Synchronization

Thread Throttles and Pools

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Introduction

Multi-threading: Designed to improve performance by allowing multiple threads to run concurrently.

Thread Creation and Destruction: Creating and destroying threads can be time-consuming and resource-intensive.
Number of Software Threads: The number of software threads is often limited by the number of hardware threads (CPU cores) available.

- Multi-threading is designed to improve performance, but it has overheads
  - Create and destroy a thread can be time-consuming
  - The number of software threads is a functions of the number of hardware threads

Over-subscription: Occurs when the number of software threads ready to start is higher than the number of hardware threads available. This can lead to performance degradation due to excessive context switching.

- The **global load** has to be managed manually
- Problem 1: **Over-subscription** occurs every time the number of software threads ready to start is higher than the number of hardware threads available in the system
- Problem 2: Even without over-subscription, the running threads can exceed the system resources (e.g., memory-related ones) in **some code section**

Even without over-subscription, running too many threads can exceed system resources (e.g., memory) in certain code sections, leading to performance issues.

Introduction

Manual and Automatic Strategies: Various strategies can be employed to control overhead and prevent over-subscription.

- There are several manual and automatic strategies to keep overhead under control and avoid over-subscription
- In this section, we present
  - Semaphore Throttles
    - A strategy to manually reduce the number of workers in critical situations
  - Thread Pools
    - A design pattern for achieving concurrency but reducing overheads
    - Thread pools are also called the **replicated worker model** or the **worker-crew model**

Semaphore Throttles

Performance Degradation: Severe performance issues occur in specific "expensive" code sections where threads use too many resources, leading to high contention.

- Scenario
  - The user activate **N** worker threads
  - Unfortunately, performance degradation is severe in specific "expensive" code section
    - Ts use too much resources and contention is **high**
- "Semaphore throttles"

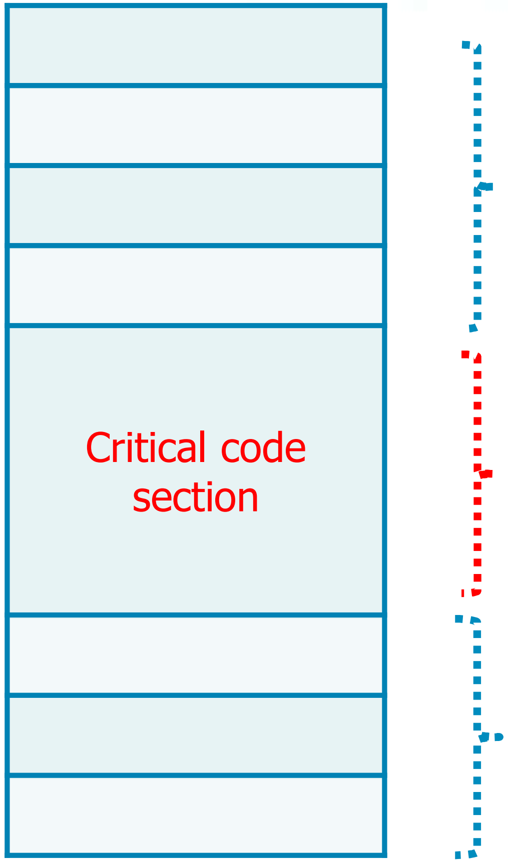
Use a Semaphore: To limit the maximum number of threads running in specific (critical) sections of the code.

  - Use a semaphore to **reduce/fix** the **maximum** amount of running Ts in specific (**critical**) sections of the code
    - The boss T creates a **new** semaphore and sets the maximum number of Ts in the CS to a "reasonable value", depending on the number of cores, processors, etc.

Semaphore Throttles

Pseudo-code

Task



N threads may run

Less than **N** threads may run

N threads may run

n<<N

Initializes the semaphore sem with a maximum count of n. This means up to n threads can enter the critical section simultaneously.

```
sem_init (sem, n)
```

Before Critical Section:

```
...
```

Each thread calls sem_wait(&sem) before entering the critical section. If the semaphore count is greater than zero, it decrements the count and allows the thread to proceed. If the count is zero, the thread is blocked until the count becomes positive.

```
sem_wait (sem);
```

The thread executes the critical section code.

```
... CS ...
```

The critical section is a part of the code where shared resources are accessed or modified. It is critical because improper access can lead to data corruption or inconsistent results. In this context, the critical section is where resource contention is high, and we want to limit the number of threads accessing it simultaneously to avoid performance degradation

```
sem_post (sem);
```

After exiting the critical section, the thread calls sem_post(&sem), which increments the semaphore count, potentially unblocking a waiting thread.

```
...
```

Semaphore Throttles
Limiting the number of threads working in the hyper-critical SC section

Task: Represents the overall task being executed by multiple threads.
Critical Code Section: A specific part of the code where resource contention is high.
Semaphore Throttles: Limits the number of threads that can enter the critical section simultaneously, reducing contention and improving performance.

