Concurrency

Interprocess Communication

A process is independent if it cannot affect or be affected by other processes running in the system. A process is cooperating if it is not independent.

There are several reasons for providing an environment that allows process cooperation:

1. Information sharing
2. Computation speedup
3. Modularity
4. Convenience

Cooperating processes need an interprocess communication scheme allowing them to exchange data and information. The two fundamental models of IPC are

1. Shared Memory
    a. Processes access a shared memory space
    b. Faster than message passing
    c. No assistance from kernel required once shared memory is established
2. Message Passing
    a. Messages are exchanged between processes
    b. Efficient for small amounts of data
    c. Easier to implement

Shared Memory Systems

Typically, a shared memory region resides in the address space of the process creating the shared memory segment. Other processes just attach to this address space. This mechanism uses buffers, where the producer of the data writes to the buffer and the consumer of the data reads what the producer has written. Two types of buffers can be used

1. Bounded buffers - Fixed size
2. Unbounded buffers - No limit on buffer size

The shared buffer is implemented as a circular array with two logical pointers. The first points to the next free position in the buffer and the other to the next full position in the buffer.

Message Passing

This allows processes to not use the same address space, but synchronize actions based on messages, particularly used in distributed systems. Messages can be fixed or variable sizes, with a tradeoff on complexity. A link between two processes can be formed through

1. Direct or Indirect Communication
2. Synchronous or Asynchronous Communication
3. Automatic or Explicit Buffering

Naming is done to allow processes to refer to each other.

1. In direct communication,
    a. Link is established between every pair of processes wanting to communicate
    b. Link is associated with two processes
    c. Between each pair of processes, there exists only one link
    d. Both sender and receiver must specify names in symmetric, but not in asymmetric
    e. This gives limited modularity in process definitions
2. In indirect communication,
    a. Messages sent from ports
    b. Link is established between a pair of processes only if both members have a shared mailbox/buffer
    c. A link may be associated with two or more processes
    d. Between each pair, multiple links can exist, each pointing to a mailbox
3. In synchronization, message passing is either blocking or non-blocking
    a. Blocking send - Sending process is blocked until message is received
    b. Non-blocking send - Sending process sends message and resumes
    c. Blocking receive - Receiver blocks until a message is available
    d. Non-blocking receive - Receiver retrieves a valid message or null.

When both send and receive are blocking, a rendezvous occurs between the processes.

Buffering is implemented in three ways

1. Zero Capacity - Link has no messages waiting (block sender)
2. Bounded Capacity - Link has finite length
3. Unbounded Capacity - Link has infinite length

A pipe acts as a conduit allowing two processes to communicate. Four issues must be considered here

1. Bidirectional communication
2. Half duplex or Full duplex
3. Relationship between processes
4. Network communication or local communication

Pipes can be ordinary or named.

1. Ordinary Pipes
    a. Allow processes to write at one end and the consumer reads at the other end
    b. Unidirectional
    c. Constructed as pipe(int fd[])
    d. Can be accessed by ordinary read() or write() calls
    e. Require a parent-child relationship
2. Named Pipes
    a. Bidirectional
    b. No relationship needed
    c. Referred to as FIFOs in UNIX
    d. Half duplex communication in UNIX, full duplex in Windows

Threads

A thread is a basic unit of CPU utilisation. It consists of

1. Thread ID
2. Program counter
3. Register set
4. Stack

It shares with other threads belonging to the same process its code, data and OS sections.

Used as an alternative to create processes upon events to save time. They play a vital role in remote procedure call (RPC) systems.

Benefits of threads

1. Responsiveness
    a. Multithreading allows a program to keep running while servicing a thread
2. Resource Sharing
    a. Threads share resources of parent process by default
    b. Allows different process threads in the same address space

3. Economy
    a. Low overhead as resources are shared
4. Scalable

Multicore Programming

Multithreaded programs allow a mechanism for more efficient use of multiple computing cores.

A system is parallel if it can perform more than one task simultaneously. A concurrent system supports more than one task by allowing all tasks to make progress.

Amdahl's law gives the speedup when the number of cores or the parallelism is increased.

Programming in multicore environments pose the following challenges

1. Identifying tasks
2. Balance of processes
3. Data splitting
4. Data dependencies
5. Testing and Debugging

The types of parallelism are

1. Data parallelism
2. Task parallelism

Multithreading Models

There are two kinds of threads, kernel and user threads. User threads require kernel threads for their execution.

