**Chapter 5: Synchronization and Scheduling**

This chapter delves into two fundamental concepts in operating systems: synchronization and scheduling. Synchronization focuses on managing shared resources accessed by multiple processes or threads to prevent data inconsistencies and race conditions. Scheduling, on the other hand, is about efficiently utilizing computing resources like CPU cores and accelerators by mapping tasks to them effectively.

**5.1 Synchronization**

**5.1.1 Introduction to Data Races**

**Explanation:** In a multicore CPU environment, multiple threads or processes might try to access and modify shared resources concurrently. A simple operation like incrementing a shared counter (count++) can lead to errors if not handled properly. This is because count++ typically translates to multiple assembly instructions (load, increment, store) which are not executed as a single, indivisible unit (atomically). If multiple threads interleave these instructions, they might read the same initial value, increment their local copies, and then write back the incorrect final value, leading to a "data race". This occurs when concurrent accesses to a shared variable happen, and at least one of them is a write.

**Key Concepts & Examples:**

* **Shared Resource:** A variable or hardware component accessed by multiple threads/processes (e.g., countvariable).
* **Race Condition:** Competition among threads/processes to acquire a shared resource.
* **Data Race:** A specific type of race condition where concurrent and conflicting accesses to a shared variable occur. Conflicting accesses involve at least one write operation.
* **Atomicity:** A sequence of operations that appears to happen instantaneously as a single, indivisible unit. The three assembly instructions for count++ are *not* atomic.
* **Example:** Figure 5.1 and 5.2 illustrate how two threads incrementing count simultaneously without synchronization can result in count being 1 instead of the expected 2.

**Diagrams:**

* Figure 5.1: Shows two threads attempting to increment count in parallel.
* Figure 5.2: Illustrates a problematic execution scenario where a data race leads to an incorrect result.

**Solution (using Locks):** To prevent data races in critical sections (code accessing shared resources), a locking mechanism is used. A thread must acquire a lock before entering the critical section and release it afterward. Only one thread can hold the lock and execute the critical section at any given time.

**Diagrams:**

* Figure 5.3: Depicts the concept of using a lock to protect a critical section.
* Figure 5.4: Shows the correct execution of count++ by two threads when protected by locks, resulting in the correct final value of 2.

**Quick Revision:**

* Data races happen with concurrent, conflicting access to shared variables.
* count++ is not atomic and can cause data races.
* Locks ensure only one thread accesses a critical section at a time.
* Acquire lock -> Critical section -> Release lock.

**5.1.2 Design of a Simple Lock**

**Explanation:** A simple lock can be designed using a shared variable (e.g., at memory address A) that indicates the lock's state: 0 for free and 1 for busy. Threads interested in acquiring the lock repeatedly check its value (test phase). If it's 1 (busy), they keep waiting. If it becomes 0 (free), they attempt to atomically set it to 1 (test-and-set phase). This test-and-set operation must be atomic to avoid race conditions in the lock acquisition itself.

**Key Concepts & Examples:**

* **Test-and-Test-and-Set (TTAS) Lock:** A type of spin lock where a thread repeatedly tests the lock status and then attempts an atomic test-and-set operation.
* **Atomic Instruction:** A hardware instruction that executes indivisibly, guaranteeing that only one thread succeeds if multiple try to execute it simultaneously on the same location. Test-and-set is an example.
* **Busy Waiting (Spin Lock):** When a thread repeatedly checks the lock status while waiting, consuming CPU cycles. A spin lock is a lock that uses busy waiting.
* **Read-Modify-Write (RMW) Operations:** Atomic hardware primitives that read a memory location, potentially test its value, modify it, and write it back. Used in implementing locks.

**Diagrams:**

* Figure 5.5: Illustrates the logic of the test-and-test-and-set (TTAS) lock.

**Challenges with Memory Reordering:** Compilers and hardware can reorder instructions for performance. This can lead to situations where memory operations within a critical section become visible to other threads before the lock is fully acquired or after it's released, violating the intended synchronization.

**Solution (using Fence Instructions):** Fence instructions (or memory barriers) are used to prevent instruction reordering across the fence. A fence after acquiring a lock ensures instructions within the critical section don't complete before the lock is fully acquired. A fence before releasing a lock ensures all writes within the critical section are visible to other threads after the release.

**Key Concepts:**

* **Fence Instruction (Memory Barrier):** An instruction that prevents reordering of memory operations across it.
* **Acquire Fence:** Ensures instructions after it in program order don't complete before it does (used during lock acquisition).
* **Release Fence:** Ensures instructions before it in program order complete before it does (used during lock release).

**Quick Revision:**

* Simple locks use a flag (0 for free, 1 for busy).
* Atomic test-and-set is crucial for lock acquisition.
* Busy waiting (spin locks) consumes CPU.
* Fence instructions prevent harmful instruction reordering around critical sections.

**5.1.3 Theory of Data Races**

**Explanation:** A data race is formally defined as concurrent and conflicting access to a shared variable. Conflicting access means at least one operation is a write. Understanding concurrent access requires the notion of a "happens-before" relationship. Event 'a' happens before event 'b' if a chain of events in a given execution leads from 'a' to 'b'. This is a transitive relationship. Concurrent accesses are those without a happens-before relationship between them. Synchronization primitives like locks and their inherent fences establish happens-before relationships, preventing data races. Data races are undesirable as they lead to hard-to-detect bugs and affect the correctness of parallel programs.

**Key Concepts:**

* **Happens-Before Relationship:** A causal ordering between events in a concurrent system.
* **Conflicting Access:** Accesses to the same shared variable where at least one is a write.
* **Concurrent Access:** Accesses without a happens-before relationship.

**Properly-Labeled Programs:** To avoid data races, shared variables should be consistently protected by the same lock or set of locks. If different locks protect different critical sections that access the same shared variable, a data race can still occur.

**Diagrams:**

* Figure 5.6: Shows a scenario where a data race can occur on variable C because it's protected by different locks (X and Y) in different critical sections.

**Quick Revision:**

* Data races = concurrent + conflicting access.
* Happens-before defines causal order.
* Synchronization creates happens-before relationships.
* Use the same lock for the same shared variable (properly labeled programs).

**5.1.4 Deadlocks**

**Explanation:** Using locks can introduce the possibility of deadlocks, a situation where threads are blocked indefinitely while waiting for resources held by other blocked threads, forming a circular dependency. In a deadlock, no thread can make progress.

**Key Concepts:**

* **Deadlock:** A state where two or more threads are blocked indefinitely, each waiting for a resource held by another blocked thread.
* **Circular Wait:** A necessary condition for deadlock where a set of threads are waiting for resources in a cyclic fashion.

**Four Conditions for Deadlock:** For a deadlock to occur, all four of the following conditions must be met:

1. **Hold-and-Wait:** A thread holds at least one resource and is waiting to acquire additional resources held by other threads.
2. **No Preemption:** Resources cannot be forcibly taken away from a thread that holds them; they can only be released voluntarily.
3. **Mutual Exclusion:** Resources are held in a non-sharable mode, meaning only one thread can use a resource at a time.
4. **Circular Wait:** A cycle exists in the wait-for graph, where each thread in the cycle is waiting for a resource held by the next thread in the cycle.

**Diagrams:**

* Figure 5.7: Illustrates a simple deadlock scenario with two threads and two locks (Thread 1 holds X, waits for Y; Thread 2 holds Y, waits for X).

**The Dining Philosopher's Problem:** This is a classic example illustrating deadlocks. Philosophers sit around a table with a fork between each pair. Each needs two forks to eat. If all philosophers pick up their left fork simultaneously, they will all be waiting for their right fork (held by their neighbor), resulting in a circular wait and a deadlock.

**Diagrams:**

* Figure 5.8: Depicts the setup of the Dining Philosopher's problem.

**Solving the Dining Philosopher's Problem (Avoiding Circular Wait):** One solution is to break the circular wait condition. This can be achieved by introducing asymmetry, for example, by having one philosopher pick up their right fork before their left, while others pick up their left before their right.

**Deadlock Prevention, Avoidance, and Recovery:**

* **Prevention:** Design the system to prevent at least one of the four necessary conditions for deadlock from occurring. An example is a two-phase locking protocol where all required locks are acquired in a predefined order (e.g., ascending order of addresses).
* **Avoidance:** Dynamically check if granting a resource request will lead to an unsafe state (one where a deadlock is possible). If so, the request is denied. The Banker's algorithm is an example.
* **Recovery:** Allow deadlocks to occur, detect them, and then break them, for example, by preempting resources or terminating processes involved in the deadlock.