Semaphore Throttles

- The number of worker is tunable
  - The boss T may dynamically decrease or increase the number of active workers by waiting or releasing semaphore units
    - Notice that the maximum number of Ts allowed is set once and only once at initialization

Set it to be "large enough"

- Workers using more resources
  - Should acquire multiple semaphore units
    - The idea is that with expensive threads we may be forced to reduce the level of parallelism
  - Caution
    - Heavy threads can wait **more** on the throttles
    - We can generate deadlocks

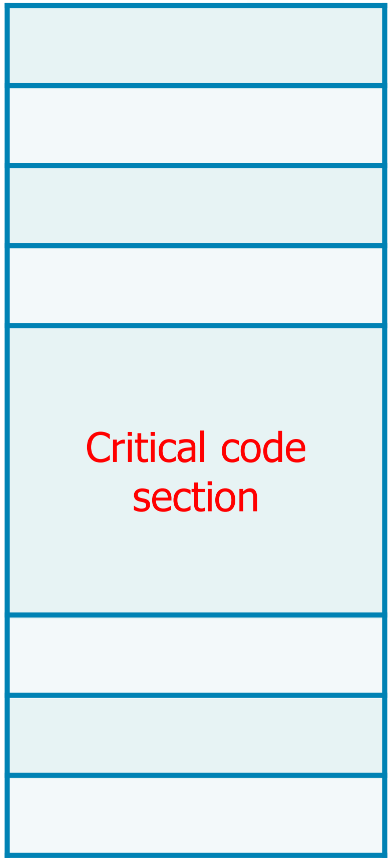
Multiple Semaphore Units:
Threads that are resource-intensive (expensive) should acquire multiple semaphore units.
This reduces the level of parallelism for these threads, ensuring that they do not overwhelm the system resources.
By acquiring more units, these threads effectively reduce the number of other threads that can enter the critical section simultaneously.

Heavy Threads and Wait Times:
Threads that acquire multiple semaphore units may experience longer wait times.
This is because they need to wait for multiple units to become available, which can lead to increased contention.
Deadlocks:
There is a risk of deadlocks if semaphore units are not managed properly.
Deadlocks occur when threads are waiting indefinitely for resources held by each other, creating a cycle of dependency that cannot be resolved.

Example (part A)

Pseudo-code

Task




```
sem_init (sem, n)
```

Standard workers follow a simple pattern of acquiring and releasing the semaphore:

```
...
sem_wait (sem);
...
sem_post (sem);
...
```

Standard workers

May cause threads to deadlock (see u02s02 ex 09)

Expensive workers may need to acquire multiple semaphore units to reduce their impact on system resources:

```
...
sem_wait (sem);
sem_wait (sem);
...
sem_post (sem);
sem_post (sem);
...
```

Expensive workers

Acquiring multiple semaphore units can lead to deadlocks if not managed carefully.
Deadlocks occur when threads are waiting indefinitely for resources held by each other.
For example, if Thread A holds one unit and waits for another, while Thread B holds the second unit and waits for the first, neither can proceed, resulting in a deadlock.
Avoiding Deadlocks:
To avoid deadlocks, ensure that threads acquire and release semaphore units in a consistent order.
Implementing timeout mechanisms or deadlock detection algorithms can also help mitigate this risk.

Example (part B)

Pseudo-code

Task

Critical code section

```
sem_init (sem, n)
mutex_init (m, 1)
```

We need to see «waits» as part of a CS and protect them with a mutex

Standard workers use both the semaphore and the mutex to manage access to the critical section:

```
...
mutex_lock (m);
sem_wait (sem);
mutex_unlock (m);
...
mutex_lock (m);
sem_post (sem);
mutex_unlock (m);
...
```

Standard workers

Expensive workers also use both the semaphore and the mutex, but they may need to acquire multiple semaphore units:

```
...
mutex_lock (m);
sem_wait (sem);
sem_wait (sem);
mutex_unlock (m);
...
mutex_lock (m);
sem_post (sem);
sem_post (sem);
mutex_unlock (m);
...
```

Expensive workers

Mutex Protection:
The mutex ensures that the semaphore operations (sem_wait and sem_post) are treated as part of a critical section.
This prevents race conditions where multiple threads might simultaneously attempt to modify the semaphore count.
Critical Section Management:
By locking the mutex before waiting on the semaphore and unlocking it after posting to the semaphore, threads ensure that the semaphore operations are atomic.
This helps in maintaining the integrity of the semaphore count and prevents inconsistencies.

