1. Threads and Processes

Def. 1. Process: A program executing inside an OS (operating-system). Processes share cpu but have their own memory space.

Def. 2. The **context** of a process is the set of utilities it uses and or operates on. That is:

- **Hardware context:** Instruction counter, Registers, Execution State, Flags ...
- **Memory context:** Virtual Memory, Heap-Space, Stack ...
- **OS-level context:** Process ID, open files ...

The **Process Lifecycle** is managed by the OS and can be in one of the following states:

- **NEW / CREATED:** The process is being created and loaded from disk into memory.
- **READY:** The process is in main memory waiting to be assigned to the CPU by the scheduler.
- **RUNNING:** The process is actively executing instructions on the CPU.
- **BLOCKED / WAITING:** The process cannot continue executing and is waiting for an event (like I/O access) to complete.
- **TERMINATED:** The process has finished execution and its resources are reclaimed.

Def. 3. Context Switch: The process of saving the context of a process and loading the context of another process.

Def. 4. Process Control Block (PCB): A table-entry storing relevant process information such as Program counter, Registers, Adress Space, Open files ...

1.1 Parallelism

Def. 5. Parallelism is when multiple processes execute simultaneously on diffrent CPU-Cores.

Def. 6. Concurrency is when multiple processes execute interleaved on a single CPU-Core.

Rem. 7.

Parallelism $\implies$ Concurrency Concurrency $\not\implies$ Parallelism

1.2 Multithreading

Def. 8. A thread is an independent sequence of exection running in the same OS-process.

Rem. 9. While processes can only communicate through main-memory and have diffrent memory spaces, threads share the same adress space. This leads to faster and more efficient context-switches as adressspace and PCB loading is not necessary.

There are diffrent levels of threads:

- **User-Level Threads:** Threads managed by the application using a thread library, ex. Java-Threads managed by JVM
- **Kernel-Level Threads:** Threads managed by to OS, which can schedule them on diffrent cores.
- **CPU-Level Threads:** Threads managed by the CPU, which can schedule them on the physical cores.

There are two ways to create a thread in java.

Code Example 10

```
class MyThread extends Thread { ...
    public void run() {
        ...
    }
}
MyThread thread = new MyThread();
thread.start(); // calls MyThread.run()
```

or using a runnable, which is recomended to do:

Code Example 11

```
class MyRunnable implements Runnable { ...
    public void run() {
        ...
    }
}
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable);
thread.start(); // calls MyRunnable.run()
```

Rem. 12. The .run() method is never called directly, because then the method is executed by the current thread (e.g. the main-thread). To spawn a new thread and run it we have to use thread.start().

In its lifecylce a thread can have the follwoing states:

- **NEW:** The thread has been created but not yet started.
- **RUNNABLE:** The thread is ready to run and waiting for the OS to schedule it.
- **RUNNING:** The thread is currently executing.
- **BLOCKED:** The thread is waiting for access to I/O devices
- **WAITING:** The thread is waiting to be notified
- **TIMED WAITING:** The thread is waiting for a specified amount of time before it continues.
- **TERMINATED:** The thread has finished its execution

Rem. 13. Every Thread has an **ID, Name, Priority and State** which might be usfull for certain tasks.

To wait for a thread to finish, we can use the .join() method on the thread we want to wait on. This lets the main-thread sleep until the joined thread terminates and wakes it up again.

Code Example 14

```
Thread t = new MyThread();
t.start(); //start t
t.join(); //sleep until t is finished
continueWork(); // this is only called after t
        finishes
```

Rem. 15. .join() creates overhead, for small thread-exection-time this is ineffiecient.

Imp. 16. If a worker-thread throws an exception the main-thread **will be woken up, but it will not know of the error**. We have to set an **UncaughtExeptionHandler using a try-catch block** when joining threads. For threads that run indefinetly we use a **timer task** that interupts threads.