**Starvation and Livelocks:**

* **Starvation:** A thread is repeatedly denied access to a shared resource and may wait indefinitely. Atomic instructions don't guarantee fairness, potentially leading to starvation. Starvation freedom implies deadlock freedom.
* **Livelock:** Threads continuously change their state in response to each other but do not make any meaningful progress towards completion. They are not blocked like in a deadlock but are stuck in a cycle of state changes. The example of two people trying to pass in a narrow corridor illustrates a livelock.

**Quick Revision:**

* Deadlock requires four conditions: hold-and-wait, no preemption, mutual exclusion, circular wait.
* Dining Philosopher's is a classic deadlock example.
* Deadlocks can be prevented, avoided, or recovered from.
* Starvation is indefinite waiting for a resource.
* Livelock is being busy but making no progress.

**5.1.5 Pthreads and Synchronization Primitives**

**Explanation:** Pthreads (POSIX threads) are a standard way to create and manage threads in Linux-like operating systems. They provide functions for creating threads, passing arguments, receiving return values, and joining threads. Pthreads also offer synchronization primitives like mutexes (locks).

**Key Concepts & Examples:**

* **Pthreads:** A standard API for thread management.
* pthread\_create: Function to create a new thread.
* pthread\_join: Function to wait for a thread to terminate and retrieve its return value.
* **Mutex:** A synchronization primitive that provides mutual exclusion, allowing only one thread to access a shared resource at a time.
* pthread\_mutex\_init: Initializes a mutex.
* pthread\_mutex\_lock: Acquires a mutex.
* pthread\_mutex\_unlock: Releases a mutex.
* **Critical Section:** The code segment that accesses shared resources and must be protected by synchronization primitives.

**Code Blocks:**

* **Listing 5.2: Code to create two pthreads and collect their return values**
  + This C code demonstrates the basic usage of pthreads.
  + #include <pthread.h>: Includes the pthreads library.
  + pthread\_t tid[2];: Declares an array to store thread IDs.
  + func(void \*arg): The function executed by each thread. It receives a void\* argument (in this case, the thread ID), prints it, calculates a return value, and returns a void\* pointer to it.
  + main(): The main function creates two threads using pthread\_create.
  + pthread\_create(&tid[i], NULL, &func, ptr): Creates a thread.
    - &tid[i]: Address of the pthread\_t variable to store the new thread's ID.
    - NULL: Pointer to thread attributes (using default attributes).
    - &func: Pointer to the function the thread will execute.
    - ptr: Pointer to the argument passed to the thread function.
  + pthread\_join(tid[i], (void\*\*)&result): Waits for a thread to finish and retrieves its return value through the result pointer.
* **Listing 5.3: Lock-unlock using pthreads**
  + This code shows how to protect a shared variable (count) using a pthread mutex.
  + pthread\_mutex\_t cntlock;: Declares a mutex variable.
  + pthread\_mutex\_init(&cntlock, NULL);: Initializes the mutex.
  + pthread\_mutex\_lock(&cntlock);: Acquires the mutex before accessing count.
  + count++;: The critical section.
  + pthread\_mutex\_unlock(&cntlock);: Releases the mutex after accessing count.
* **Listing 5.4: Lock-unlock using atomic fetch and add**
  + This demonstrates using atomic operations (specifically atomic\_fetch\_add) to increment a shared variable without explicit locks.
  + #include <stdatomic.h>: Includes the atomic operations library.
  + atomic\_int count = 0;: Declares an integer variable as atomic.
  + atomic\_fetch\_add(&count, 1);: Atomically adds 1 to count. This operation is guaranteed to complete without interruption, preventing data races for this specific operation.
* **Listing 5.5: The operation of the CAS method in C-like code**
  + This pseudocode illustrates the logic of a Compare-and-Swap (CAS) operation.
  + bool CAS(int \*valptr, int \*oldptr, int new): The function takes a pointer to the value (valptr), a pointer to the expected old value (oldptr), and the new value (new).
  + if (\*valptr == \*oldptr): Compares the current value at valptr with the expected old value.
  + \*valptr = new;: If they are equal, the value at valptr is updated to new.
  + \*oldptr = \*valptr;: If they are not equal, the current value at valptr is written back to oldptr.
  + The function returns true if the swap was successful (values were equal) and false otherwise. This entire operation is intended to be atomic.
* **Listing 5.6: Lock-unlock using the compare and swap instruction**
  + This code shows how to implement a lock-free counter increment using a CAS operation.
  + atomic\_int count = 0;: Declares count as an atomic integer.
  + #define CAS atomic\_compare\_exchange\_strong: Defines a macro for the CAS function.
  + do { ... } while (!CAS(&count, &oldval, newval));: This loop repeatedly attempts to increment count.
  + oldval = atomic\_load(&count);: Reads the current value of count.
  + newval = oldval + 1;: Calculates the desired new value.
  + CAS(&count, &oldval, newval): Attempts the atomic CAS operation. If the current value of count is still oldval, it is updated to newval, and CAS returns true. If count has been changed by another thread, CAS returns false and updates oldval with the new value of count, and the loop retries. This process continues until the increment is successful.

**Non-Blocking Algorithms:** Atomic operations like atomic\_fetch\_add and compare-and-swap are used to implement non-blocking algorithms, which do not rely on locks and can offer better performance and avoid issues like deadlocks. However, they can be complex to design and verify.

**Quick Revision:**

* Pthreads for thread management in Linux.
* Mutexes for mutual exclusion (locks).
* Atomic operations allow lock-free synchronization for simple operations.
* CAS is a key atomic primitive for building non-free algorithms.
* Non-blocking algorithms avoid locks but can be complex.

**5.1.6 Theory of Concurrent Non-Blocking Algorithms**

**Explanation:** Non-blocking algorithms avoid the use of locks for synchronization. They can be classified based on their progress guarantees: obstruction-free, lock-free, and wait-free.

**Key Concepts:**

* **Obstruction-Free:** If a thread is running alone (no other threads are actively trying to complete their operations), it will complete its operation in a bounded number of steps. Does not use locks.
* **Lock-Free:** If the system as a whole makes sufficient progress (cumulative number of steps across all threads exceeds a bound), at least one thread is guaranteed to complete its operation. Guarantees no deadlocks or livelocks, but starvation is possible.
* **Wait-Free:** Every thread is guaranteed to complete its operation within a bounded number of its own internal steps, regardless of the activity of other threads. Guarantees no starvation, livelocks, or deadlocks.

**Relationship between Progress Guarantees:** Wait-free implies lock-free, and lock-free implies obstruction-free. The converse is not necessarily true.

**Diagrams:**

* Figure 5.11: Venn diagram showing the relationship between obstruction-free, lock-free, and wait-free algorithms.

**Correctness of Concurrent Algorithms (Linearizability):** A key correctness criterion for concurrent data structures is linearizability. A parallel execution is linearizable if each method call appears to take effect instantaneously at some point between its invocation and return time (a "completion point"). This allows reasoning about parallel executions as if they were sequential.

**Key Concepts:**

* **Linearizability:** A correctness property for concurrent objects where operations appear to take effect atomically at some point within their execution interval.
* **Completion Point:** A hypothetical point in time between the start and end of an operation where it appears to take effect instantaneously.

**Diagrams:**

* Figure 5.9: Illustrates a parallel execution of queue operations and an equivalent sequential execution that satisfies linearizability.

**Memory Models:** Memory models (or memory consistency models) define the rules for when memory operations (reads and writes) become visible to other threads in a multicore system. Relaxed memory models allow reordering of instructions for performance, which can lead to unexpected outcomes if not managed carefully (e.g., using fences).

**Key Concepts:**

* **Memory Model:** Defines the rules for visibility and ordering of memory operations across multiple processors.
* **Sequential Consistency (SC):** A strong memory model where all operations appear to execute in a single global order, and operations within a single thread appear in program order.
* **Weak Memory Models:** Allow more reordering than SC for performance.

**Quick Revision:**

* Non-blocking algorithms: obstruction-free, lock-free, wait-free.
* Wait-free > Lock-free > Obstruction-free in terms of progress guarantees.
* Linearizability: Parallel operations appear instantaneous within their duration.
* Memory models define memory operation visibility; fences control reordering.

**5.1.7 Progress Guarantees**

**Explanation:** Progress guarantees for concurrent algorithms are typically defined in terms of "internal steps" rather than physical time, as delays between steps can be arbitrary due to factors like context switches.

**Key Concepts:**

* **Internal Step:** A basic atomic action within a thread (e.g., read, write, instruction execution).
* **Obstruction Freedom:** A thread makes progress in a bounded number of steps if unobstructed by other threads.
* **Wait Freedom:** Every thread makes progress in a bounded number of its own steps.
* **Lock Freedom:** The system as a whole makes progress (at least one operation completes) within a bounded number of total steps across all threads.