Example (part C)

Pseudo-code

Task

Critical code section

```
sem_init (sem, n)
mutex_init (m, 1)
```

Even this scheme has limited applications and it can create a deadlock.

For example, with n=3 the first expensive worker passes. but the second blocks everybody else because it stops on the second wait and it does not release the mutex.

We need to see «waits» as part of a CS and protect them with a mutex

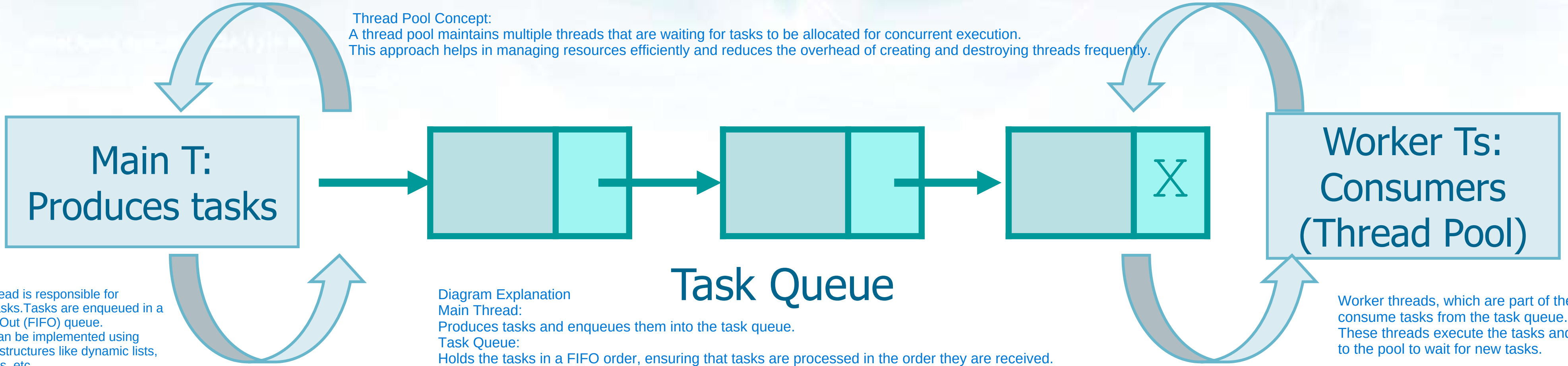
```
...
mutex_lock (m);
sem_wait (sem);
sem_wait (sem);
mutex_unlock (m);
...
mutex_lock (m);
sem_post (sem);
sem_post (sem);
mutex_unlock (m);
...
```

Expensive workers

Deadlock Scenario:
If an expensive worker acquires the mutex and then waits on the semaphore multiple times, it might block other threads from acquiring the mutex.
If the semaphore count is not sufficient to allow the second sem_wait to succeed, the thread will be stuck holding the mutex, preventing other threads from making progress.

Thread Pools

Thread Pool Concept:
A thread pool maintains multiple threads that are waiting for tasks to be allocated for concurrent execution.
This approach helps in managing resources efficiently and reduces the overhead of creating and destroying threads frequently.



The main thread is responsible for generating tasks.Tasks are enqueued in a First-In-First-Out (FIFO) queue.
The queue can be implemented using various data structures like dynamic lists, circular arrays, etc.

Diagram Explanation
Main Thread:
Produces tasks and enqueues them into the task queue.
Task Queue:
Holds the tasks in a FIFO order, ensuring that tasks are processed in the order they are received.
Worker Threads:
Fetch tasks from the task queue, execute them, and then return to the pool to wait for more tasks.

Worker threads, which are part of the thread pool, consume tasks from the task queue.
These threads execute the tasks and then return to the pool to wait for new tasks.