Code Example 17

```
Thread t = new MyThread();
Thread timer = new MyTimerThread();
t.start();
timer.start(t, 1000);
try {
    t.join();
} catch (UncaughtExeptionHandler) {
    System.out.println("Error in thread t");
}
continueWork();

MyTimerThread extends Thread {
    run(Thread thread, int time) {
        timer.sleep(time);
        thread.interupt();
    }
}
```

Imp. 18. Thread scheduling is not deterministic. Don't rely on the execution order (interleaving).

Imp. 19. The program execution is finished if all non-deamon threads have terminated, not necessarily when the main method or main thread terminates.

2. Synchronization

Def. 20. Bad Interleaving: Interleaving of operations from multiple threads that leads to incorrect program state. **Critical Section:** Codeblocks that might lead to bad interleavings.

Def. 21. Thread Safety: Nothing bad happening in all possible interleavings.

Def. 22. Thread Liveness: Something good happening eventually.

2.1 Race Conditions

Def. 23. Race condition: correctness depends on specific interleaving.

Def. 24. Low-level race condition: insufficient synchronization of memory access (Data Races). **High-level race condition:** wrong decomposition/algorithm (bad interleavings despite mutual exclusion).

Generally we want one of three things when it comes to shared ressources:

- **Thread local (Isolated Mutability):** We dont use the ressource of other threads.
- **Read only (Immutable):** We only read and do not change the ressource
- **Mutual exclusion (Synchronization):** We use the ressource of other threads but we ensure that only one thread accesses it at a time.

2.2 Locks and Synchronized

Every Java-Object has an **intrinsic lock** or lock monitor. These locks are **unique** and can be optaind by threads which allows us to lock a certain codeblock.

Code Example 25

```
synchronized(lockObject) {
    i++;
}
```

Rem. 26. Synchronized can also be used as a keyword on a method to sync. the whole method.

Rem. 27. By default synchronized uses "this" as the lock-object and unlocks when an exception occurs.

Rem. 28. Java-Locks are reentrant meaning they can be locked multiple times by the same Thread recursively (It will then have to release the look as many times as it aquired it).

Atomic integers have builtin synchronization, for example for incrementing/decrementing.

Code Example 29

```
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet();
```

2.3 Wait, notify, notifyAll

We can use wait() on a lock-object to tell the current thread to sleep until further notice if the lock is not available. This allows use to save ressources.

Code Example 30

```
synchronized(lockObject) {
    while(!condition) {
        lockObject.wait();
    }
    // do work
}
```

Imp. 31. Always wait inside a while loop, as this ensures checking the condition after beeing notified.

To wake one arbitrary or all waiting threads we can use notify() or notifyAll() on the lock-object.

Code Example 32

```
synchronized(lockObject) {
    while (!condition) {
        lockObject.wait();
    }
    //do work
    notifyAll();
}
```

2.4 Lock API

Instead of using built-in synchronized blocks we can also use locks from the java.util.concurrent.locks package.