**Quick Revision:**

* Progress measured by internal steps, not physical time.
* Obstruction-free: Progress when alone.
* Wait-free: Every thread progresses.
* Lock-free: System progresses (at least one thread).

**5.1.8 Semaphores**

**Explanation:** Semaphores are synchronization primitives that act as a generalization of locks. Unlike a binary lock (0 or 1), a semaphore maintains a non-negative integer count.

**Key Concepts:**

* **Semaphore:** A synchronization primitive with an integer count.
* **sem\_wait (or wait):** Decrements the semaphore count. If the count is 0, the thread blocks (waits in a queue).
* **sem\_post (or post, signal):** Increments the semaphore count. If there are threads waiting, one is woken up.
* **Binary Semaphore:** A semaphore with a maximum count of 1, equivalent to a lock.

**Code Blocks:**

* **Listing 5.7: The sem\_wait operation**
  + This pseudocode shows the atomic operation of sem\_wait.
  + if (count == 0): Checks if the semaphore count is zero.
  + insert\_into\_wait\_queue(current\_task);: If the count is zero, the current thread is added to a waiting queue.
  + else count--;: If the count is greater than zero, it is atomically decremented, and the thread acquires the semaphore.
* **Listing 5.8: The sem\_post operation**
  + This pseudocode shows the atomic operation of sem\_post.
  + if ((count == 0) && process\_waiting()): Checks if the count is zero AND there are processes waiting.
  + wake\_from\_wait\_queue();: If the condition is true, a waiting process is woken up.
  + else count++;: Otherwise (if the count is not zero or no processes are waiting), the count is atomically incremented.

**Quick Revision:**

* Semaphores are generalized locks with a count.
* sem\_wait decrements, blocks if zero.
* sem\_post increments, wakes a waiting thread if count was zero.
* Binary semaphore = lock.

**5.1.9 Condition Variables**

**Explanation:** Condition variables are synchronization primitives used with mutexes to allow threads to wait for a certain condition to become true. They don't have a counter like semaphores but act as a rendezvous point for threads.

**Key Concepts:**

* **Condition Variable:** A synchronization primitive used with a mutex to block threads until a condition is met.
* **Mutex:** A lock associated with a condition variable, required to protect the shared state related to the condition.
* pthread\_cond\_init: Initializes a condition variable.
* pthread\_cond\_wait: Atomically releases the mutex and blocks the calling thread until the condition variable is signaled. When the thread wakes up, it reacquires the mutex.
* pthread\_cond\_signal: Wakes up at least one thread waiting on the condition variable.
* pthread\_cond\_broadcast: Wakes up all threads waiting on the condition variable.
* **Lost Wakeup Problem:** A signal is sent to a condition variable when no threads are waiting, and the signal is lost.

**Code Blocks:**

* **Listing 5.9: Condition variables in pthreads**
  + This code snippet shows the basic usage of condition variables with a mutex.
  + pthread\_mutex\_t mlock;: Declares a mutex.
  + pthread\_cond\_t cond;: Declares a condition variable.
  + pthread\_cond\_init(&count, NULL);: Initializes the condition variable. (Note: The code snippet shows &count which seems like a typo in the source and should likely be &cond).
  + pthread\_mutex\_lock(&mlock);: Acquires the mutex before waiting.
  + pthread\_cond\_wait(&cond, &mlock);: Waits on the condition variable, atomically releasing and reacquiring the mutex.
  + pthread\_mutex\_unlock(&mlock);: Releases the mutex after waking up.
  + pthread\_mutex\_lock(&mlock);: Acquires the mutex before signaling.
  + pthread\_cond\_signal(&cond);: Signals the condition variable.
  + pthread\_mutex\_unlock(&mlock);: Releases the mutex after signaling.

**Quick Revision:**

* Condition variables are used with mutexes to wait for conditions.
* pthread\_cond\_wait blocks and releases mutex.
* pthread\_cond\_signal/broadcast wake waiting threads.
* Condition variables have no memory (can suffer from lost wakeups).

**5.1.10 Reader-Writer Lock**

**Explanation:** A reader-writer lock is a synchronization primitive that differentiates between read (non-modifying) and write (modifying) operations. It allows multiple readers to access a shared resource concurrently but requires exclusive access for writers.

**Key Concepts:**

* **Reader-Writer Lock:** A lock that allows multiple concurrent readers or a single exclusive writer.
* **Read Lock:** Acquired by readers; allows multiple threads to hold it simultaneously.
* **Write Lock:** Acquired by writers; requires exclusive access (no other readers or writers).

**Code Blocks:**

* **Listing 5.10: Code of the reader-writer lock**
  + This pseudocode outlines the implementation of a reader-writer lock using two internal mutexes: \_\_rwlock(for both readers and writers) and \_\_rdlock (for readers).
  + readers: A shared counter for the number of active readers, protected by \_\_rdlock.
  + get\_write\_lock(): Acquires the \_\_rwlock, ensuring exclusive access for a writer.
  + release\_write\_lock(): Releases the \_\_rwlock.
  + get\_read\_lock(): Acquires \_\_rdlock to update the readers count. If readers is 0, it also acquires \_\_rwlock to block writers. Increments readers and releases \_\_rdlock.
  + release\_read\_lock(): Acquires \_\_rdlock, decrements readers. If readers becomes 0, it releases \_\_rwlock to allow writers to proceed. Releases \_\_rdlock.

**Challenges with Reader-Writer Locks:** A basic reader-writer lock implementation can lead to starvation of writers if there is a continuous stream of readers.

**Quick Revision:**

* Allows multiple readers or one writer.
* Uses separate read and write locks.
* Can suffer from writer starvation.

**5.1.11 Barriers and Phasers**

**Explanation:** Barriers and phasers are synchronization primitives used to coordinate the execution of multiple threads, ensuring that they all reach a certain point before any are allowed to proceed.

**Key Concepts:**

* **Barrier:** A synchronization point that all threads in a group must reach before any of them can continue execution beyond the barrier. Useful in parallel computations like map-reduce.
* **Phaser:** A more flexible barrier that allows defining multiple synchronization points. Threads must reach an earlier point before being allowed to proceed past a later point. Useful for pipelined computations.

**Diagrams:**

* Figure 5.12: Illustrates the concept of a barrier.
* Figure 5.13: Illustrates the concept of phasers with two synchronization points.

**Quick Revision:**

* Barriers make threads wait at a point.
* Phasers allow synchronization across multiple points.

**5.2 Queues**

**Explanation:** Queues are important data structures in operating systems, often used for communication between different parts of the system. Bounded queues have a maximum size and are often implemented using a circular buffer with head and tail pointers.

**Key Concepts:**

* **Bounded Queue:** A queue with a limited capacity.
* **Circular Buffer:** An array used to implement a queue, where the head and tail pointers wrap around.
* **Head:** Pointer to the front of the queue (dequeue from here).
* **Tail:** Pointer to the back of the queue (enqueue here).
* INC(x): Macro for incrementing a pointer with wraparound ((x+1) % BUFSIZE).
* **Queue Empty Condition:** tail == head.
* **Queue Full Condition:** INC(tail) == head.

**Diagrams:**

* Figure 5.14: Conceptual view of a bounded queue with producers and consumers.

**5.2.1 Wait-Free Queue**

**Explanation:** A wait-free queue guarantees that every enqueue and dequeue operation completes within a bounded number of steps, even with concurrent access. A simple wait-free queue can be implemented with a single dedicated enqueuer and a single dedicated dequeuer using atomic operations, without needing locks.

**Code Blocks:**

* **Listing 5.11: A simple wait-free queue with one enqueuer and one dequeuer**
  + This C code implements a wait-free bounded queue for a single enqueuer and single dequeuer.
  + BUFSIZE: Defines the size of the circular buffer.
  + INC(x): Macro for calculating the next index in the circular buffer.
  + atomic\_int queue[BUFSIZE];: The shared buffer, declared as an array of atomic integers.
  + atomic\_int head = 0, tail = 0;: Atomic head and tail pointers.
  + enq(int val): Enqueues a value. It atomically loads head and tail, checks if the queue is full, atomically stores the value in the queue, and atomically updates the tail pointer.
  + deq(): Dequeues a value. It atomically loads head and tail, checks if the queue is empty, atomically loads the value from the queue, and atomically updates the head pointer.
  + enqfunc() and deqfunc(): Functions executed by the enqueuer and dequeuer threads, respectively. They repeatedly call enq and deq.
  + main(): Creates two threads, one for enqueueing and one for dequeueing.

**Quick Revision:**

* Guarantees bounded completion time for all operations.
* Can be implemented without locks using atomics for simple cases (single enqueuer/dequeuer).