- A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution
  - One main thread generates the tasks
    - Tasks are enqueued in a (FIFO) queue
      - Dynamic list, circular array, etc.
  - The thread pool organizes the working threads manipulating the tasks

Thread Pools

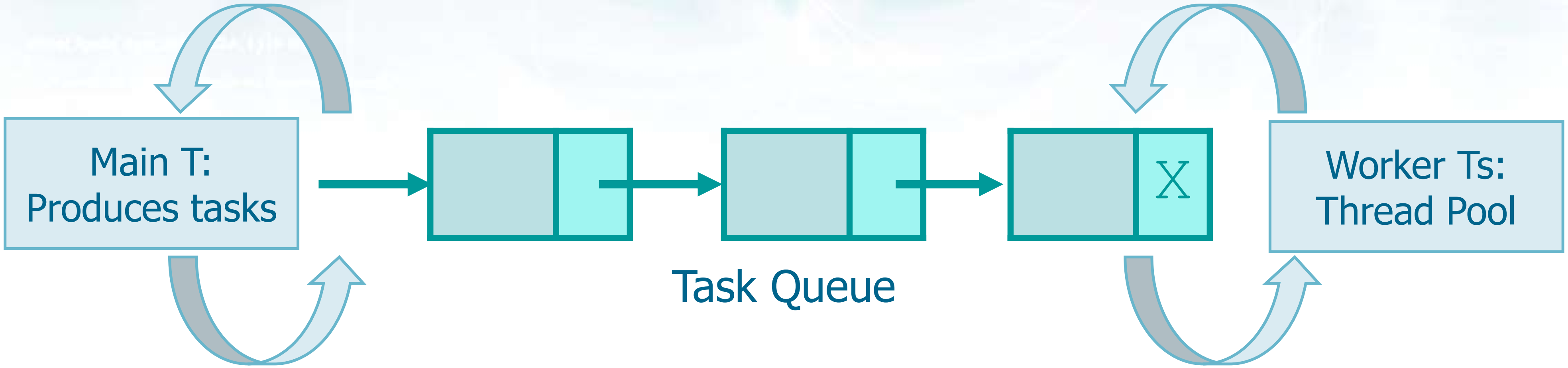


Diagram Explanation
Main Thread:
Initializes the task queue and the worker threads.
Creates tasks and inserts them into the task queue.
Worker Threads:
Continuously fetch tasks from the task queue, execute them, and then return to the pool to wait for more tasks.

- More specifically
  - The Main thread
    - Initializes the “task queue” and the “working theads”
    - Creates “work objects” (or “tasks”)
    - Inserts tasks into the queue
  - Worker threads in the pool
    - Get tasks from the queue

Main Thread Responsibilities:
Initialization:
Initializes the task queue and the worker threads.
Task Creation:
Creates work objects or tasks that need to be executed.
Task Insertion:
Inserts tasks into the task queue for the worker threads to process.
2. Worker Threads Responsibilities:
Task Retrieval:
Worker threads retrieve tasks from the task queue.
Task Execution:
Execute the tasks and then return to the pool to wait for new tasks.

Thread Pools

- The size of a thread pool is the number of threads ready to execute tasks
  - It is a tunable parameter
  - It is crucial to optimize performance
    - Instead of a new thread for each task, thread **creation (destruction)** is restricted to the **initial (final)** generation of the pool
    - This often results in better performance and better system stability
    - An excessive number of threads may waste memory and increase context-switching incurring in performance penalties

Thread Pool Size:
The size of a thread pool refers to the number of threads that are ready to execute tasks.
This size is a tunable parameter, meaning it can be adjusted based on the needs of the application and the available system resources.

Performance Optimization:
Optimizing the size of the thread pool is crucial for performance.
Instead of creating a new thread for each task, threads are created once during the initial generation of the pool and reused for multiple tasks.
This approach reduces the overhead associated with thread creation and destruction, leading to better performance and system stability.

Resource Management:
An excessive number of threads can waste memory and increase context-switching, which incurs performance penalties.
Context-switching occurs when the CPU switches from one thread to another, which can be costly in terms of performance if done too frequently.

Tunable Parameter:
The thread pool size can be adjusted to find the optimal balance between resource utilization and performance.
Too few threads may lead to underutilization of system resources, while too many threads can cause excessive context-switching and memory usage.
Initial Generation:
By restricting thread creation and destruction to the initial generation of the pool, the system avoids the overhead of frequently creating and destroying threads.
This results in more stable and predictable performance.
Context-Switching:
Context-switching is the process of storing the state of a thread and restoring the state of another thread.
While necessary for multitasking, excessive context-switching can degrade performance due to the overhead involved.