Code Example 33

```
private Lock lock = new ReentrantLock();
...
void doWork(int x) {
    lock.lock();
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

Imp. 34. Always use a **try-finally block to ensure the lock is released** even if an error occures.

Rem. 35. Use lock.tryLock(100, TimeUnit.MILLISECONDS) to accuire the lock only for a given amount of time, avoiding deadlocks.

Aspect	synchronized	Lock API
Handling	Auto (JVM)	Manual (lock/unlock)
Release	Automatic	Manual (finally!)
Scope	Block/Method	Flexible/Multi-method
Waiting	Infinite block	tryLock() avoids block
Interrupt	No	lockInterruptibly() ok
Flex.	Low	High
Comp.	Simple	Complex/Error-prone
Risk	Lower	Higher (forget unlock)
Usage	Simple sync	Advanced control

2.5 Guidelines for Concurrent programming

Imp. 36. Thread-Locality: Only share ressources if needed.

Imp. 37. Immutability: Use immutable data structures whenever possible.

Imp. 38. Consitent locking: Use locks to ensure mutual exclusion.

Def. 39. Lock granularity refers to the amount of code that is proteted by a lock in ways:

- Size of synchronized blocks
- Number of objects a lock is used for

Thm. 40. Coarse-grained locking is easier to implement but leads to less concurrency. **Fine-grained locking** is harder to implement without creating bugs but leads to more concurrency.

Imp. 41. Start with coarse-grained (simpler) and move to fine- grained (performance) only if contention on the coarser locks becomes an issue.

Def. 42. Atomicity: An operation is atomic if it is performed as a single, indivisible unit.

Imp. 43. Do not do expensive computations or I/O in critical sections, but also don't introduce race conditions

3. Optimization

Thm. 44. Moore's Law: Number of transistors on a microchip doubles every two years. Hitting the **Power-Consumption** (cooling limits) and **Memory** walls necessitated parallelism.

3.1 Caching

Faster CPU's does not mean faster execution. Writing and loading data forms a bottleneck.

Def. 45. Cache is a fast memory that preloads and stores frequently used data form the memory to reduce latency.

The usage of cache reduces stalls (times in which the CPU can not execute as the data is not yet present).

Rem. 46. Programmers improve cache usage by organizing data to increase **locality**, enabling efficient prefetching.

3.2 Vectorization

Def. 47. SIMD (Single Instruction Multiple Data) same instruction is applied to multiple data points simultaneously by using multiple ALUs inside the CPU.

3.3 Superscalar and Speculative Execution (ILP)

Def. 48. Superscalar execution refers to out of order execution of independent instructions in parallel using multiple ALUs.

Def. 49. Speculative execution refers to the execution of instructions before the actual need for them is known.

3.4 Pipelining

Pipelining increases throughput by overlapping the execution of multiple instructions in diffrent stages.

Def. 50. Throughput is the amout of work that can be done by a system in a given period of time.

$$tp = \frac{1}{max(compTime(stages))}$$

Def. 51. Latency is the time needed to perform a given computation.

$$L = numStages * max(compTime(stages))$$

Def. 52. A pipeline is **balanced** if all stages take the same amount of time.

Imp. 53. Balancing a pipeline can increase throughput.

3.5 Parallel Performance

Def. 54. Scalability describes how performance improves with additional resources (cores, memory, etc.).

Def. 55. Speedup is the ratio of the execution time of the sequential algorithm to the execution time of the parallel algorithm.

$$S_p = \frac{T_1}{T_p}$$

Def. 56. Efficiency is the ratio of speedup to the number of processors.

$$E_p = \frac{S_p}{p}$$

Rem. 57. Speedup depends on the definition of T_1 (execution time of the optimized sequential algorithm vs. the unoptimized one).

Thm. 58. Amdahl's Law sets a limit on speedup, as the sequential part of the algorithm can not be parallelized.

$$S(p) = \frac{1}{f + \frac{1-f}{p}} \implies {}_{p\to\infty}S(p) \leq \frac{1}{f}$$

Thm. 59. Gustafson's Law proposes an optimistic view compared to Amdahl. It shows that we might not be able to shrink execution time but we can perform more operations in that same execution time.

$$S_p = f + P(1-f)$$

4. Parallel Patterns

4.1 Divide and Conquer

Def. 60. Divide and Conquer refers to the strategy of splitting a problem into smaller subproblems, solving them recursively, and combining the results.

Problems of DC with manual fork/join:

- Poor reusability of code
- Commit to a number of threads a priori and can not dynamically adjust.

- Subproblems are not automatically assigned to a thread.

Rem. 61. Some manual fork/join fixes are: sequential-cutoffs or running one subproblem on the thread itself, instad of creating two new threads.

4.1.1 Executor Service

The Executor Service framework provides the way to schedule and run tasks on a thread-pool automatically.

Def. 62. Runnable: Task without return value. (Passed to `execute()` or `submit()`). **Callable**: Task returning `Future<T>`. (Passed to `submit()`).

Rem. 63. Pool Types: `newFixedThreadPool(n)` (keeps n threads), `newCachedThreadPool()` (scales dynamically).

Pros and Cons of Executor service:

- Not suited for: Recursive subproblems that depend on each other and have a deep fork-tree. (run out of threads)
- Suited for: flat structures that can run independently in parallel.

4.1.2 Fork-Join

Manages a pool of threads to solve recursive tasks using a work-stealing strategy, where threads have local task-queues which they work on and if empty steal from other threads.

Def. 64. RecursiveTask: Returns a result (like Callable). **RecursiveAction**: Does not return a result (like Runnable). **Key Methods**: `fork()` (async push), `join()` (wait, but steals work while waiting), `compute()` (main logic).

Code Example 65

