CS 3113 – Intro to Operating Systems – Spring 2025
Project Five and BONUS: Due April 29, 20205, 11:59 PM

Description: This project aims to compute the sum of an integer array using a structured hierarchy of threads arranged as a **full binary tree**. It will introduce students to thread creation, condition-based synchronization, structured computation, and top-down controlled termination in a shared-memory environment using POSIX threads in C++.

Students will gain practical experience with:

- Thread creation and identification
- Shared data management
- Use of pthread_mutex_t and pthread_cond_t
- Tree-based hierarchical computation
- Controlled cleanup and termination signaling

Inputs

1. **Tree Height (H)**
   An integer representing the height of a full binary tree.
   - The root resides at level 1 (height 1).
   - A full binary tree of height H contains 2^{H-1} **leaf threads**.
2. **Array Size (M)**
   An integer specifying the number of user-provided input integers.
3. **Array Elements**
   A list of M integers entered by the user.

Computation Setup

1. Chunk Division

- Compute N=2^{H-1}, the number of leaf threads.
- If M is **not divisible** by N, **pad** the array with zeros to make its length M' divisible by N.
- Divide the array into N **equal chunks** of size = $\frac{M'}{N}$

2. Thread Tree Construction

- The threads are organized in a **full binary tree** of height H.
- Total number of threads: 2N−1=$2^{H} - 1$,
- Each thread has a **unique index** in the array-based tree (root is at index 0).
- For a thread at index i:
  - Left child index: 2i + 1
  - Right child index: 2i + 2
- Each thread also has:
  - **Level**: $\lfloor \log_2(i+1) \rfloor + 1$
  - **Position from the left**: $i - (2^{level-1} - 1)$

Leaf Node Indexing

- Leaf threads begin at index $2^{H-1} - 1$ and go up to $2^{H-1} - 2$

- The j^{th}leaf thread has index:

  $leftIndex_j = 2^{H-1} - 1 + j$ for $0 \leq j < N$

- The chunk assigned to leaf j is from:

  $start = j \times chunkSize$ to $end = (j + 1) \times chunkSize - 1$

Thread Behavior

Each thread consists of **two coordinated phases**:

Phase 1: Computation

• Leaf Threads:

- Wait until they are signaled (via compute_cond) by the **main thread** to begin computation.
- Compute the sum of their assigned chunk in the array.
- Print:
  - Level
  - Position from the left
  - Thread index
  - Thread ID (pthread_self())
  - Computed local sum
- Signal their **parent thread** (using a shared status mechanism) that computation is complete.

• Internal Threads

- Wait until they are signaled (via compute_cond) by their **parent thread** to begin computation.
- Wait for **both child threads** to complete computation (via shared flags or other signaling).
- Retrieve the computed values from the results array.
- Compute the sum of the two child results.
- Print:
  - Their level, position, index, thread ID
  - The values received from left and right child threads
- Signal their **parent thread** that their own computation is complete.

Phase 2: Controlled Termination

- **Threads do not terminate immediately** after computation.
- Each thread has a **mutex** and **condition variable**.
- After computation:
  - The thread locks its mutex and waits on its condition variable.
  - It exits only after receiving a termination signal from its parent.
- **Root Thread**:
  - After printing the final result, it initiates the **tree cleanup** process.
  - It signals its two children to terminate.
- **Internal Threads**:
  - Once signaled by their parent, they:
    - Print their termination message
    - Signal their two children to terminate
    - Wait for both child threads to terminate using pthread_join
    - Exit themselves

- **Leaf Threads**:
    - Once signaled, print their termination message and exit immediately.