Thread Pools

- Smart implementations of a thread pool may use specific tasks and specific functions
  - The queue stores the tasks to solve but also the functions to solve it
    - Tasks and functions **may vary** for each run
  - Functions are usually called **callback** functions

Exercise

- Implement a thread pool
  - In C
    - Use the producer-and-consumer paradigm
    - Producers create tasks and insert them into the queue
    - Consumers get tasks from the queue and manage them
    - Task can be a randomly **generated strings** to be **capitalized** and display on standard output
  - In C++
    - Use generic tasks and thread (callback) functions

C Solution I

(Trivial) C Version

❖ Logic behavior of a producer-consumer scheme

Producer-Consumer Paradigm:
This paradigm involves two types of threads: producers and consumers.
Producers generate tasks and insert them into a shared queue.
Consumers retrieve tasks from the queue and process them.
Synchronization Mechanisms:
Semaphores:
full: Indicates the number of tasks in the queue.
empty: Indicates the number of empty slots in the queue.
Mutexes:
MEp: Ensures mutual exclusion for the producer when accessing the queue.
MEc: Ensures mutual exclusion for the consumer when accessing the queue.
Producer Logic:
The producer generates a task and waits if the queue is full (wait(empty)).
It then locks the producer mutex (wait(MEp)), inserts the task into the queue (enqueue(val)), and unlocks the mutex (signal(MEp)).
Finally, it signals that the queue is not empty (signal(full)).
Consumer Logic:
The consumer waits if the queue is empty (wait(full)).
It then locks the consumer mutex (wait(MEc)), retrieves a task from the queue (dequeue(&val)), and unlocks the mutex (signal(MEc)).
Finally, it signals that the queue is not full (signal(empty)) and processes the task (consume(val)).

init(full, 0): Initializes the full semaphore to 0, indicating that the queue is initially empty.
init(empty, SIZE): Initializes the empty semaphore to the size of the queue, indicating that the queue has SIZE empty slots.
init(MEp, 1): Initializes the MEp mutex to 1, ensuring mutual exclusion for the producer.
init(MEc, 1): Initializes the MEc mutex to 1, ensuring mutual exclusion for the consumer.

Initialization

```
init (full, 0);
init (empty, SIZE);
init (MEp, 1);
init (MEc, 1);
```

Producer

```
Producer () {
  int val;
  while (TRUE) {
    produce (&val);     Generate a task
    wait (empty);       wait for the queue if full
    wait (MEp);         Lock the producer mutex
    enqueue (val);      Insert the task into the queue
    signal (MEp);       Unlock the producer queue
    signal (full);      Signal that the queue is not empty
  }
}
```

Consumer

```
Consumer () {
  int val;
  while (TRUE) {
    wait (full);        Wait if the queue is empty
    wait (MEc);         Lock the consumer mutex
    dequeue (&val);     Retrieve a task from the queue
    signal (MEc);       Unlock the consumer mutex
    signal (empty);     Signal that the queue is not full
    consume (val);      Process the task
  }
}
```

C Solution II

The slide defines a thread_s structure that encapsulates all the necessary data for managing the producer-consumer thread pool.

```
typedef struct thread_s {
    int n, nP, nC;
    char **v;
    int size;
    int head;
    int tail;
    pthread_mutex_t meP;
    pthread_mutex_t meC;
    sem_t empty;
    sem_t full;
} thread_t;
```

n = Number of (total) tasks
nP = Number of tasks produced
nC = Number of tasks consumed

Task array (pointer to strings)

head: Index for dequeueing tasks (out).
tail: Index for enqueueing tasks (in).

Size, head and tail index of the queue (for in and out)

Mutexes for mutual exclusion for producers (meP) and consumers (meC).

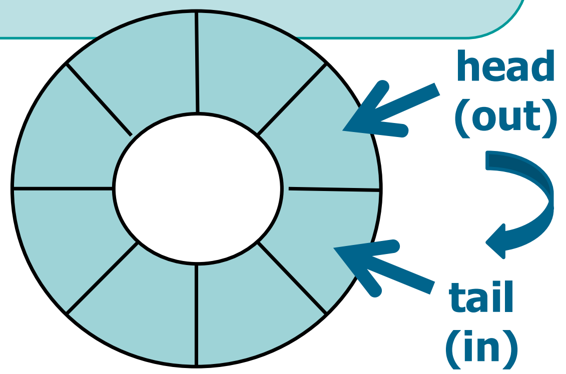
Mutual exclusion for producers and consumers

Semaphores to manage the empty and full states of the task queue.

Empty and full task queue

The task queue is implemented as a circular buffer, where head and tail indices wrap around when they reach the end of the buffer.