```
class MyTask extends RecursiveTask<Integer> {
    int start, end;
    MyTask(int start, int end) { this.start =
          start; this.end = end; }

    protected Integer compute() {
        if (end - start <= SEQUENTIAL_CUTOFF) {
            return baseCase();
        } else {
            int mid = (start + end) / 2;
            MyTask left = new MyTask(start, mid);
            MyTask right = new MyTask(mid, end);

            left.fork(); // Async execute left
                  half
            int rightRes = right.compute(); //
                  Optimize: compute right half
                  now
            int leftRes = left.join();      //
                  Wait for left half

            return leftRes + rightRes;
        }
    }
}
```

Imp. 66. Make sure that every task has enough work to do (aprox. 100-10'000 ops.) (else the overhead will be larger than the parallel speedup gain)

4.1.3 Task graphs

The execution of parallel programs using Fork-Join can be represented as a Task-Graph (DAG).

Def. 67. T_1 is the execution time on a sequential machine (sum of all task durations in the DAG). **Span** T_∞: execution time with unlimited hardware (sum of longest dependent path in DAG). Max speedup is limited by T_1/T_∞ (Amdahl's Law).

Rem. 68. It is the programmers responsibility to find the optimal decomposition for minimal span (balanced tree). **Fork-join guarantees optimal scheduling with minimal idle time.**

4.2 Parallel Patterns

Def. 69. A **reduction** produces a single answer from a collection via an associative operator.

Ex. 70. Max, Min, Sum, Product of an array.

Def. 71. A **map** applies a function to each elements of a collection producing an output of the same size.

Ex. 72. Vector addition: $v_i = u_i + w_i \forall i \in [1, n]$

Rem. 73. Maps and Reductions have an optimal span of $O(\log n)$.

Def. 74. A **stencil** extends the map pattern by using the element itself and its neighbours.

Ex. 75. A smoothing kernel (ex. 3x3) that calculates the average value is sled over an image to smooth/blur the image.

Rem. 76. Arrays and balanced trees are usually parallelizable while linked lists are not even if the per element operation is parallelizable.

4.3 Parallel Algorithms

4.3.1 Prefix-Sum

Prefix-Sum: Sum of first n inputs.

Algorithm 77

```
upTask(); //divide problem and build tree with
      sums of subproblems
downTask(); //fill in the correction values (
      left_sum + parent_correction)
```

Thm. 78. The span of the prefix sum optimization is $O(\log n)$, therefore parallelism of $\frac{n}{\log n}$.

4.3.2 Pack

Pack pack all elements from an array that satisfy a certain condition into a seperate output. (by using prefix sum)

Algorithm 79

```
applyMap(); //applies a map to the input
//results in a binary vector [1, 0, 1, 1, 0,
      ...]\\
parallelPrefixSum(); //calculates the prefix sum
      of the binary vector\\
//results in [1, 1, 2, 3, 3, ...]\\
applyPackMap(); //applies another map to the
      result of the prefix sum\\
//results in [0, 2, 3, ...] (indices of elements
      to write to output)
```

4.3.3 Quicksort

We can reduce the **limiting factor of the problem division** from a span of $O(n)$ to a span of $O(\log n)$ using the pack-pattern.

Algorithm 80