**5.2.2 Queue with Mutexes**

**Explanation:** A bounded queue supporting multiple concurrent enqueuers and dequeuers can be implemented using a mutex to protect the critical sections (enqueue and dequeue operations).

**Code Blocks:**

* **Listing 5.12: A queue with mutexes**
  + This code shows a mutex-protected bounded queue.
  + pthread\_mutex\_t qlock;: The mutex protecting the queue.
  + LOCK(x) and UNLOCK(x): Macros wrapping pthread\_mutex\_lock and pthread\_mutex\_unlock.
  + enq(int val): Enqueues a value. It acquires qlock, checks for full condition, adds the value, updates tail, and releases qlock. It loops if the queue is full.
  + deq(): Dequeues a value. It acquires qlock, checks for empty condition, retrieves the value, updates head, and releases qlock. It loops if the queue is empty.
  + main(): Initializes and destroys the mutex.

**Quick Revision:**

* Uses a single mutex to protect enqueue and dequeue.
* Suitable for multiple concurrent threads.

**5.2.3 Queue with Semaphores**

**Explanation:** A bounded queue can also be implemented using semaphores. A binary semaphore can be used as a simple lock to protect the shared queue data structure.

**Code Blocks:**

* **Listing 5.13: A queue with semaphores**
  + This code shows a bounded queue using a semaphore as a binary lock.
  + sem\_t qlock;: The semaphore used as a lock.
  + sem\_init(&qlock, 0, 1);: Initializes the semaphore with a count of 1 (binary semaphore), shared between threads of a process (0).
  + LOCK(x) and UNLOCK(x): Macros wrapping sem\_wait and sem\_post.
  + The enq and deq functions are similar to the mutex version, using LOCK and UNLOCK around the critical sections.

**Quick Revision:**

* Uses a binary semaphore as a lock.
* Similar logic to the mutex-based queue.

**5.2.4 Queue with Semaphores but No Busy Waiting**

**Explanation:** To avoid busy waiting in a semaphore-based queue, additional semaphores can be used to signal when the queue has empty slots or filled slots. This allows threads to block when the queue is full (enqueuers) or empty (dequeuers) and be woken up when the condition changes.

**Key Concepts:**

* empty semaphore: Initialized to the buffer size; decremented on enqueue, incremented on dequeue. Threads wait on this when the queue is full.
* full semaphore: Initialized to 0; incremented on enqueue, decremented on dequeue. Threads wait on this when the queue is empty.
* qlock semaphore: A binary semaphore (lock) to protect access to the shared queue data structure itself.

**Code Blocks:**

* **Listing 5.14: A queue with semaphores but does not have busy waiting**
  + This code implements a bounded queue using three semaphores to avoid busy waiting.
  + sem\_t qlock, empty, full;: Declares the three semaphores.
  + sem\_init(&qlock, 0, 1);: Initializes qlock as a binary semaphore.
  + sem\_init(&empty, 0, BUFSIZE);: Initializes empty to the buffer size.
  + sem\_init(&full, 0, 0);: Initializes full to 0.
  + WAIT(x) and POST(x): Macros wrapping sem\_wait and sem\_post.
  + enq(int val): Waits on empty (checks for space), waits on qlock (enters critical section), enqueues the value, posts qlock (exits critical section), and posts full (signals a slot is filled).
  + deq(): Waits on full (checks for data), waits on qlock (enters critical section), dequeues the value, posts qlock (exits critical section), and posts empty (signals a slot is empty).

**Quick Revision:**

* Uses empty and full semaphores to manage queue state and block threads.
* Avoids busy waiting by blocking threads in the kernel's wait queue.

**5.2.5 Reader-Writer Lock (in Queue Context)**

**Explanation:** Introducing a read-only operation like peak (reading the head without removing it) to the queue makes a reader-writer lock a suitable synchronization mechanism. Multiple threads can peak concurrently (readers), but enq and deq (writers) require exclusive access.

**Code Blocks:**

* **Listing 5.15: A queue with reader-writer locks**
  + This code shows a bounded queue using a reader-writer lock for synchronization, in addition to the emptyand full semaphores.
  + peak(): Acquires the read lock using get\_read\_lock(), checks if the queue is empty, reads the value at the head, and releases the read lock using release\_read\_lock().
  + enq(int val): Waits on the empty semaphore, acquires the write lock using get\_write\_lock(), enqueues the value, releases the write lock using release\_write\_lock(), and posts the full semaphore.
  + deq(): Waits on the full semaphore, acquires the write lock using get\_write\_lock(), dequeues the value, releases the write lock using release\_write\_lock(), and posts the empty semaphore.

**Quick Revision:**

* Uses a reader-writer lock to allow concurrent peak (read) operations.
* enq and deq (write) operations require the exclusive write lock.

**5.3 Concurrency within the Kernel**

**Explanation:** Concurrency is a critical concern within the operating system kernel, especially in symmetric multiprocessor (SMP) systems where multiple kernel threads share data structures and resources. Ensuring correctness and freedom from data races, deadlocks, livelocks, and starvation is paramount.

**5.3.1 Kernel-Level Locking: Spinlocks**

**Explanation:** Kernel code often uses spinlocks for synchronization. Unlike user-space mutexes, kernel spinlocks are typically held by the CPU and require disabling interrupts within the critical section to prevent the lock-holding thread from being preempted. This ensures that the lock holder completes the critical section quickly and is lock-free (assuming no deadlocks). Allowing preemption while holding a spinlock can lead to deadlocks if a higher-priority thread waiting for the lock preempts the lock holder.

**Key Concepts:**

* **Kernel Spinlock:** A lock used in the kernel that involves busy waiting and typically requires disabling interrupts on the CPU holding the lock.
* **Symmetric Multiprocessor (SMP):** A system with multiple identical processors sharing memory and I/O.

**Code Blocks:**

* **Listing 5.17: Wrapper of a spinlock**
  + This shows the structure of a raw spinlock in the Linux kernel.
  + raw\_spinlock\_t: The type for a raw spinlock.
  + arch\_spinlock\_t raw\_lock;: The architecture-specific spinlock implementation.
  + struct lockdep\_map dep\_map;: Used for deadlock detection (optional).
* **Listing 5.18: Inner workings of a spinlock**
  + This code shows the implementation of a ticket lock, a type of spinlock used in the kernel.
  + **Ticket Lock:** A fair spinlock where threads acquire a "ticket" and wait for their turn (ticket number to match the "next" serving number).
  + A single 32-bit integer stores both the ticket (upper 16 bits) and the next serving number (lower 16 bits).
  + atomic\_fetch\_add(1<<16, lock): Atomically increments the ticket number (adds 216) and returns the old value, which is used to determine the thread's ticket.
  + atomic\_cond\_read\_acquire(lock, ticket == (u16)VAL): Busy waits until the thread's ticket matches the next serving number (lower 16 bits of the lock value).
  + smp\_mb(): A memory barrier (fence) to prevent instruction reordering.
* **Listing 5.19: The code for the busy-wait loop**
  + This macro implements the busy-wait loop used in the spinlock.
  + for (;;): An infinite loop for busy waiting.
  + VAL = READ\_ONCE(\*\_\_PTR);: Reads the current value of the lock variable without allowing the compiler to reorder this read.
  + if (cond\_expr) break;: Checks the condition (ticket == (u16)VAL); if true, breaks the loop (lock acquired).
  + cpu\_relax();: Inserts a delay and potentially gives hints to the hardware to reduce power consumption while busy waiting.
* **Listing 5.20: The code for unlocking a spinlock**
  + This code shows how a ticket spinlock is unlocked.
  + It calculates the address of the "next" serving number field (lower 16 bits) considering endianness.
  + smp\_store\_release(ptr, (u16)val + 1);: Atomically increments the "next" serving number and stores it, using release consistency semantics which includes a memory barrier to ensure writes within the critical section are visible.

**Fast Path / Slow Path:** Spinlock acquisition often uses a fast path for the common case of no contention and a slow path for when contention exists. The fast path attempts to acquire the lock quickly (e.g., using atomic compare-and-exchange), and if unsuccessful, falls back to the slower, potentially busy-waiting, path.

**Code Blocks:**

* **Listing 5.21: The code to try to acquire a spinlock (fast path)**
  + This function implements the fast path for spinlock acquisition.
  + atomic\_read(lock);: Reads the current lock value.
  + if ((old >> 16) != (old & 0xffff)) return false;: Quickly checks if the ticket and next numbers are different (lock is busy); if so, returns false.
  + atomic\_try\_cmpxchg(lock, &old, old + (1<<16));: Attempts to atomically compare the lock value with old and, if they match, update it by incrementing the ticket number (adds 216). Returns true on success (lock acquired), false otherwise.