Circular buffer



Data Structure:
The thread_s structure encapsulates all the necessary data for managing the producer-consumer thread pool.
It includes counters for tasks, a task array, indices for the circular buffer, mutexes for mutual exclusion, and semaphores for managing the queue state.
Circular Buffer:
The circular buffer allows efficient use of the task queue by reusing the buffer space as tasks are enqueued and dequeued.
The head index points to the next task to be dequeued, and the tail index points to the next position to enqueue a new task.

C Solution III

Initialization:
Memory allocation for producer and consumer thread arrays.
Initialization of the thread_s structure members.
Initialization of mutexes and semaphores.

Initialization

Stop the pool after N tasks

```
tp = my_malloc (P, sizeof (pthread_t));
tc = my_malloc (C, sizeof (pthread_t));
thread_d.n = N;
thread_d.nP = thread_d.nC = 0;
thread_d.size = SIZE;
thread_d.v = my_malloc (thread_d.size, sizeof (char *));
thread_d.head = thread_d.tail = 0;
pthread_mutex_init (&thread_d.meP, NULL);
pthread_mutex_init (&thread_d.meC, NULL);
sem_init (&thread_d.empty, 0, SIZE);
sem_init (&thread_d.full, 0, 0);

for (i=0; i<P; i++)
    pthread_create(&tp[i], NULL, producer, (void *) &thread_d);
for (i=0; i<C; i++)
    pthread_create(&tc[i], NULL, consumer, (void *) &thread_d);

for (i=0; i<P; i++)
    pthread_join (tp[i], NULL);
for (i=0; i<C; i++)
    pthread_join (tc[i], NULL);
```

Allocate memory for producer (tp) and consumer (tc) thread arrays.
Initialize the thread_s structure members.
Initialize mutexes for producers (meP) and consumers (meC).
Initialize semaphores for empty (empty) and full (full) states of the task queue.

Create P producer threads, passing the thread_s structure as an argument.
Create C consumer threads, passing the thread_s structure as an argument.

Creating producer threads using pthread_create.
Creating consumer threads using pthread_create.

Use pthread_join to wait for all producer threads to complete.
Use pthread_join to wait for all consumer threads to complete.

Create the worker threads

Using pthread_join to wait for all producer and consumer threads to complete

Wait all threads

C Solution IV

Producer

```
static void *producer (void *arg) {
  thread_t *p; int goon = 1;
  p = (thread_t *) arg;
  while (goon == 1) {
    waitRandomTime (...);
    sem_wait (&p->empty);
    pthread_mutex_lock (&p->meP);
    if (p->nP > p->n) {
       goon = 0;
    } else {
       p->nP = p->nP + 1; p->v[p->tail] = generate();
       printf ("Producing %d: %s\n", p->nP, p->v[p->tail]);
       p->tail = (p->tail+1) % SIZE;
    }
    pthread_mutex_unlock (&p->meP);
    sem_post (&p->full);
  }
  pthread_exit ((void *) 1);
}
```

Protect queue

Protect producers

Stop after N tasks

See complete solution for details

Insert the task in the queue

C Solution V

Consumer

```
static void *consumer (void *arg) {
  thread_t *p; int goon = 1; char *str;
  p = (thread_t *) arg;
  while (goon == 1) {
    pthread_mutex_lock (&p->meC);
    if (p->nC > p->n) {
      goon = 0;
    } else {
      p->nC = p->nC + 1;
      sem_wait (&p->full);
      str = p->v[p->head]; convert (str);
      printf ("--- CONSUMING %d: %s\n", p->nC, str);
      free (str); p->head = (p->head+1) % SIZE;
      sem_post (&p->empty);
    }
    pthread_mutex_unlock (&p->meC);
  }
  pthread_exit ((void *) 1);
}
```

Protect consumers

Protect queue

See complete solution for details

C++ Solution I

(Complex) C++ Version (Partial)

The ThreadPool class encapsulates the functionality of a thread pool, managing a pool of worker threads that execute tasks.

In the general case, tasks are call-back functions that must be run by the working thread

```
class ThreadPool {
public:
  void Start();
  void QueueJob(const std::function<void()>& job);
  void Stop();
  void busy();

private:
  void ThreadLoop();

  // Tells threads to stop looking for jobs
  bool should_terminate = false;
  // Prevents data races to the job queue
  std::mutex queue_mutex;
  // Allows threads to wait on new jobs or termination
  std::condition_variable mutex_condition;
  std::vector<std::thread> threads;
  std::queue<std::function<void()>> jobs;
};
```

void Start(): Starts the thread pool by initializing and running the worker threads.

void QueueJob(const std::function<void()>& job): Adds a new job (task) to the job queue.

void Stop(): Stops the thread pool, signaling all threads to terminate.

void busy(): (Not defined in the slide, but presumably checks if the pool is busy).

void ThreadLoop(): The main loop for each worker thread, where they wait for and execute tasks.