```
applyMaps(); //apply maps for greater and smaller
      than pivot
parallelPrefixSum(); //count how many smallerthan
      /greatherthan elements come bofore
applyMap(); //index = numLessThan ||
      numTotalLessThan + numGreaterThan
```

We can do this by using two pack-patterns to partition the data. This leads to a partitioning of $O(\log n)$ levels times $O(\log n)$ for prefixSum hence $O(\log^2 n)$ total span.

This means a parallelism of $\frac{n \log n}{\log^2 n} = \frac{n}{\log n}$.

5. Memory Models

Def. 81. Memory Reordering: Compilers/hardware may reorder memory operations as long as *sequential* semantics hold (optimizes registries, caching, dead code). May break concurrency (e.g., hoisted checks $\implies$ infinite loops). L1 caches are private, writes may not immediately publish.

Rem. 82. `volatile`**:** Prevents local caching/reordering; ensures immediate visibility across threads. Memory barriers strictly enforce ordering.

Imp. 83. Volatile arrays only guarantee that the array reference is volatile, not the elements.

Rem. 84. Almost all common architectures allow the reordering of **Stores after Loads** to maximize performance, unless explicitly prevented by memory barriers.

5.1 Java Memory Model (JMM)

Contract providing minimal memory operation guarantees.

Def. 85. Program Order (PO): Total intra-thread execution trace order (no global sync).

Def. 86. Synchronization Actions (SA): E.g., Read/Write `volatile`, lock acquire/release.

Rem. 87. SA establish HB dependencies between threads:

- Write to volatile in thread $A \rightarrow$ Read from volatile in thread B
- Lock release in thread $A \rightarrow$ Lock acquire in thread B

Def. 88. Synchronization Order (SO): Total global order of all SA. All threads see these actions in the same global order.

Def. 89. Happens-Before (HB): Transitive closure of PO & Synchronizes-With (SW). $A \xrightarrow{HB} B \implies B$ sees results of A.

Rem. 90. HB-Consistency & Data Races: JMM permits data races (unsynchronized concurrent access where ≥ 1 is a write). Unsynchronized reads may observe stale values or non-HB ordered writes.

6. Locks

6.1 Hardware Support for Parallelism

Rem. 91. Software algorithms such as Bakery or Filter locks have **linear space complexity** $O(n)$, making them impractical for large numbers of processes.

Rem. 92. Modern architectures offer **atomic instructions** that allow for operations to happen in a single atomic step. This allows mutex-implementation with **constant space complexity** $O(1)$ and can be used for lock-free programming.

6.1.1 Typical Hardware Instructions

Def. 93.
- **Compare and Swap (CAS):** Atomically compares the value of a memory location with a given value. If equal, updates the memory location with a new value. Otherwise, does nothing.
- **Test and Set (TAS):** Atomically reads the value of a memory location and sets it to 1.
- **Load Linked / Store Conditional (LL/SC):** LL reads the value and sets a "link". SC writes only if the link is still valid (no other writes since LL).

Rem. 94. Java offers these via the classes **AtomicInteger**, **AtomicLong**, etc.

6.1.2 Locks using atomic operations

TAS Lock: `while(state.getAndSet(true));` Causes high cache coherence traffic (continuous atomic updates).

TATAS Lock: Spin on *read* before trying *write*.

Code Example 95

```
1  public void lock() {
2      do { while (state.get()); }
3      while (state.getAndSet(true));
4  }
```

Rem. 96. Backoff Strategy: E.g., exponential wait after failed acquire, significantly reduces cache traffic and contention.

6.2 Locking Algorithms

Properties for correct Lock (Mutex) execution:

- **Mutual Exclusion:** ≤ 1 process in critical section (CS).
- **Deadlock Free:** 1+ processes trying $\implies$ 1 eventually succeeds.
- **Starvation Free:** Every trying process eventually succeeds.

6.2.1 State Space Diagrams

Def. 97. State Space Diagram: Tracks every concurrent configuration. States are tuples $[p, q, want_p, want_q]$. Used to prove lock correctness:

- **No Mutex:** Reachable state where both processes are in the CS.
- **Deadlock:** A state with no outgoing transitions where threads are stuck.
- **Starvation:** An execution path where one thread never reaches the CS.

6.2.2 Attempts to mutual exclusion

Rem. 98. Attempt 1 (check then set flag): Both threads may see 0 and both enter CS $\implies$ No mutex. **Attempt 2** (set flag then check): Both threads may set flag to 1 and spin forever $\implies$ Deadlock. **Attempt 3** (turn variable): Does not guarantee reset if thread crashes, enforces strict alternation $\implies$ Not correct.

6.2.3 Successful Algorithms

Def. 99. Atomic register: Operations occur at a single instant. Event $A \prec B$ (precedes) if execution intervals don't overlap.

Peterson's Algorithm (2 Processes): Combines flags and a *victim* tie-breaker. Guarantees Mutex, Deadlock Freedom, and Starvation Freedom.

Code Example 100