Synchronization and Termination Control

Each thread maintains:

- A **mutex** to guard its internal state
- A **condition variable** to block until explicitly released

This mechanism ensures:

- Precise control over the termination sequence
- Top-down order of cleanup
- Graceful and observable shutdown

Thread Identification and Indexing

- The entire thread tree is represented as an array of size 2^{H-1}.
- Index i uniquely identifies the thread.
- Each thread logs:
    - Index
    - Level
    - Position from the left
    - Thread ID
    - Computed result (if applicable)
    - Termination message (after being signaled)

Expected Output Summary

Each thread prints:

During Computation

- **Leaf Threads**:

```
[Thread Index X] [Level L, Position P] [TID ####] computed leaf sum: Y
```

- **Internal Threads**:

```
[Thread Index X] [Level L, Position P] [TID ####] received:
  Left child [Index A, Level AL, Pos AP, TID ###]: valueA
  Right child [Index B, Level BL, Pos BP, TID ###]: valueB
[Thread Index X] computed sum: Z
```

During Termination

Each thread prints:

```
[Thread Index X] [Level L, Position P] terminated.
```

Sample Output

```
[Thread Index 3] [Level 3, Position 0] [TID 140083955230464] computed
leaf sum: 6
[Thread Index 4] [Level 3, Position 1] [TID 140083946837760] computed
leaf sum: 15
[Thread Index 5] [Level 3, Position 2] [TID 140083938445056] computed
leaf sum: 24
[Thread Index 6] [Level 3, Position 3] [TID 140083930052352] computed
leaf sum: 10

[Thread Index 1] [Level 2, Position 0] [TID 140083921659648] received:
  Left child [Index 3, Level 3, Pos 0, TID 140083955230464]: 6
  Right child [Index 4, Level 3, Pos 1, TID 140083946837760]: 15
[Thread Index 1] computed sum: 21

[Thread Index 2] [Level 2, Position 1] [TID 140083913266944] received:
  Left child [Index 5, Level 3, Pos 2, TID 140083938445056]: 24
  Right child [Index 6, Level 3, Pos 3, TID 140083930052352]: 10
[Thread Index 2] computed sum: 34

[Thread Index 0] [Level 1, Position 0] [TID 140083904874240] received:
  Left child [Index 1, Level 2, Pos 0, TID 140083921659648]: 21
  Right child [Index 2, Level 2, Pos 1, TID 140083913266944]: 34
[Thread Index 0] computed final sum: 55

[Thread Index 0] now initiates tree cleanup:
  - Terminates Left child [Thread Index 1], which terminates its own
children [3, 4]
  - Terminates Right child [Thread Index 2], which terminates its own
children [5, 6]

Thread termination log:
[Thread Index 3] [Level 3, Position 0] terminated.
[Thread Index 4] [Level 3, Position 1] terminated.
[Thread Index 1] [Level 2, Position 0] terminated.
[Thread Index 5] [Level 3, Position 2] terminated.
[Thread Index 6] [Level 3, Position 3] terminated.
[Thread Index 2] [Level 2, Position 1] terminated.
[Thread Index 0] [Level 1, Position 0] terminated.
```

Main Program Structure

```
int main() {
    // Step 1: Read inputs
    // - Read H and M
    // - Read M integers into input array

    // Step 2: Compute tree parameters
    // - N = 2^(H-1)
    // - Pad input array with zeros to get M' divisible by N
    // - chunkSize = M' / N
    // - totalThreads = 2^H - 1

    // Step 3: Allocate arrays
    // - ThreadArg threadArgs[totalThreads]
    // - int results[totalThreads]
    // - int input[M']

    // Step 4: Initialize each thread's data
    // - For each index i in 0 to totalThreads - 1:
    //     - Compute level and position
    //     - Assign index, level, position
    //     - Assign array, results pointers
    //     - For leaf threads:
    //         - Set isLeaf = true
    //         - Assign start and end indices in input array
    //     - Initialize mutexes and condition variables

    // Step 5: Create all threads (loop)
    // - For each index i, call pthread_create(&threadArgs[i].thread, ...,
    //computeSum, &threadArgs[i])

    // Step 6: Trigger leaf threads to begin computation
    // - For each leaf thread (index from 2^(H-1)-1 to 2^H-2):
    //     - Lock compute_mutex
    //     - Set compute_ready = true
    //     - Signal compute_cond
    //     - Unlock compute_mutex

    // Step 7: Wait for root thread to finish computation
    // - Wait on a condition variable OR
    // - Use polling to detect when result[0] is ready

    // Step 8: Trigger root thread to begin termination
    // - Lock terminate_mutex for root
    // - Set terminate_ready = true
    // - Signal terminate_cond
    // - Unlock terminate_mutex

    // Step 9: Join all threads
    // - For all threads in threadArgs, call pthread_join

    // Step 10: Destroy mutexes and condition variables
    // - For each threadArg, destroy compute/terminate mutexes and conds

    return 0;
}
```