**Quick Revision:**

* Kernel spinlocks use busy waiting and disable interrupts.
* Ticket locks are a fair type of spinlock.
* Fast path/slow path optimizes lock acquisition.

**5.3.2 Kernel Mutexes**

**Explanation:** Kernel mutexes are different from spinlocks in that they are held by a task rather than a CPU and allow the task to sleep if the mutex is unavailable, avoiding busy waiting. They include a waiting list for blocked tasks.

**Key Concepts:**

* **Kernel Mutex:** A synchronization primitive in the kernel that allows tasks to block when the mutex is unavailable.
* **owner field:** Pointer to the task holding the mutex (or 0 if free).
* **wait\_list:** A list of tasks waiting to acquire the mutex.
* **wait\_lock:** A spinlock protecting the wait\_list.

**Code Blocks:**

* **Listing 5.22: A kernel mutex**
  + This shows the structure of a kernel mutex.
  + atomic\_long\_t owner;: Stores the owner task pointer (or 0). Made atomic for fast-path acquisition attempts.
  + raw\_spinlock\_t wait\_lock;: Spinlock protecting the waiting list.
  + struct list\_head wait\_list;: The list of waiting tasks.
  + struct lockdep\_map dep\_map;: For deadlock detection (optional).
* **Listing 5.23: The mutex\_lock operation**
  + This function attempts to acquire a kernel mutex.
  + might\_sleep();: Checks if the current context allows sleeping; flags an error if not.
  + \_\_mutex\_trylock\_fast(lock);: Attempts the fast-path acquisition (atomic compare-and-exchange on the owner field).
  + \_\_mutex\_lock\_slowpath(lock);: If the fast path fails, calls the slow-path function which involves acquiring the wait\_lock, adding the task to wait\_list, and putting the task to sleep.

**Quick Revision:**

* Kernel mutexes allow tasks to sleep when waiting.
* Use a fast path (atomic check) and a slow path (waiting list).

**5.3.3 Kernel Semaphores**

**Explanation:** The Linux kernel also has its own implementation of semaphores, similar in concept to user-level semaphores but designed for use within the kernel context. They have a count, a spinlock to protect the count and waiting list, and functions for waiting (down) and signaling (up).

**Key Concepts:**

* **Kernel Semaphore:** A semaphore implemented within the Linux kernel.
* down: The wait operation (decrements count, blocks if zero).
* up: The post/signal operation (increments count, wakes a waiting task if count was zero).
* count: The semaphore's integer count.
* lock: A spinlock protecting the count and wait\_list.
* wait\_list: A list of tasks waiting for the semaphore.

**Code Blocks:**

* **Listing 5.24: The kernel semaphore**
  + This shows the structure of a kernel semaphore.
  + raw\_spinlock\_t lock;: Spinlock protecting the semaphore.
  + unsigned int count;: The semaphore count.
  + struct list\_head wait\_list;: The list of waiting tasks.

**Quick Revision:**

* Kernel version of semaphores.
* down (wait) and up (signal).
* Uses a spinlock to protect its internal state.

**5.3.4 The Lockdep Mechanism**

**Explanation:** Lockdep is a kernel mechanism for validating lock usage and detecting potential deadlocks. It checks for issues like excessive lock nesting depth, trivial deadlocks (lock inversions), and potential deadlocks arising from interrupt contexts. For the general case, it uses a graph-based approach to detect cycles (circular waits) in lock dependencies. To improve performance, it uses a caching mechanism (hash table) to store the deadlock status of previously encountered lock acquisition chains.

**Key Concepts:**

* **Lockdep:** The Linux kernel's lock validator and deadlock detection mechanism.
* **Lock Depth:** The number of locks currently held by a task.
* **Lock Inversion:** A trivial deadlock where two locks are acquired in different orders by different threads (e.g., acquire A then B, and acquire B then A).
* **Circular Wait:** A cycle in the lock dependency graph, indicating a potential deadlock.
* **Interrupt Context:** Code executing within an interrupt handler, where preemption rules can differ.
* **Hash Table:** Used by Lockdep to cache the deadlock status of lock chains.

**Diagrams:**

* Figure 5.15: Illustrates the hash table used by Lockdep to cache lock chain statuses.

**Quick Revision:**

* Detects potential deadlocks in the kernel.
* Checks lock depth, lock inversions, and circular waits.
* Uses caching for performance.

**5.3.5 The RCU (Read-Copy-Update) Mechanism**

**Explanation:** RCU is a synchronization mechanism optimized for read-heavy workloads in the kernel. It allows multiple readers to access shared data structures concurrently with minimal overhead. Updates are done by creating a copy, modifying the copy, and then atomically publishing the new version by changing a pointer. The key challenge is determining when it is safe to reclaim the memory of the old version, which is done after a "grace period" during which all active readers of the old version have finished.

**Key Concepts:**

* **RCU (Read-Copy-Update):** A synchronization mechanism for read-mostly data structures.
* **Read-Side Critical Section:** A code region where a thread reads RCU-protected data, typically protected by disabling and enabling preemption.
* **Grace Period:** A time interval after an update during which all readers active before the update are guaranteed to finish.
* **Quiescent State:** A state where a CPU is not executing within an RCU read-side critical section.
* synchronize\_rcu(): A function that waits for a grace period to elapse.
* rcu\_read\_lock(): Enters an RCU read-side critical section (typically disables preemption).
* rcu\_read\_unlock(): Exits an RCU read-side critical section (typically enables preemption).
* rcu\_assign\_pointer(): Atomically updates an RCU-protected pointer, often including checks and memory barriers.
* rcu\_dereference(): Reads an RCU-protected pointer, typically including memory barriers.

**Code Blocks:**

* **Listing 5.25: Example code that traverses a list within an RCU read context**
  + This shows how to traverse an RCU-protected linked list.
  + rcu\_read\_lock();: Enters the RCU read-side critical section.
  + list\_for\_each\_entry\_rcu(...): A macro for iterating through an RCU-protected list.
  + rcu\_read\_unlock();: Exits the RCU read-side critical section.
* **Listing 5.26: Replace an item in a list and then wait till all the readers finish**
  + This shows how to update an item in an RCU-protected list and wait for readers to finish before freeing the old item.
  + list\_replace\_rcu(...): Replaces an item in the list.
  + synchronize\_rcu();: Waits for the grace period to end.
  + kfree(p);: Frees the memory of the old item.
* **Listing 5.27: Assign an RCU-protected pointer**
  + This macro (simplified) shows how RCU-protected pointers are assigned.
  + It involves checks and memory barriers to ensure proper synchronization and visibility.
* **Listing 5.28: Implementation of the list\_replace\_rcu function**
  + This shows a simplified implementation of replacing a node in an RCU-protected doubly linked list.
  + rcu\_assign\_pointer(...): Used to atomically update the list pointers in an RCU-safe manner.
* **Listing 5.29: Code to dereference an RCU pointer**
  + This macro (simplified) shows how RCU-protected pointers are dereferenced.
  + READ\_ONCE(p): Reads the pointer value without allowing the compiler to reorder the read.
* **Listing 5.30: Example that uses the RCU dereference operation**
  + This shows an example of iterating through an RCU-protected list using rcu\_dereference.
* **Listing 5.31: Code to run tasks on each CPU**
  + This simplified pseudocode illustrates a basic way to wait for all CPUs to become quiescent by running a task on each. More efficient mechanisms exist in real RCU implementations.

**Diagrams:**

* Figure 5.14: Deleting a linked list node using RCU, illustrating the removal, synchronization, and reclamation steps.
* Figure 5.17: Depicts the removal and reclamation process in RCU with the grace period.
* Figure 5.18: Shows the structure of Tree RCU, an efficient implementation that uses a tree to track the quiescent state of CPUs.

**Preemptible RCU:** Standard RCU disables preemption in read-side critical sections. Preemptible RCU allows preemption within read blocks, which is important for real-time systems but adds complexity to tracking quiescent states.

**Quick Revision:**

* Optimized for read-heavy workloads.
* Updates use copy-on-write.
* Grace period allows safe memory reclamation.
* Readers are low-overhead (often just disabling preemption).
* synchronize\_rcu waits for grace period.
* Preemptible RCU for real-time systems adds complexity.

**5.4 Scheduling**

**Explanation:** Scheduling is a core operating system function responsible for deciding which tasks run on which CPU cores and when. The goal is to efficiently utilize computing resources and meet various performance objectives.