```
1  wantp = true;
2  victim = 1; // "Give way" to other thread
3  while (wantq && victim == 1); // spin
4  /* CS */ wantp = false;
```

Thm. 101. Peterson's Algorithm provides Mutual Exclusion, Deadlock Freedom, and Starvation Freedom using only atomic registers.

Rem. 102. N-Processes: Filter Lock: $N-1$ levels, uses Peterson's at each level to block 1 victim. **Bakery Algorithm**: Threads take ticket $>$ all outstanding. Wait for lowest number.

Thm. 103. If S is a read/write system with at least two processes and S solves mutual exclusion with deadlock freedom, then S must have at least as many variables as processes.

6.3 Locking Strategies

6.3.1 Fine-Grained Locking

Rem. 104. Instead of locking everything coarse-grained (the whole data structure) we can lock individual parts of it. This is called **fine-grained locking** and allows for higher concurrency.

6.3.2 Read-Write Locks

Def. 105. A **read-write lock** allows fair access to resources by allowing one writer (every N reads) or multiple readers at the same time.

Rem. 106. To implement a RWL we need to keep track of the number of readers waiting, the number of writers waiting and the number of reads a writer has to wait for (ensures fairness).

6.3.3 Optimistic Locking

Def. 107. Optimistic Locking: Traverse a data structure without locking and only lock the nodes of interest. If upon locking the nodes have been modified and locks on predecessor/successor are no longer valid, restart the operation.

Rem. 108. Less overhead on locking but might require multiple traversals if there is high contention.

6.3.4 Lazy Locking

Def. 109. Lazy locking: Data-elements are not immediately updated but rather marked as deleted (or inserted) and only physically updated later. This saves the overhead of multiple traversals.

6.3.5 Lock-Free Data Structures

Rem. 110. Using atomic operations (mostly CAS) we can implement data structures without locks.

Code Example 111

```
1  class ConcurrentCounter {
2      private AtomicLong value = new AtomicLong(0);
3
4      public void increment() {
5          long expected;
6          do {
7              expected = value.get();
8          } while (!value.compareAndSet(expected,
                   expected + 1));
9      }
10 }
```

Def. 112. Lock-Free: System wide progress is guaranteed. At least one process can always make progress. A lock-free algorithm is always deadlock free.

Def. 113. Wait-Free: Any operation on the data structure will terminate in a bounded number of steps.

6.3.6 The ABA Problem

Imp. 114. The **ABA problem** is a common issue in lock-free programming where an atomic operation (like CAS) succeeds because the value has changed twice (A $\rightarrow$ B $\rightarrow$ A), making it appear unchanged. This can lead to incorrect program states.

Ex. 115. Pop element a from a stack, store in pool. Push b, then push a again (same address from pool). A thread that read the top before the pop and now checks with CAS will not notice the stack changed.

Def. 116. Tagged Pointers: Increment a **tag** along with the pointer in CAS to detect up to 2^k changes where k is the number of tag bits.

Def. 117. Hazard Pointers: Each thread stores addresses of nodes it is currently accessing. Other threads check hazard pointers before modifying/reclaiming nodes.

7. Beyond Locks

7.1 Deadlocks

Rem. 118. Deadlock detection in systems is implemented by finding cycles in the dependency graph.

Rem. 119. Deadlock avoidance techniques to ensure a cycle can never arise:

- **Imposing an ordering** on the resources and requiring processes to acquire resources in that order
- **Two-phase locking protocol** with retry and release on failure

Code Example 120

```
1  //Programming trick to create a global ordering
2  class Resource {
3      private static final AtomicLong counter =
               new AtomicLong();
4      private final long index =
               counter.incrementAndGet();
5      ...
6  }
```

7.2 Semaphores

Def. 121. A **semaphore** is a synchronization primitive that acts as a thread-safe counter to control access to a shared resource by multiple processes.

Code Example 122