1. Many-to-one model
    a. Maps many user threads to one kernel thread
    b. Entire process blocks if one thread issues a blocking call
2. One-to-one model
    a. One kernel thread associated to one user thread
    b. Provides more concurrency than many-to-one
    c. Creating user thread requires kernel threads, reduces performance
3. Many-to-many model
    a. Many user threads to a number of kernel threads
    b. Referred to as a two level model because it binds a user thread to a kernel thread

The Thread API

1. Thread creation is done using the

   pthread_create(pthread_t * thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg);

   function. The attr variable is initialised to a call to pthread_attr_init().

2. Thread completion is done using

   int pthread_join(pthread_t thread, void **value_ptr);

   thread is the thread to wait for.

Locks

Beyond thread creation and join, mutual exclusion is provided through locks, provided by

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

This is used so that two threads do not simultaneously work on the critical section. Once a thread acquires a lock to that section, it can execute there.

Trylock and timedlock can be used to poll for locks and keep a timeout for processing holding locks respectively.

Condition Variables

Condition variables are useful for signalling mechanisms between threads to parse events to different threads. pthread_cond_wait() puts the calling thread to sleep, and pthread_cond_signal() wakes a sleeping thread up.

The signal call takes only one condition, but the wait call takes the lock as well. This is because in the wait call, in addition to putting the calling thread to sleep, any locks held by the thread need to be released, or this results in a deadlock.

Process Synchronization and the Critical Section Problem

When several processes access a shared variable concurrently, a race condition might occur, as the order of accessing the shared variable may change. To guard against such conditions, scheduling algorithms are designed and processes are synchronized with each other.

The segment of code containing shared variables is called the critical section. The important feature considered by the system is, when a process is executing in the critical section, no other process is allowed to execute in that section. Each process must request entry into a critical section from an entry section, and once it has finished executing, it exits through an exit section.

A solution to the critical-section problem must address three requirements

1. Mutual exclusion : If a process is executing in the critical section, no other process is allowed in.
2. Progress : If no process is executing in the critical section and some processes wish to enter the critical section, then only processes not executing in the remainder section can be chosen, and selection cannot be postponed indefinitely.
3. Bounded waiting : There exists a limit as to how many times other processes are allowed to enter a critical section while a process is waiting.

Two general approaches to handling this problem are

1. Preemptive Kernels
    a. Allows process to be preempted while running in kernel mode.
    b. More responsive
    c. More suitable for real-time programming
2. Non-preemptive Kernels
    a. Does not allow a process running in kernel mode to be preempted.
    b. Free from race conditions

Peterson's solution

This is a solution to the critical section problem, assumed for two processes.

Peterson's solution requires the two processes to share a turn integer and a 2 element boolean flag array. Turn indicates whose turn it is to enter the critical section. The flag array indicates whether the process is ready to enter the critical section.

To enter the critical section, the process P first sets flag[P] to be true, and then sets the turn value to 1 - P, indicating other processes can enter the critical section. If both processes do this simultaneously, one of the processes prevails.

To prove this is a valid solution,

1. Mutual Exclusion
    a. A process enters the critical section only if its flag is set and it is its turn to enter
2. Progress and Bounded Waiting
    a. A process can be prevented from entering only if it is not its turn or its flag is not set.
    b. P does not change the value of the turn when executing, but only after at most one entry by another process.

Synchronization Hardware

Solutions to the critical section problem can be by

1. In a single processor environment,
    a. Disable interrupts when a shared variable is being modified
    b. Ensures correct sequence of instructions
    c. Not feasible in a multiprocessor environment
    d. Disabling interrupts can be time consuming
    e. System efficiency is also decreased
2. Test and modify
    a. Allows modification and swapping as atomic operations
    b. Uses a test_and_set and a compare_and_swap function
    c. Both functions execute atomically and sequentially
    d. Test_and_set checks if a lock has been set, and if it hasn't, it sets the lock.
    e. Compare_and_swap checks if the lock has been acquired, and changes the value only if it has.
    f. These satisfy only mutual exclusion, but not bounded waiting
    g. For this, we add a waiting array, which satisfies all three conditions as
        i. At a time, only one waiting[i] is false
        ii. Since a process either sets the lock to false or waiting[i] to false, it allows other processes to enter
        iii. Since the queue is cyclic, bounded wait does not occur, as the process waiting will be encountered in n-1 turns.

Mutex Locks

Mutex is short for Mutual Exclusion. Mutex locks are used to protect critical regions and prevent race conditions. A process must first acquire this mutex lock to execute in the critical section, and release the lock when it exits.

Calls to acquiring and releasing the lock must be atomic. The main disadvantage of the implementation is busy waiting.