**5.4.1 Space of Scheduling Problems**

**Explanation:** Scheduling problems involve scheduling a set of jobs with arrival times and durations onto a set of cores, aiming to optimize a specific objective function. The complexity varies based on factors like whether job durations are known, if preemption is allowed, arrival time characteristics, and job dependencies.

**Key Concepts:**

* **Job:** A task or unit of work to be scheduled.
* **Arrival Time:** The time a job becomes available for scheduling.
* **Duration (Processing Time):** The time required to execute a job.
* **Preemption:** The ability to interrupt a running job and reschedule it later.
* **Makespan:** The total time from the start of scheduling until the last job completes (minimizing this is a common objective).
* **Completion Time:** The time a job finishes execution (relative to its arrival time).
* **Mean Job Completion Time:** The average completion time of all jobs (minimizing this is an objective related to system responsiveness).
* **Weighted Completion Time:** A sum of completion times weighted by job priorities.
* **Dependencies:** Constraints where one job must complete before another can start.
* **Deadlines:** A time by which a job must be completed.

**Diagrams:**

* Figure 5.19: Example of a set of jobs with known processing times awaiting scheduling.
* Figure 5.20: Illustrates the concept of mean completion time.

**KSW Model:** The KSW model is a framework for classifying scheduling problems using a 3-tuple (α | β | γ).

* **α (Machine Environment):** Specifies the number of cores, jobs, and their execution times.
* **β (Constraints):** Defines restrictions like preemption (allowed or not), arrival times (same or different), dependencies, and deadlines.
* **γ (Optimality Criterion):** The objective function to be optimized (e.g., minimizing makespan, mean completion time, or weighted completion time).

**NP-Complete Problems:** Many scheduling problems, especially non-preemptive variants with different arrival times, are NP-complete, meaning finding an optimal solution is computationally difficult. These problems are related to classical NP-complete problems like bin packing and set partition.

**Key Concepts:**

* **NP-Complete:** A class of problems for which no known polynomial-time algorithm exists.
* **Bin Packing Problem:** Packing items of different sizes into a minimum number of bins with fixed capacity.
* **Set Partition Problem:** Dividing a set of numbers into subsets with a specific sum.

**Predicting CPU Burst Lengths:** In real systems, job execution times are often unknown. Scheduling algorithms may rely on predicting the length of the next CPU burst (a period of CPU activity between I/O waits) based on historical data, often using time-series analysis techniques.

**Diagrams:**

* Figure 5.22: Illustrates CPU and I/O bursts in a task's execution.

**Conventional Algorithms (without known execution times):**

* **FIFO (First-In, First-Out):** Schedules jobs in the order they arrive. Simple but can suffer from the "convoy effect" where a long job delays many short ones.
* **Round-Robin:** Each job runs for a fixed time quantum, then the CPU switches to the next job. Offers better fairness than FIFO for short jobs.

**Fairness vs. Priority:** There is a trade-off between giving priority to certain tasks and ensuring fairness among all tasks.

**Diagrams:**

* Figure 5.23: Illustrates the trade-off between fairness and priority.

**Queue-based Scheduling (Multilevel Feedback Queue):** This approach uses multiple queues with different priorities. The scheduler picks tasks from the highest-priority non-empty queue. Tasks can migrate between queues based on their behavior (e.g., interactive tasks move to higher priority queues) to balance performance and fairness.

**Diagrams:**

* Figure 5.24: Illustrates a multi-level feedback queue.

**Quick Revision:**

* Scheduling involves mapping jobs to cores to meet objectives.
* Problems vary based on known durations, preemption, arrival times, dependencies, deadlines.
* Makespan and mean completion time are common objectives.
* Some problems are NP-complete.
* Can predict CPU bursts using time-series analysis.
* FIFO and Round-Robin are simple algorithms.
* Multilevel feedback queues balance priority and fairness.

**5.4.2 Single Core Scheduling**

**Explanation:** This section focuses on scheduling algorithms for a single processor, examining how to optimize specific criteria like minimizing total completion time or maximum lateness under various constraints.

**Key Concepts:**

* **Shortest Job First (SJF):** A non-preemptive algorithm that minimizes mean completion time by scheduling the shortest jobs first.
* **Weighted Shortest Job First (WSJF):** Schedules jobs based on the ratio of their weight to their processing time, minimizing weighted completion time.
* **Earliest Deadline First (EDF):** A preemptive algorithm that minimizes maximum lateness by scheduling the job with the earliest deadline first.
* **Shortest Remaining Time First (SRTF):** A preemptive algorithm that minimizes mean completion time by scheduling the job with the shortest remaining execution time.

**Diagrams:**

* Figure 5.21: Illustrates Shortest Job First scheduling.

**KSW Notation Examples:**

* 1∣∣∑Cj : Single core, no constraints, minimize sum of completion times. SJF is optimal for this.
* 1∣∣∑wj Cj : Single core, no constraints, minimize weighted sum of completion times. Scheduling by wj /pj in descending order is optimal.
* 1∣ri ,dli ,pmtn∣Lmax : Single core, different arrival times, deadlines, preemption allowed, minimize maximum lateness. EDF is optimal for this if the jobs are schedulable.
* 1∣ri ,pmtn∣∑Ci : Single core, different arrival times, preemption allowed, minimize sum of completion times. SRTF is optimal.
* 1∣ri ∣∑Ci : Single core, different arrival times, no preemption, minimize sum of completion times. NP-complete.
* 1∣ri ∣Lmax : Single core, different arrival times, no preemption, minimize maximum lateness. NP-complete.
* 1∣ri ,pmtn∣∑wi Ci : Single core, different arrival times, preemption allowed, minimize weighted sum of completion times. NP-complete for generic weights.

**Quick Revision:**

* SJF minimizes mean completion time (non-preemptive).
* SRTF minimizes mean completion time (preemptive).
* EDF minimizes max lateness (preemptive, deadlines).
* Many single-core problems are NP-complete when preemption is not allowed or with weighted objectives.

**5.4.3 Multicore Scheduling**

**Explanation:** Multicore scheduling involves distributing tasks across multiple CPU cores, aiming to minimize objectives like makespan. Preemptive multicore scheduling problems are generally easier to solve than non-preemptive ones.

**Key Concepts:**

* **P || pmtn | Cmax:** P processors, preemption allowed, minimize makespan. Simple solution: distribute work evenly.
* **P || Cmax:** P processors, no preemption, minimize makespan. NP-complete.
* **List Scheduling:** A non-preemptive algorithm where ready jobs are kept in a prioritized list. When a CPU is free, it takes the highest-priority executable job from the list.
* **Critical Path:** The longest path in a job dependency graph, which often determines the minimum possible makespan. Scheduling jobs on the critical path is prioritized.
* **LPT (Longest Processing Time First):** A list scheduling heuristic where jobs are prioritized by their processing time in descending order. Provides a bound on the makespan relative to the optimal.

**Theorem 5.4.1 Makespan:** For list scheduling on m CPUs, regardless of the priority scheme, the makespan (Cmax ) is bounded by Cmax ≤(2−1/m)C∗ where C∗ is the optimal makespan.

**Formula:** The bound for list scheduling makespan: Cmax ≤m∑pi +pk (1−m1 )≤C∗+C∗(1−m1 ) where ∑pi is the total processing time, m is the number of CPUs, and pk is the processing time of the last completing job. This simplifies to Cmax /C∗≤2−1/m.

**Quick Revision:**

* Preemptive multicore scheduling is easier than non-preemptive.
* Non-preemptive makespan minimization is NP-complete.
* List scheduling is a common non-preemptive approach.
* Critical path jobs are important for makespan.
* List scheduling makespan is bounded relative to the optimal.

**5.4.4 Banker’s Algorithm**

**Explanation:** The Banker's algorithm is a deadlock avoidance algorithm that can handle multiple instances of resources. It determines if a system is in a "safe state" by simulating hypothetical resource allocations to see if all processes can complete without causing a deadlock. If a resource request would lead to an unsafe state, the request is denied.

**Key Concepts:**

* **Deadlock Avoidance:** Preventing deadlocks by checking the safety of resource requests before granting them.
* **Safe State:** A system state where it is possible to satisfy the resource requests of all processes in some order, thus avoiding deadlock.
* **Unsafe State:** A system state where a deadlock is potentially possible.
* **Multiple Copies of Resources:** The Banker's algorithm handles scenarios where there is more than one unit of each resource type.

**Data Structures:**

* avlbl[m]: Number of available copies for each of the m resource types.
* max[n][m]: Maximum number of copies of each resource type that each of the n processes may request.
* acq[n][m]: Number of copies of each resource type currently acquired by each process.
* need[n][m]: Number of additional copies of each resource type that each process may still need (need = max - acq).
* req[m]: The current resource request of a specific process.