```
public class Semaphore {
    private int n, S;
    acquire() {
        wait until S > 0;
        S--;
    }
    release() {
        S++;
    }
}
```

Code Example 123

```
Semaphore semaphore = new Semaphore(n);
semaphore.acquire();
    //no more than n threads can be here at once
semaphore.release();
```

Rem. 124. Semaphores are a generalization of locks and can be used to implement locks and other synchronization primitives.

Def. 125. A **binary semaphore** is a semaphore that can only have values 0 and 1, this can be used to implement locks.

Def. 126. A **rendezvous** is a location in the code where threads have to wait until all threads have reached that location before they can continue.

Rem. 127. For two threads, a rendezvous can be implemented with two binary semaphores.

Code Example 128

```
void thread1() {
    A.acquire();
    B.release();
    B.acquire();
    A.release();
}
void thread2() {
    B.acquire();
    A.release();
    A.acquire();
    B.release();
}
```

Imp. 129. Using Semaphores is often problematic as the programmer has to have deep knowledge of the algorithm to correctly implement the synchronization.

7.3 Barriers

Def. 130. A **barrier** is a synchronization primitive that coordinates threads to wait until all have reached the barrier, then releases them all at once.

Def. 131. A **two-phase barrier** is implemented with two semaphores and a counter.

Code Example 132

```
mutex=1, barrier1=0, barrier2=1, count=0;

aquire(mutex);
    count++;
    if (count == n) {
        accuire(barrier2);
        release(barrier1);
    }
release(mutex);

acuire(barrier1); release(barrier1);

aquire(mutex);
    count--;
    if (count == 0) {
        accuire(barrier1);
        release(barrier2);
    }
release(mutex);
```

7.4 Producer Consumer Pattern

Def. 133. The **producer consumer pattern** is a synchronization pattern where producers generate data and consumers process it, coordinated through a shared buffer.

Rem. 134. We can use producer-consumer patterns to build data flow parallel programs (pipelines): $T_0 \to T_1 \to T_2 \dots T_n$ where T_0 is a producer, T_n is a consumer, and intermediate nodes are both.

Rem. 135. A **Producer-Consumer Queue** with buffer improves throughput. We stall T_i iff the queue Q_i is full. Implemented using Semaphores that signal producers (space available) and consumers (data available).

Code Example 136

```
void enqueue(long x) {
    try {
        nonFull.acquire();
        manipulation.acquire();
        buffer[in] = x;
        in = (in + 1) % n;
    } catch (Exception e) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

Code Example 137

```
long dequeue() {
    long x = 0;
    try {
        nonEmpty.acquire();
        manipulation.acquire();
        x = buffer[out];
        out = (out + 1) % n;
    } catch (Exception e) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

7.5 Monitors

Def. 138. A **monitor** is a synchronization primitive that allows threads to wait on condition variables to be notified about changes in state.

Def. 139. A **condition variable** is a condition that threads can wait on. Other threads can signal it to notify waiting threads.

Code Example 140

```
Monitor buffer = new BufferMonitor(10);
Condition nonFull = buffer.newCondition();
Condition nonEmpty = buffer.newCondition();

void producer() {
    buffer.lock();
    try {
        while (buffer.isFull()) {
              nonFull.await(); }
        buffer.enqueue(i);
        nonEmpty.signal();
    } finally { buffer.unlock(); }
}

