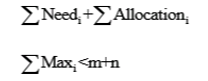
There are two if () conditions given in the code segment and a for loop which will be executed till m times. So, in this case the collective run time will be O(mn^2).

In the worst case if both the conditions are satisfied, the for () loop will be executed till O ((m + n) n^2) times and m + m represents m run time. Thus, in any case the run time of the code segment will be O(mn^2).

**7.18**

A system consisting of m resources of the same type being shared by n processes. Resources can be requested and released by processes only once at a time.

1. the maximum need of each process is between 1 and m resources.



The maximum need of each process = 

Since Max(i) >= 1. It follows the P(i) has the least one resource that it can release.

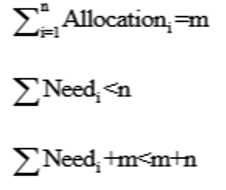
Hence the system is in not deadlock state.

1. The sum of all maximum needs is less than m + n

For all I Max(i) >=1

Need (i) = Max (i) – Allocation (i)

If there exists a deadlock state, then



This implies that there exists a process P(i) such that Need(i) =0.

Since Max(i) >=1. It follows that P(i) has at least one resource that it can release.

Hence the system is in not deadlock state.

**7.24**

The deadlock algorithm given makes the most optimistic assumption about a running process P i.e. P will return all its resources and terminate normally. This assumption gets violated if we still find processes that remain blocked, and then they are deadlocked.

The banker’s algorithm makes the pessimistic assumption about a running process it immediately asks for all the resources it can. If, even with such demanding processes, the resource manager can assure that all process terminates, that we can assure that deadlock is avoided.

***Data Structures:***

**Hash tables** are used to quickly store and retrieve data (or records). Records are stored in **buckets** using **hash keys. Hash keys** are calculated by applying a hashing algorithm to a chosen value contained within the record. This chosen value must be a common value to all the records. Each **bucket** can have multiple records which are organized in a particular order. In operating system, hash table is required for handling address spaces larger than 32 bits, with the hash value being the virtual page number.

Each entry in the hash table contains a linked list of elements that hash to same location to handle collisions. Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows:

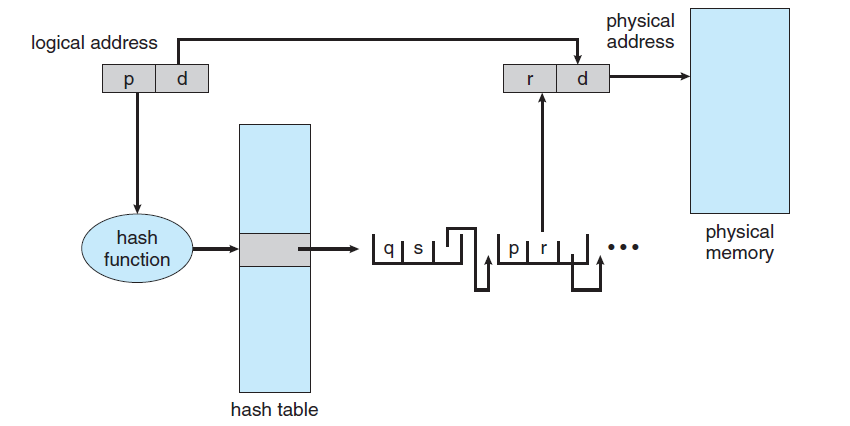
1) the virtual page number in the virtual address is hashed into the hash table.

2) the virtual page number is compared with field 1 in the first element in the linked list.

3) If there is a match, the corresponding page frame (field2) is used to form the physical address.

4) If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

**Design and efficiency:**



The above design uses clustered page tables, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space. The hash table size is based on the number of physical pages. The number of physical pages is usually a lot smaller than the number of all virtual pages put together. The hash table implementation improves the efficiency and performance of the memory access from O(n) to **O (1)**. Each bucket is a functional set containing O (1) elements and the elements of the set as a whole are partitioned among all the buckets.

**Re-size-able hash tables:**

The claim that hash tables give *O* (1) performance is based on the assumption that *n* = *O*(*m*). If a hash table has many elements inserted into it, *n* may become much larger than *m* and violate this assumption. The effect will be that the bucket set will become large enough that their bad asymptotic performance will show through. The solution to this problem is relatively simple: the array must be increased in size and all the element **rehashed** into the new buckets using an appropriate hash function when the load factor exceeds some constant factor αmax. Because resizing is not visible to the client, it is a **benign side effect**. Each resizing operation takes *O*(*n*) time where *n* is the size of the hash table being resized. Therefore the *O* (1) performance of the hash table operations no longer holds in the case of **add**: its worst-case performance is *O*(*n*).

**Time and Efficiency:**

If we start from an empty hash table, any sequence of *n* operations will take *O*(*n*) time, even if we resize the hash table whenever the load factor goes outside the interval [αmax/4, αmax].

To see this, we need to evaluate the amortized complexity of the hash table operations. This formalizes the reasoning we used earlier. To do this, we define a **potential function** that measures the recharged time for a given state of the data structure. The potential function saves up time that can be used by later operations.

**Hash Table for Cache Implementation:**

Hash tables can be used to implement [caches](https://en.wikipedia.org/wiki/Cache_(computing)), auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually erasing the old item that is currently stored in the table and overwriting it with the new item, so every item in the table has a unique hash value.

**Application of Hash Table for Distributed Systems:**

The hash table data structure can be used for distributed file systems. It can be helpful in fast retrial of the data from the inner directories. The implementation can be as follows:

1) Here a linear list stores the directory entries, but a hash data structure is also used.

2) The hash table takes a value computer from the file name and returns a pointer to the file name in the linear list.

3) Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions – situations in which two file names hash to the same selection.

4) The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. This problem can be solved by using a special type of page tables called chained-overflow hash table.

5) All these properties of leads to new type of has table called distributed hash table (DHT) that can be applicable for large scale distributed systems.

A distributed hash table (DHT) is a class of a decentralized [distributed system](https://en.wikipedia.org/wiki/Distributed_computing) that provides a lookup service similar to a [hash table](https://en.wikipedia.org/wiki/Hash_table): (key, value) pairs are stored in a DHT, and any participating [node](https://en.wikipedia.org/wiki/Node_(networking)) can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to [scale](https://en.wikipedia.org/wiki/Scale_(computing)) to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

DHTs form an infrastructure that can be used to build more complex services, such as [any cast](https://en.wikipedia.org/wiki/Anycast), cooperative [Web caching](https://en.wikipedia.org/wiki/Web_cache), [distributed file systems](https://en.wikipedia.org/wiki/Distributed_file_system), [domain name services](https://en.wikipedia.org/wiki/Domain_name_system), [instant messaging](https://en.wikipedia.org/wiki/Instant_messaging), [multicast](https://en.wikipedia.org/wiki/Multicast), and also [peer-to-peer](https://en.wikipedia.org/wiki/Peer-to-peer) [file sharing](https://en.wikipedia.org/wiki/File_sharing) and distribution systems. Notable distributed networks that use DHTs include Bit Torrent’s distributed tracker,