While a process is in the critical section, the other processes need to keep looping calling acquire(). This type of lock is called a spinlock because the process spins while trying to acquire the lock. This wastes a lot of CPU cycles.

Spinlocks have an advantage though, in that no context switch is required when a process must wait on a lock, and a context switch may take a considerable amount of time. Thus, when locks are being expected to be held for a short time, spinlocks are useful.

Semaphores

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

```
wait(S) {

while(S<=0)

        ;

S--;

}

signal(S) {

S++;

}
```

There are two main types of semaphores: counting semaphores and binary semaphores.

Binary semaphores can take values 0 and 1, and are similar to mutex.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. Each process who wants to use a resource calls a wait(), and calls signal() when it releases the resource.

To overcome the problem of busy waiting, the process, when not able to acquire the resource, blocks itself and puts itself in a waiting queue, and the state is changed to waiting. The CPU scheduler then chooses another process to execute. When the resource is freed, the process from the waiting queue is made ready, and executed, negating the need for constant polling.

The list of waiting processes can be implemented by a link field in each PCB. To ensure bounded waiting, we can use a FIFO queue, where the semaphore has the head and tail pointers of the queue.

Deadlocks and Starvation

This happens when two processes are waiting indefinitely for an event that can be triggered by one of the waiting processes only. Another problem related to deadlocks is starvation or indefinite blocking, in which a process waits indefinitely within the semaphore.

Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that is being accessed by a lower-priority process. Since kernel data is held by a lock, the high priority process has to wait. This is called priority inversion.

This is solved by implementing a priority-inheritance protocol. According to the protocol, all processes that are accessing resources needed by the higher process inherit a higher priority until they are done with the resources in question. The priority is restored when execution is done.

Classic Synchronization Problems

1. **Bounded Buffer Problem**

   There is a pool consisting of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool, and is initialised with value 1. The semaphore empty is initialised to a value n and the semaphore full to 0.

2. **Readers-Writers Problem**

   This problem occurs on multiple read/writes concurrently with a write. For this, we require the writers to have exclusive access to the file. The solutions are

   1. First-Readers-Writers
      a. No reader is kept waiting unless a writer has acquired permission to write
   2. Second-Readers-Writers
      a. Once a writer is ready, the writer writes as soon as possible.

   Both of the above lead to starvation. So we define semaphores on the files, where one semaphore keeps track of the readers and one of the writers.

3. **Dining-Philosophers Problem**

   This problem details about how to allocate several resources among several processes in a deadlock-free and starvation-free manner. One solution is to represent each resource with a semaphore. This could create a deadlock. This is fixed by

   1. Allow at most k processes to be executing simultaneously
   2. Allow a semaphore release only if another instance of a resource is available
   3. Use some asymmetric solution to pick resources

More on Deadlocks

In the normal mode of operation, a process may utilise a resource in the following sequence

1. Request
2. Use
3. Release

A deadlocked situation arises when all these are satisfied simultaneously

1. Mutual exclusion - at least one resource is held in non-shareable mode
2. Hold and wait - A process must be holding at least one resource and waiting to acquire additional resources
3. No preemption
4. Circular wait - For a set of waiting processes {P0..Pn}, P0 is waiting for P1, P1 for P2...Pn for P1.

Deadlocks can be defined more precisely by a resource allocation graph, where the vertices denote the processes and the resources. A directed edge from a process to a resource signifies that a process P has requested for a resource R, and a directed edge from R to P denotes that R has been assigned to P.

If this graph contains no cycles, then no process is deadlocked. If the graph contains a cycle, a deadlock may exist. (Certain only when the resource in the cycle has only one instance).

When there are several instances of a resource type, we implement deadlock detection where there are time varying data structures

1. Available : vector of length m indicating number of available resources of each type
2. Allocation : An n x m matrix defining allocations
3. Request : An n x m matrix indicating outstanding requests

The detection algorithm is

1. Let Work and Finish be vectors of length m and n.
2. Initialise Work = Available.
3. For i = 0…n-1, if Allocation(i) != 0, then Finish[i] = false. Else Finish[i] = true.
4. Find index i such that
    a. Finish[i] = false
    b. Request[i] <= Work
5. If no such i exists, go to 4.
6. Work += Allocation[i], Finish[i] = true. Go to 1.
7. If Finish[i] = false for some 0 <= i < n, then the system is deadlocked, and the process i is in deadlock.

We can invoke this algorithm every time a request for allocation cannot be granted, as computing this algorithm every time is computationally very expensive.