void consumer() {
    buffer.lock();
    try {
        while (buffer.isEmpty()) {
              nonEmpty.await(); }
        buffer.dequeue();
        nonFull.signal();
    } finally { buffer.unlock(); }
}
```

8. Concurrency Theory

8.1 Linearizability

Def. 141. A **history** is a partially ordered set of invocations and responses of operations in a distributed program.

Def. 142. A history H is **linearizable** if it can be extended to a history G by

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take effect

such that G is equivalent to a legal sequential history S with $\to_G \subseteq \to_S$.

Thm. 143. History H is linearizable $\iff$ for every object x, $H|x$ is linearizable.

Rem. 144. Linearizability is a **local property**: it can be proven in isolation for each object and then composed.

8.2 Sequential Consistency

Def. 145. A history H is **sequentially consistent** if it can be extended to a history G by

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take effect

such that G is equivalent to a legal sequential history S (no real-time order constraint, only program order of each thread).

Thm. 146. Sequential consistency is **not a local property**, thus we lose composability.

Rem. 147. Quiescence consistency requires non-overlapping methods to take effect in real-time order (periods of quiescence separate operations).

8.3 Consensus

Def. 148. A **consensus object** has a function *decide*(*value*) that returns the same value for all threads, where the value is one of the values v_i proposed by the threads.

Def. 149. The **Consensus Number** is the maximum number of nodes that can solve consensus validly, sequentially consistently and wait-free.

Thm. 150. Consensus Numbers:

- Atomic Registers: consensus number = 1
- Test-and-Set, Get-and-Set: consensus number = 2
- Compare-and-Swap (CAS): consensus number = ∞

Rem. 151. CAS has infinite consensus number because we can implement a valid, sequentially consistent, and wait-free consensus object using CAS:

Code Example 152

```
class CASConsensus {
    private final int FIRST = -1;
    private AtomicInteger r =
          AtomicInteger(FIRST);
    private AtomicIntegerArray proposed;

    public int decide(int value) {
        int i = ThreadID.get();
        proposed.set(i, value);
        if (r.compareAndSet(FIRST, i)) {
            return proposed.get(i);
        } else {
            return proposed.get(r.get());
        }
```

```
    }
}
```

Rem. 153. There is **no wait-free FIFO Queue using atomic registers**: a wait-free FIFO Queue can solve 2-consensus, contradicting the consensus number of 1 for atomic registers.

8.4 Transactional Memory

Rem. 154. Synchronization with locks is hard to implement, not composable and pessimistic. The idea behind TM is to remove the burden from the programmer and let the system enforce synchronization.

Def. 155. In transactional memory the programmer declares sections of code as **atomic** (a **transaction**), and the system ensures this code block takes effect atomically.

Lem. 156. Transactions run isolated, consistent and atomic: other threads can only observe the start or end state. **HTM** offers fast execution at limited size, **STM** offers larger transactions at higher cost.

Def. 157. For each transaction: the **read set** is the set of variables read, the **write set** is the set of variables written.

Rem. 158. Conflicts occur when the write set of transaction A overlaps with the read or write set of transaction B.

Rem. 159. TM can be implemented in hardware (**HTM**, using cache coherence infrastructure) or software (**STM**).

Def. 160. Weak isolation only isolates transactions from each other. **Strong isolation** isolates transactions from each other and from operations outside the transaction.

Def. 161. Flat Nesting: inner transactions are invisible to the outside; if an inner transaction aborts, the outer transaction aborts too.

8.5 Distributed Memory and Message Passing

Rem. 162. Shared mutable data is difficult to handle. Alternative: local mutable data where threads send data to each other via **message passing**.

Def. 163. Message Passing can be:

- **Synchronous**: Sender blocks until the message is received.
- **Asynchronous**: Fire and forget (messages are buffered).

Def. 164. MPI (Message Passing Interface): Standard for distributed memory programming. Processes are grouped into **communicators**, each with a unique **process rank**.

Def. 165. Message passing operations can be:

- **Blocking**: Operation blocks until data is safely sent/received.
- **Non-blocking**: Returns immediately, allowing overlap between computation and communication.

Def. 166. MPI collective operations:

- **Barrier**: Synchronize all processes
- **Broadcast**: Send data from one process to all
- **Reduce**: Perform operation on data from all processes, result to one process
- **Scatter**: Distribute data in chunks from one process to all
- **Gather**: Collect data chunks from all processes into one

8.5.1 Actor Model

Def. 167. Actor Model: A model of distributed computing where actors are the fundamental units of computation. Actors communicate by sending messages.

Rem. 168. Actors are isolated from each other and can only communicate by sending messages. This eliminates the need for locks or other synchronization primitives.