**Programming Shared Address Space Platforms**

**Author: Aliza Saeed**

**1. Introduction**

* **Explicit Parallel Programming**: Requires specifying parallel tasks and their interactions (synchronization or communication).
* **Shared Address Space Architectures**:
  + Communication is implicit because some/all memory is accessible to all processors.
  + Focus is on constructs for expressing **concurrency**, **synchronization**, and minimizing overheads.

**Variations in Paradigms**:

* Mechanisms differ based on:
  1. **Data Sharing**: How data is shared among threads.
  2. **Concurrency Models**: How tasks are parallelized.
  3. **Synchronization Support**: Tools to coordinate threads.

**Process-Based vs. Lightweight Threads**:

* **Process-Based Models**:
  + Default: Data is private unless explicitly shared.
  + Overhead: High due to protection domains (suitable for multiuser systems but inefficient for parallel programming).
* **Lightweight Threads**:
  + Default: All memory is global.
  + Advantage: Faster manipulation due to relaxed protection, making them preferred for parallel programming.

**2. Thread Basics**

**Definition**: A thread is a single stream of control in a program.

**Example**: Matrix Multiplication

* Code snippet computes the product of two matrices (n x n).
* Each iteration of the nested loop is independent, forming a **thread**.
* **Parallelization**: Transform the loop to create threads for each iteration:

c

Copy

Download

for (row = 0; row < n; row++)

for (column = 0; column < n; column++)

c[row][column] = create\_thread(dot\_product(get\_row(a, row), get\_col(b, col)));

* **Concurrency**: Threads can be scheduled on multiple processors.

**3. Logical Memory Model of a Thread**

* **Shared Address Space**: All memory is globally accessible to threads.
* **Thread-Local Data**:
  + Stacks (for function calls) are treated as local to avoid race conditions.
  + Liveness: Stacks cannot be assumed globally accessible due to dynamic scheduling.

**Physical Realities**:

* **Non-Uniform Memory Access (NUMA)**:
  + Local memory access is faster than remote (e.g., Origin 2000).
* **Caches**: Introduce variability in access times even in uniform architectures.
* **Locality**: Critical for performance optimization.

**4. Why Threads?**

**Advantages**:

1. **Software Portability**:
   * Threaded apps can run on serial or parallel machines without modification.
2. **Latency Hiding**:
   * While one thread waits (e.g., for I/O), others utilize CPU.
3. **Scheduling and Load Balancing**:
   * System dynamically maps tasks to processors, reducing idle time.
4. **Ease of Programming**:
   * Simpler than message-passing APIs (e.g., MPI).
   * Widespread support (e.g., POSIX threads).

**Disadvantage**:

* Performance tuning may require additional effort.

**5. The POSIX Thread (Pthread) API**

**Standardization**: IEEE 1003.1c-1995 (Pthreads) is widely adopted.

**Thread Creation and Termination**

* pthread\_create:

c

Copy

Download

int pthread\_create(pthread\_t \*thread\_handle, const pthread\_attr\_t \*attribute,

void \*(\*thread\_function)(void \*), void \*arg);

* + Creates a thread executing thread\_function.
  + thread\_handle: Unique identifier for the thread.
  + attribute: Thread properties (NULL for defaults).
  + arg: Arguments passed to the thread function.
* **Race Conditions**: Thread initialization must complete before creation to avoid scheduling issues.

**6. Synchronization Primitives in Pthreads**

**Mutual Exclusion (Mutex) Locks**

**Problem**: Race conditions when threads access shared data (e.g., updating best\_cost).  
**Solution**: Mutex locks ensure atomic access to critical sections.

**Functions**:

1. pthread\_mutex\_lock:
   * Locks a mutex; blocks if already locked.
2. pthread\_mutex\_unlock:
   * Releases the lock.
3. pthread\_mutex\_init:
   * Initializes a mutex to unlocked state.

**Example**:

c

Copy

Download

pthread\_mutex\_t lock;

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

pthread\_mutex\_lock(&lock);

// Critical section

pthread\_mutex\_unlock(&lock);

**Overheads**:

* Serialization: Large critical sections degrade performance.
* **Alleviation**: Use pthread\_mutex\_trylock to poll without blocking.

**7. Condition Variables**

**Purpose**: Synchronize threads based on predicates (e.g., producer-consumer).  
**Function**:

* pthread\_cond\_wait:

c

Copy

Download

int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);

* + Blocks until signaled; releases mutex while waiting.

**Workflow**:

1. Thread locks mutex and checks predicate.
2. If predicate false, waits on condition variable.
3. Another thread signals (pthread\_cond\_signal) to resume waiting threads.

**8. Controlling Thread and Synchronization Attributes**

**Attributes Objects**:

* Define properties of threads/mutexes (e.g., scheduling policy, stack size).  
  **Advantages**:
* Separates semantics from implementation.
* Improves modularity and portability.

**Example**:

c

Copy

Download

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_attr\_setschedpolicy(&attr, SCHED\_RR); // Round-robin scheduling

**9. Thread Cancellation**

**Purpose**: Terminate threads prematurely (e.g., chess move evaluation).  
**Function**:

* pthread\_cancel:

c

Copy

Download

int pthread\_cancel(pthread\_t thread);

* + Sends cancellation request (no guarantee of immediate termination).
  + Cleanup handlers reclaim resources before termination.