**Diagrams:**

* Figure 5.26: Illustrates a circular dependency with multiple resource copies, where a deadlock does not occur.

**Algorithms:**

* **Algorithm 1: Algorithm to check for the safety of the system**
  + Initializes cur\_cnt (available resources) and done array (tracks if a process's needs can be met).
  + Iteratively finds a process whose current resource need can be satisfied by cur\_cnt.
  + If such a process is found, assumes it completes, adds its acquired resources (acq) to cur\_cnt, and marks it as done.
  + Repeats until no such process is found.
  + Finally, checks if all processes are marked as done. If so, the state is safe; otherwise, it's unsafe.
* **Algorithm 2: Algorithm to request for resources**
  + Takes a resource request (req) from a process.
  + Checks if the request exceeds the process's maximum declared need. If so, denies the request.
  + Checks if the request can be satisfied by the currently avlbl resources. If not, the process waits.
  + Hypothetically allocates the resources (updates avlbl, acq, need).
  + Calls Algorithm 1 to check if this hypothetical allocation results in a safe state. If not, the original request is denied.
* **Algorithm 3: Algorithm for finding deadlocks**
  + Initializes cur\_cnt and done array (tracks if a process's current reqs can be met).
  + Iteratively finds a process whose current resource reqs can be satisfied by cur\_cnt.
  + If such a process is found, assumes it completes, adds its acquired resources (acq) to cur\_cnt, and marks it as done.
  + Repeats until no such process is found.
  + Finally, checks if all processes are marked as done. If not, a deadlock is detected.

**Quick Revision:**

* Deadlock avoidance for multiple resource copies.
* Checks if a state is safe (all processes can eventually finish).
* Resource requests are granted only if they lead to a safe state.
* Can also be used for deadlock detection.

**5.4.5 Scheduling in the Linux Kernel**

**Explanation:** The Linux kernel has a sophisticated scheduling subsystem responsible for managing tasks (kernel threads and user processes) on CPU cores. The main entry point is the schedule() function. Linux employs different scheduling classes and algorithms to handle tasks with varying priorities and requirements.

**Key Concepts:**

* schedule(): The main function for invoking the scheduler.
* kworker threads: Kernel threads that perform deferred work.
* Runqueue (struct rq): A per-CPU data structure holding tasks scheduled to run on that CPU.
* **Scheduling Classes:** A hierarchy of classes representing different types of tasks and their associated schedulers (e.g., deadline, real-time, fair).
* struct sched\_class: A generic structure defining function pointers for scheduler-specific operations.
* pick\_task() / pick\_next\_task(): Functions within a scheduling class to select the next task to run.
* cgroups: Control groups for grouping processes and managing their resource allocation, including scheduling.
* core\_node: The preferred CPU for a task (relevant in NUMA systems).
* core\_cookie: Identifies a group of tasks that can safely run on the same core due to security considerations.

**Code Blocks:**

* **Listing 5.32: The schedule function**
  + Shows the basic structure of the schedule() function.
  + It may dispatch work to other kernel threads, disables preemption, calls the internal \_\_schedule()function, and updates worker thread status.
* **Listing 5.33: List of important functions in struct sched\_class**
  + Lists function pointers for common scheduling operations: enqueue/dequeue tasks, pick next task, migrate task, update current task statistics.
* **Listing 5.34: The runqueue**
  + Shows the structure of a runqueue (struct rq).
  + Includes a spinlock (\_\_lock) for protection, basic CPU statistics, pointers to scheduler-specific runqueues (CFS, RT, DL), and pointers to the current and idle tasks.
* **Listing 5.35: Scheduling-related fields in the task struct**
  + Shows fields in the task structure related to scheduling, including per-scheduler statistics structures (sched\_entity, sched\_rt\_entity, sched\_dl\_entity), a pointer to the task's scheduling class, and fields for preferred CPU and core cookie.

**Quick Revision:**

* Kernel uses scheduling classes for different task types.
* Runqueues hold tasks for each CPU.
* schedule() is the main scheduler function.
* pick\_task / pick\_next\_task select the next task.
* cgroups manage resource allocation for process groups.
* core\_cookie for security-based task grouping on cores.

**5.4.6 Completely Fair Scheduling (CFS)**

**Explanation:** CFS is the default scheduler for regular processes in Linux, aiming for fairness by ensuring every runnable task gets a proportional share of CPU time. It uses the concept of "virtual runtime" (vruntime) to track how long each task has run, scaled by its priority (nice value). Tasks with lower vruntime are prioritized. Tasks are organized in a red-black tree within the runqueue, sorted by vruntime.

**Key Concepts:**

* **Completely Fair Scheduling (CFS):** Default scheduler for regular processes, aims for fairness.
* **Virtual Runtime (vruntime):** A time counter for each task, scaled by its priority. Lower vruntime indicates a task has received less CPU time relative to its priority.
* **Nice Value:** A user-adjustable priority value for processes (lower nice value = higher priority).
* **Red-Black Tree:** A balanced binary search tree used to store runnable tasks in the CFS runqueue, sorted by vruntime.
* **Scheduling Period:** A time interval during which CFS aims to give all runnable tasks a chance to execute.
* **Scheduling Slice:** The amount of time a task is allowed to run within a scheduling period, proportional to its weight (priority).
* **min\_vruntime:** The lowest vruntime among all tasks in a CFS runqueue, used to prevent new or waking tasks from having an unfair advantage.

**Code Blocks:**

* **Listing 5.36: struct sched\_entity**
  + This structure contains scheduling-related information for CFS tasks, including vruntime, execution time statistics, and a pointer to the CFS runqueue.
* **Listing 5.37: weight as a function of the nice value**
  + This table shows the mapping between nice values and weights used by CFS. Lower nice values have higher weights, leading to a slower increase in vruntime.
* **Listing 5.38: Implementation of scheduling quanta in CFS**
  + This code shows how the scheduling period is determined based on the number of runnable tasks.
* **Listing 5.39: hrtick\_start\_fair**
  + This code snippet (simplified) shows how CFS determines how long a task should run based on its scheduling slice and already executed time.

**Formula:** The relationship between actual runtime (δ) and virtual runtime (δvruntime ) is δvruntime =δ×weight(nice)weight(nice=0) .

**Calculating CPU Load:** Linux calculates CPU load average using a decaying average over time, giving more weight to recent activity. This is used for load balancing decisions.

**Formula:** Load average formula using a decaying average: loadavg=u0 +u1 ×y+u2 ×y2+... where ui is the fraction of time a runnable task executed in interval pi , and y is a decay term.

**Quick Revision:**

* CFS provides fair CPU time sharing using virtual runtime.
* vruntime is scaled by task priority (nice value).
* Tasks are in a red-black tree sorted by vruntime.
* Scheduling period and slice determine how long tasks run.
* min\_vruntime prevents unfairness for new/waking tasks.
* CPU load calculation uses a decaying average.

**5.4.7 Real-Time and Deadline Scheduling**

**Explanation:** Linux supports real-time and deadline scheduling classes for tasks with strict timing requirements.

**Key Concepts:**

* **Deadline Scheduler:** Schedules tasks based on their deadlines, prioritizing the earliest deadline first (EDF algorithm). Aims to minimize maximum lateness.
* **Real-Time Scheduler:** Uses fixed priorities. Tasks are organized into queues based on their real-time priority. Higher priority queues are always scheduled before lower priority ones. Within the same priority, scheduling can be FIFO or Round-Robin.

**Quick Revision:**

* Deadline scheduler (EDF) prioritizes earliest deadline.
* Real-time scheduler uses fixed priorities and priority queues (FIFO or RR within priority).

**5.5 Real-Time Systems**

**Explanation:** Real-time systems have tasks with deadlines that must be met to ensure correctness and stability. They differ from general-purpose systems in their emphasis on predictability and timeliness.

**5.5.1 Types of Real-Time Systems**

**Explanation:** Real-time systems are categorized based on the consequences of missing a deadline.

**Key Concepts:**

* **Soft Real-Time Systems:** Missing a deadline is undesirable but does not cause system failure; utility of the result decreases after the deadline. Examples: audio/video processing.
* **Firm Real-Time Systems:** Missing a deadline makes the result useless (zero utility), but does not cause system failure. Examples: ATM machines, flight reservation systems.
* **Hard Real-Time Systems:** Missing a deadline is catastrophic and leads to system failure. Examples: missile guidance, medical devices, nuclear plant control.

**Task vs. Job:**

* **Task:** A single schedulable entity with potential time constraints.
* **Job:** May consist of a set of tasks, often arriving periodically.