Rules and Submission Policy

All projects in this course are **individual assignments** and are not to be completed as group work. The use of **automatic plagiarism detection tools** will be employed to ensure academic integrity. Collaboration with others or the use of outside third parties to complete this project is strictly prohibited. Submissions must be made through **GradeScope**, where automated grading will be conducted. Additionally, manual grading may be performed to ensure correctness and adherence to requirements.

Several input files will be used to evaluate your program. While a subset of these input files will be provided to you for testing, additional files not shared beforehand will also be used during grading. Your score on this project will depend on producing correct results for all input files, including the undisclosed ones used in GradeScope evaluation.

All programs must be written in **C++** and must compile successfully using the **GNU C++ compiler**. It is your responsibility to ensure that your program adheres to these requirements.

The course syllabus provides more details on the late and submission policy. You should also go through that.

BONUS: 100 POINTS Submit this as a separate project.

Project: Parallel Array Summation Using a Process Tree with Pipes

In this variation of the parallel summation project, the goal is to compute the sum of a given array of integers using a **hierarchy of processes** arranged as a **full binary tree**. Instead of using threads and shared memory, this version utilizes **process creation via fork()** and **inter-process communication (IPC)** using **Unix pipes**. This approach emphasizes process management, independent address spaces, and explicit message passing.

The binary tree of processes is constructed such that each process corresponds to a node in a complete binary tree. The tree has a specified height H, with 2^{H-1}leaf processes at the bottom and a total of $2^{H} - 1$ processes in the system. The root process (tree index 0) is created when the program starts, and it recursively forks its child processes to build the complete tree.

Each **leaf process** is responsible for computing the sum of a portion of the array. The array is divided into equal-sized chunks (with padding added as needed), and each leaf process reads its assigned chunk from inherited memory, computes the sum, and sends the result back to its parent process through a **pipe**.

Each **internal process** waits for data from its two child processes. After reading both results from its pipes, it computes the sum of those two values and sends the result up to its own parent. This continues recursively until the **root process** receives two values from its immediate children and computes the final result, which represents the sum of the entire array.

After computing the final result, the root process prints the total sum and coordinates a **controlled termination** of all child processes. Each process prints a termination message after completing its task and before exiting.

Sample Output

[PID 40103] [Index 6] [Level 3, Position 3] computed sum: 10
[PID 40102] [Index 5] [Level 3, Position 2] computed sum: 24
[PID 40101] [Index 4] [Level 3, Position 1] computed sum: 15
[PID 40100] [Index 3] [Level 3, Position 0] computed sum: 6

[PID 40105] [Index 1] [Level 2, Position 0] received: 6, 15 → sum: 21
[PID 40106] [Index 2] [Level 2, Position 1] received: 24, 10 → sum: 34

[PID 40107] [Index 0] [Level 1, Position 0] received: 21, 34 → Final sum: 55

[PID 40103] [Index 6] terminated.
[PID 40102] [Index 5] terminated.
[PID 40101] [Index 4] terminated.
[PID 40100] [Index 3] terminated.
[PID 40105] [Index 1] terminated.
[PID 40106] [Index 2] terminated.
[PID 40107] [Index 0] terminated.