A flag to signal threads to stop looking for jobs.

A condition variable to allow threads to wait for new jobs or termination signals.

A vector to hold the worker threads.

A queue to hold the jobs (tasks) to be executed by the worker threads.

ThreadPool Class:
The ThreadPool class manages a pool of worker threads that execute tasks from a job queue.
It provides methods to start the pool, add jobs to the queue, and stop the pool.
Synchronization Mechanisms:
queue_mutex ensures that only one thread can access the job queue at a time, preventing data races.
mutex_condition allows threads to wait for new jobs or termination signals, enabling efficient synchronization.

C++ Solution II

- Running threads in the pool
  - Each thread should run its own infinite loop, constantly waiting for new tasks to grab and execute

Summary
ThreadPool Class:
Manages a pool of worker threads that execute tasks from a job queue.
Provides methods to start the pool, add jobs to the queue, and stop the pool.
Uses synchronization mechanisms like mutexes and condition variables to manage access to the job queue and coordinate thread execution.
Start Method:
Initializes and runs the worker threads in the pool.
Determines the number of threads based on hardware concurrency.
Resizes the thread vector and creates worker threads to run the ThreadLoop method.

The Start method initializes and runs the worker threads in the pool.

```
void ThreadPool::Start() {
  // Max # of threads the system supports
  const uint32_t num_threads =
    std::thread::hardware_concurrency();

  threads.resize(num_threads);
  for (uint32_t i = 0; i < num_threads; i++) {
    threads.at(i) = std::thread(ThreadLoop);
  }

  return;
}
```

Define pool size

Run worker threads

const uint32_t num_threads = std::thread::hardware_concurrency(): Determines the maximum number of threads the system supports, based on the hardware concurrency.

threads.resize(num_threads): Resizes the threads vector to hold the appropriate number of worker threads.

A loop creates and starts each worker thread, assigning them to run the ThreadLoop method:

The method returns after initializing and starting all worker threads.

Determining Number of Threads:
std::thread::hardware_concurrency() returns the number of concurrent threads supported by the hardware, which is used to determine the size of the thread pool.
Resizing the Thread Vector:
threads.resize(num_threads) resizes the threads vector to hold the appropriate number of worker threads.
Creating Worker Threads:
The loop iterates over the number of threads, creating and starting each worker thread to run the ThreadLoop method.

C++ Solution III

- The infinite loop function
  - A loop waiting for the task queue to open up

```
void ThreadPool::ThreadLoop() {
  while (true) {
    std::function<void()> job;
    {
      std::unique_lock<std::mutex> lock(queue_mutex);
      mutex_condition.wait(lock, [this] {
        return !jobs.empty() || should_terminate;
      });
      if (should_terminate) {
        return;
      }
      job = jobs.front();
      jobs.pop();
    }
    job();
  }
}
```

ThreadLoop Function:
The ThreadLoop function is the main loop for each worker thread in the thread pool.
It continuously waits for tasks to be added to the job queue and executes them.

The worker threads wait on a condition variable (mutex_condition) for new tasks to be added to the job queue or for a termination signal.

Workers wait on a condition variable

std::unique_lock<std::mutex> lock(queue_mutex): Acquires a unique lock on the queue_mutex to ensure mutual exclusion when accessing the job queue.

Waits on the condition variable until there are jobs in the queue or a termination signal is received.

If should_terminate is true, the thread exits the loop and terminates.

Terminate worker thread

Retrieves the job from the front of the queue (job = jobs.front()) and removes it from the queue (jobs.pop()). Executes the job (job()).

Get a task from a queue (and erase it in the queue)

Waiting for Tasks:
The worker thread acquires a unique lock on the queue_mutex and waits on the condition variable until there are jobs in the queue or a termination signal is received.
Job Execution:
If there are jobs in the queue, the worker thread retrieves and removes the job from the queue and then executes it.