**Note**: Similar to pthread\_exit but initiated by another thread.

**10. Additional Topics**

* **Read-Write Locks**: Allow concurrent reads but exclusive writes.
* **Barriers**: Synchronize threads at specific points (e.g., phases of computation).

**Key Takeaways**

1. **Shared Memory Model**: Simplifies communication but requires careful synchronization.
2. **Threads vs. Processes**: Threads are lightweight and share memory by default.
3. **Pthreads API**: Standard for thread management (creation, synchronization, cancellation).
4. **Synchronization**: Mutexes and condition variables prevent race conditions.
5. **Performance**: Minimize critical sections and leverage attributes for optimization.

**1. What is the main difference between process-based and thread-based models in shared address space programming?**

**Answer**:

* **Process-Based Models**:
  + Data is private by default (unless explicitly shared).
  + Enforces memory protection, leading to higher overhead due to separate address spaces.
  + Suitable for multiuser systems but inefficient for parallel programming.
* **Thread-Based Models**:
  + All memory is globally accessible by default.
  + Lightweight due to shared address space and relaxed protection.
  + Preferred for parallel programming due to lower overhead.

**2. Explain the concept of a race condition with an example. How can it be resolved?**

**Answer**:

* **Race Condition**:
  + Occurs when multiple threads access shared data concurrently, leading to unpredictable results.
  + **Example**:

c

Copy

Download

if (my\_cost < best\_cost)

best\_cost = my\_cost; // Non-atomic update by multiple threads.

If two threads execute this simultaneously, best\_cost may end up with an inconsistent value (e.g., 75 instead of 50).

* **Resolution**:
  + Use **mutex locks** to enforce atomicity:

c

Copy

Download

pthread\_mutex\_lock(&lock);

if (my\_cost < best\_cost) best\_cost = my\_cost;

pthread\_mutex\_unlock(&lock);

**3. Why are condition variables used in thread synchronization? Provide a real-world analogy.**

**Answer**:

* **Purpose**:
  + Condition variables allow threads to wait for a predicate (condition) to become true before proceeding.
  + Avoids busy-waiting (polling), reducing CPU overhead.
* **Example**: **Producer-Consumer Problem**
  + **Producer** signals the **consumer** when data is available (task\_available = true).
  + **Consumer** waits using pthread\_cond\_wait until signaled.
* **Analogy**:
  + Like a **taxi stand** where drivers (producers) wait for passengers (consumers) and vice versa, avoiding constant checking.

**4. Compare mutex locks and read-write locks. When would you use each?**

**Answer**:

| **Mutex Locks** | **Read-Write Locks** |
| --- | --- |
| Only one thread can hold the lock. | Multiple readers or one writer. |
| No distinction between read/write. | Optimized for read-heavy workloads. |
| **Use Case**: Critical sections with writes. | **Use Case**: Databases (many reads, rare writes). |

**5. What is false sharing? How does it impact performance in shared address space systems?**

**Answer**:

* **False Sharing**:
  + Occurs when threads on different processors modify **different variables** that reside on the **same cache line**.
  + Causes unnecessary cache invalidations, leading to high latency.
* **Impact**:
  + Degrades performance due to frequent cache coherence traffic.
* **Solution**:
  + **Padding** variables to ensure they occupy separate cache lines.
  + **Example**:

c

Copy

Download

struct { int data; char padding[CACHE\_LINE\_SIZE - sizeof(int)]; };

**6. How does thread cancellation work in Pthreads? What precautions are needed?**

**Answer**:

* **Mechanism**:
  + pthread\_cancel(thread) sends a cancellation request.
  + The target thread may defer cancellation until it reaches a **cancellation point** (e.g., I/O operations).
* **Precautions**:
  + Use **cleanup handlers** (pthread\_cleanup\_push/pop) to release resources.
  + Avoid cancellation in critical sections (may leave mutexes locked).

**7. Why is the logical memory model of threads idealized, and how do physical systems deviate from it?**

**Answer**:

* **Logical Model**:
  + Assumes uniform memory access (UMA) for all threads.
* **Physical Reality**:
  + **NUMA**: Remote memory access is slower (e.g., Origin 2000).
  + **Caches**: Introduce non-uniform latency even in UMA.
  + **Locality**: Performance depends on data placement (e.g., cache lines, NUMA nodes).

**8. Explain the role of barriers in parallel programming. Give an example.**

**Answer**:

* **Purpose**:
  + Synchronize threads at a specific point (e.g., end of a computation phase).
* **Example**:

c

Copy

Download

#pragma omp barrier // OpenMP example

* + All threads wait here until every thread reaches the barrier.
* **Use Case**:
  + Parallel matrix multiplication (wait for all threads to finish a row/column).

**9. What are the advantages of using Pthreads over message-passing (e.g., MPI)?**

**Answer**:

1. **Implicit Communication**: Shared memory avoids explicit data transfers.
2. **Lower Overhead**: No serialization/deserialization of messages.
3. **Dynamic Load Balancing**: Threads can be scheduled adaptively.
4. **Ease of Debugging**: Single address space simplifies tools like gdb.

**10. How do attributes objects improve modularity in Pthreads programs?**

**Answer**:

* **Separation of Concerns**:
  + Thread properties (stack size, scheduling policy) are defined separately from logic.
* **Example**:

c

Copy

Download

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_attr\_setstacksize(&attr, 16384); // 16KB stack

pthread\_create(&thread, &attr, worker, NULL);

* **Benefits**:
  + Portable (attributes adapt to OS implementations).
  + Reusable (same attributes for multiple threads).