**Schedulability:** Determining if a set of real-time jobs can be scheduled such that all tasks meet their deadlines.

**Quick Revision:**

* Soft real-time: Deadline misses decrease utility, no failure.
* Firm real-time: Deadline misses make result useless, no failure.
* Hard real-time: Deadline misses cause catastrophic failure.
* Task = schedulable entity, Job = set of tasks.
* Schedulability: Can all tasks meet deadlines?

**5.5.2 EDF Scheduling (in Real-Time Systems)**

**Explanation:** Earliest Deadline First (EDF) is an optimal preemptive scheduling algorithm for uniprocessor real-time systems. It prioritizes tasks with the earliest deadlines.

**Key Concepts:**

* **Optimal Scheduler:** An algorithm that can find a feasible schedule if one exists, or optimize a specific metric.
* **Utilization (U):** The sum of (task duration / period) for all periodic jobs. For EDF on a uniprocessor, if U≤1, the set of periodic jobs is schedulable.
* **Dynamic Priorities:** Priorities change over time based on deadlines.

**Formula:** Utilization U=∑Pi di where di is the duration and Pi is the period of job i.

**Quick Revision:**

* EDF is optimal for uniprocessor real-time.
* Prioritizes earliest deadline.
* Schedulable if Utilization (U) <= 1.
* Uses dynamic priorities.

**5.5.3 RMS Scheduling**

**Explanation:** Rate Monotonic Scheduling (RMS) is a preemptive scheduling algorithm for uniprocessor real-time systems with static priorities. Priority is inversely proportional to the job's period (higher frequency = higher priority).

**Key Concepts:**

* **Rate Monotonic Scheduling (RMS):** Static priority scheduling, priority is 1/Period.
* **Static Priorities:** Priorities are fixed at system startup.
* **Liu-Layland Bound:** A sufficient condition for schedulability for RMS on a uniprocessor. If U≤n(21/n−1)where n is the number of jobs, the system is schedulable.
* **Lehoczky Test:** A necessary and sufficient condition for RMS schedulability, more complex to compute than the Liu-Layland bound. It checks if each task meets its first deadline in the worst-case scenario (all tasks starting in phase).

**Formulas:**

* Liu-Layland Bound: U≤n(21/n−1).
* Lehoczky Test (simplified): Checks if ∃t∈(0,Pi ] such that Wi (t)/t≤1 for all tasks i, where Wi (t)=∑j=1i ⌈t/Pj ⌉×dj is the cumulative CPU load of the first i tasks in time t.

**Quick Revision:**

* RMS uses static priorities based on job periods.
* Liu-Layland bound: sufficient condition for schedulability.
* Lehoczky test: necessary and sufficient condition for schedulability.

**5.5.4 DMS Scheduling**

**Explanation:** Deadline Monotonic Scheduling (DMS) is a preemptive static priority algorithm for uniprocessor real-time systems where the deadline is not necessarily equal to the period (di <Pi ). Priority is inversely proportional to the task's deadline (earlier deadline = higher priority). Schedulability analysis considers the task's execution time and interference from higher-priority tasks.

**Key Concepts:**

* **Deadline Monotonic Scheduling (DMS):** Static priority scheduling where priority is based on the deadline (di <Pi ).
* **Interference (Ii (t)):** The total execution time of higher-priority tasks within a time interval t.

**Formula:** Interference of higher-priority tasks on task i in time t: Ii (t)=∑j=1i−1 ⌈t/Pj ⌉×dj where tasks are ordered by DMS priority.

**Algorithm 4: The DMS algorithm (Schedulability Test)**

* Iteratively checks if a task can complete within a certain time interval t, considering interference.
* Starts with an initial t (sum of execution times of higher-priority tasks + current task).
* In a loop, calculates interference Ii (t) and checks if Ii (t)+di ≤t. If true, the task is schedulable within this interval.
* If not, increases t to Ii (t)+di and repeats the check.
* If t exceeds the task's deadline (Di ), the task is not schedulable.

**Quick Revision:**

* DMS uses static priorities based on deadlines (for di <Pi ).
* Schedulability test considers interference from higher-priority tasks.

**5.5.5 Priority Inheritance Protocol (PIP)**

**Explanation:** Priority Inversion is a problem in real-time systems where a lower-priority task blocks a higher-priority task by holding a resource that the higher-priority task needs. The Priority Inheritance Protocol (PIP) aims to mitigate this by temporarily boosting the priority of the resource-holding low-priority task to the priority of the highest-priority task waiting for that resource.

**Key Concepts:**

* **Priority Inversion:** A low-priority task blocking a high-priority task by holding a needed resource.
* **Bounded Priority Inversion:** The blocking time is predictable.
* **Unbounded Priority Inversion:** The blocking time is unpredictable, potentially extended by medium-priority tasks preempting the low-priority resource holder.
* **Chain Blocking:** A task is blocked multiple times while trying to acquire a sequence of resources, potentially leading to long blocking times.
* **Priority Inheritance Protocol (PIP):** A protocol where a resource-holding task inherits the priority of the highest-priority task waiting for that resource.

**Definition 5.5.1 Priority Inversion:** A phenomenon where a low-priority task blocks a high-priority task because the former holds a resource that the latter is interested in.

**Definition 5.5.2 Unbounded Priority Inversion and Chain Blocking:**

* Unbounded priority inversion: A high-priority task is blocked for an unbounded time because the low-priority resource holder is continuously preempted by medium-priority tasks.
* Chain blocking: A task is blocked for a long time while acquiring multiple resources sequentially due to priority inversions for each resource.

**Issues with PIP:** While PIP prevents unbounded priority inversion, it can still suffer from deadlocks and chain blocking.

**Point 5.5.2:** Deadlocks and chain blocking are the major issues in the PIP protocol.

**Quick Revision:**

* Priority inversion: Low priority blocks high priority.
* PIP boosts resource holder's priority.
* PIP prevents unbounded priority inversion but not deadlocks or chain blocking.

**5.5.6 Highest Locker Protocol (HLP)**

**Explanation:** The Highest Locker Protocol (HLP) is another priority-based resource allocation protocol that aims to address some of the issues in PIP. It uses the concept of a resource ceiling, which is the priority of the highest-priority task that might acquire that resource. When a task acquires a resource, its priority is immediately boosted to ceil(resource) + 1.

**Key Concepts:**

* **Highest Locker Protocol (HLP):** A protocol that boosts a task's priority upon resource acquisition to ceil(resource) + 1.
* **Resource Ceiling (ceil(resource)):** The priority of the highest-priority task that could potentially acquire a given resource.

**Properties of HLP:** HLP prevents unbounded priority inversion, deadlocks, and chain blocking. However, it can lead to "inheritance blocking," where high-priority tasks unrelated to the resource are blocked because a low-priority task's priority is boosted significantly.

**Lemma 1:** If there is no chain blocking, there can be no deadlocks.

**Point 5.5.3:** Inheritance blocking is the major issue in the HLP protocol. It does not suffer from chain blocking, deadlocks, and unbounded priority inversion.

**Quick Revision:**

* HLP boosts resource holder's priority to ceiling + 1.
* Prevents priority inversion, deadlocks, and chain blocking.
* Can cause inheritance blocking (unrelated high-priority tasks blocked).

**5.5.7 Priority Ceiling Protocol (PCP)**

**Explanation:** The Priority Ceiling Protocol (PCP) is a resource allocation protocol that improves upon HLP and PIP, aiming to prevent deadlocks, chain blocking, and unbounded priority inversion while minimizing inheritance blocking. It uses resource ceilings and a "system ceiling" (CSC), the maximum of the ceilings of currently acquired resources.

**Key Concepts:**

* **Priority Ceiling Protocol (PCP):** A resource allocation protocol using resource ceilings and a system ceiling.
* **System Ceiling (CSC):** The maximum of the ceilings of all resources currently held by some task.
* **Resource Grant Clause:** Rules for granting a resource request based on the task's priority and the current CSC.
* **Inheritance Clause:** Similar to PIP, a task inherits the priority of a blocked higher-priority task.

**Properties of PCP:** PCP prevents deadlocks, chain blocking, and unbounded priority inversion. Inheritance blocking is also significantly controlled compared to HLP.

**Lemma 4:** After a task acquires a resource, no other task with a priority less than or equal to the CSC at that point can acquire any resource until the CSC is lowered.

**Theorem 5.5.1 Chain Blocking in the PCP Protocol:** There is no chain blocking in the PCP protocol.

**Point 5.5.4:** The PCP protocol does not suffer from deadlocks, chain blocking and unbounded priority inversion. The problem of inheritance blocking is also significantly controlled.

**Quick Revision:**

* PCP uses resource and system ceilings.
* Prevents deadlocks, chain blocking, and unbounded priority inversion.
* Controls inheritance blocking.