C++ Solution IV

Summary
ThreadLoop Function:
The ThreadLoop function is the main loop for each worker thread, continuously waiting for tasks to be added to the job queue and executing them.
Worker threads wait on a condition variable for new tasks or a termination signal, ensuring efficient synchronization and mutual exclusion.
QueueJob Function:
The QueueJob function adds a new job to the job queue and signals a worker thread to wake up and execute the job.
It uses a unique lock to ensure mutual exclusion when accessing the job queue and a condition variable to notify worker threads.
By understanding these functions, you can see how the thread pool manages the execution of tasks by worker threads, ensuring proper synchronization and efficient task execution. The ThreadLoop function handles the continuous execution of tasks, while the QueueJob function manages the addition of new tasks to the job queue

- Add a new job to the pool
  - Use a lock so that there is not a data race
  - Once the job is there, signal a condition variable to wake-up one worker

1. QueueJob Function:
The QueueJob function adds a new job (task) to the job queue.
2. Unique Lock:
std::unique_lock<std::mutex> lock(queue_mutex): Acquires a unique lock on the queue_mutex to ensure mutual exclusion when accessing the job queue.
3. Adding the Job:
The job is added to the job queue (jobs.push(job)).
4. Notifying Worker Threads:
mutex_condition.notify_one(): Signals one worker thread waiting on the condition variable that a new job is available.

```
void ThreadPool::QueueJob(const std::function<void()>& job) {
  {
  std::unique_lock<std::mutex> lock(queue_mutex);
  jobs.push(job);
  }
  mutex_condition.notify_one();
}
```

Insert a tack in the queue

Protect the task queue

Notify one working tread

C++ Solution V

- Use the thread pool
  - The function **busy** can be used in a while loop, such that the main thread can wait the thread pool to complete all the tasks before calling the thread pool destructor

1. Queueing a Job:
The QueueJob method is used to add a new job (task) to the thread pool's job queue.
Example usage: thread_pool->QueueJob([] { /* ... */ });
2. Busy Method:
The busy method checks if the thread pool is currently busy (i.e., if there are any jobs in the queue).
This method can be used in a while loop to allow the main thread to wait for the thread pool to complete all tasks before proceeding.
3. Polling for Job Completion:
The busy method uses a unique lock to ensure mutual exclusion when accessing the job queue.
It returns true if the job queue is empty, indicating that all jobs have been processed.

```
thread_pool->QueueJob([] { /* ... */ });


void ThreadPool::busy() {
  bool poolbusy;
  {
    std::unique_lock<std::mutex> lock(queue_mutex);
    poolbusy = jobs.empty();
  }
  return poolbusy;
}
```

Polling to know whether all job have been done

Queueing a Job:
The QueueJob method is called with a lambda function representing the job to be executed.
This job is added to the job queue, and a worker thread will pick it up and execute it.
Busy Method:
The busy method acquires a unique lock on the queue_mutex to ensure mutual exclusion.
It checks if the job queue is empty and returns the result.
This method can be used to poll the status of the thread pool, allowing the main thread to wait until all jobs are completed.

C++ Solution VI

- Stop the pool

1. Stop Method:
The Stop method is used to stop the thread pool and ensure that all worker threads terminate gracefully.
2. Setting Termination Flag:
The should_terminate flag is set to true to signal all worker threads to stop looking for new jobs.
3. Notifying All Threads:
mutex_condition.notify_all(): Signals all worker threads waiting on the condition variable that they should terminate.
4. Joining Worker Threads:
A loop iterates over all worker threads, calling join on each one to wait for their completion.
threads.clear(): Clears the threads vector after all threads have been joined.

Setting Termination Flag:
The should_terminate flag is set to true inside a unique lock to ensure mutual exclusion.
This flag signals all worker threads to stop looking for new jobs and terminate.
Notifying All Threads:
The notify_all method of the condition variable is called to wake up all worker threads.
This ensures that all threads receive the termination signal and exit their loops.
Joining Worker Threads:
A loop iterates over all worker threads, calling join on each one to wait for their completion.
This ensures that the main thread waits for all worker threads to finish before proceeding.
The threads vector is cleared after all threads have been joined.

```
void ThreadPool::Stop() {
  {
    std::unique_lock<std::mutex> lock(queue_mutex);
    should_terminate = true;
  }
  mutex_condition.notify_all();
  for (std::thread& active_thread : threads) {
    active_thread.join();
  }
  threads.clear();
}
```

Notify all threads

Join all working threads

Conclusions

- Thread pools are used to limit the cost of re-creating threads over and over again
  - There are languages / environments in which thread pools have an explicit support
    - Windows API, C++
  - Smart implementations may use a **callback** function
    - The queue stores the tasks to solve and the functions to solve them
      - Functions and task can differ
      - Functions are usually called callback function