Synchronization

Thread Throttles and Pools

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

License Information

This work is licensed under the license

Introduction

❖ 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
  - 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**

Introduction

❖ **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

❖ 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 **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<<N

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

N threads may run

```
...

sem_wait (sem);

... CS ...

sem_post (sem);

...
```
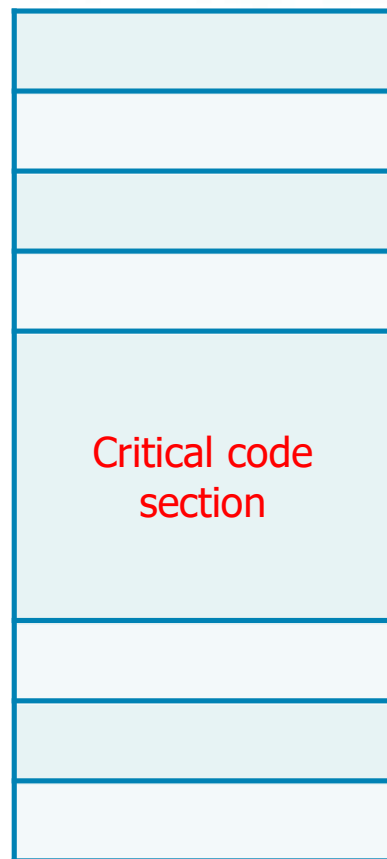
Critical code section

Less than **N** threads may run

N threads may run

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

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

Set it to be "large enough"

▪ Notice that the maximum number of Ts allowed is set once and only once at initialization

❖ 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

Example (part A)

Pseudo-code

Task

Critical code section
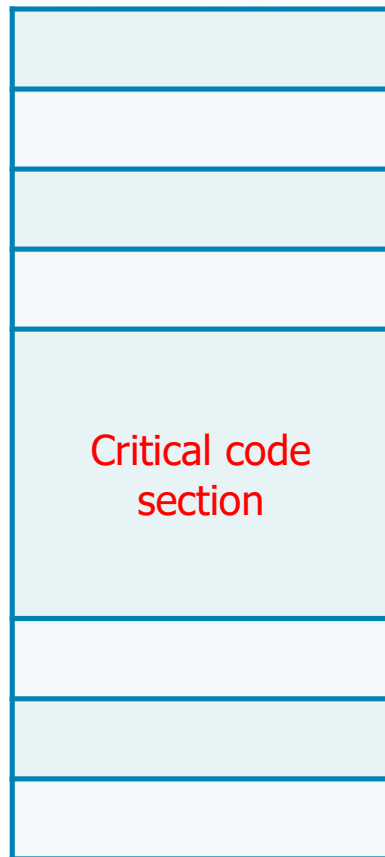
```
sem_init (sem, n)
```

```
...

sem_wait (sem);

...

sem_post (sem);

...
```

Standard workers

May cause threads to deadlock (see u02s02 ex 09)

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

Expensive workers

Example (part B)

Pseudo-code

Task

Critical code section
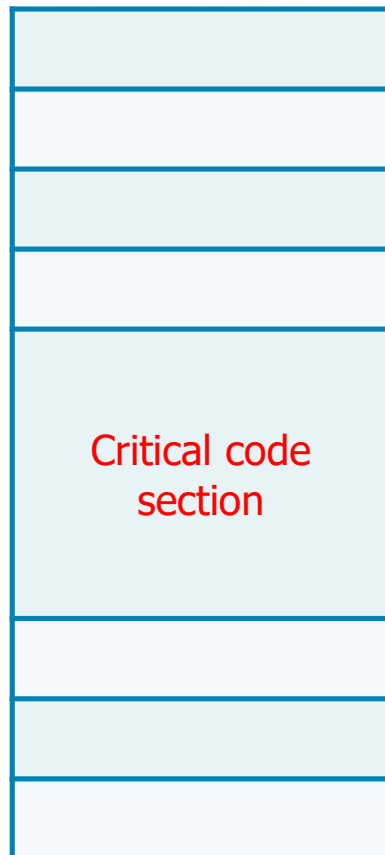
```
sem_init (sem, n)
mutex_init (m, 1)
```

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

Standard workers

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

Example (part C)

Pseudo-code

Task


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

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

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.
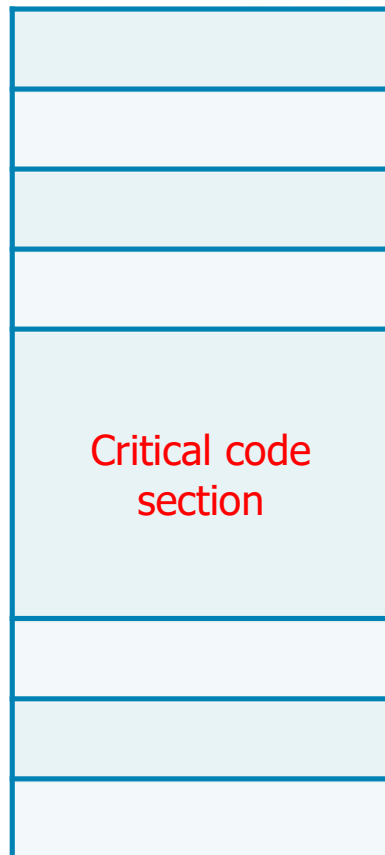
Critical code section

```
...
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

Thread Pools

Main T:
Produces tasks

X

Worker Ts:
Consumers
(Thread Pool)

Task Queue
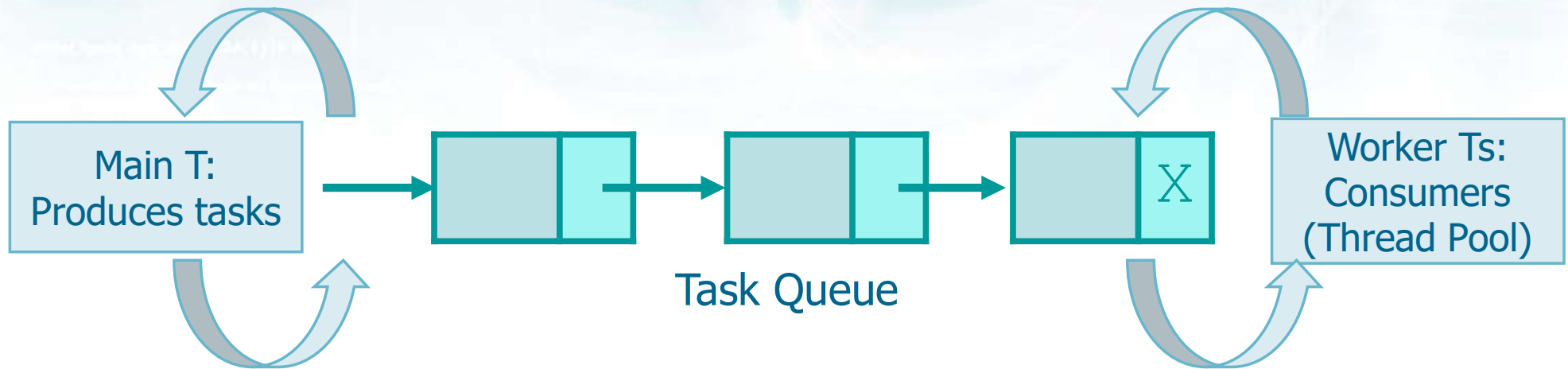
❖ A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution

➢ 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

Main T:
Produces tasks

X

Worker Ts:
Thread Pool
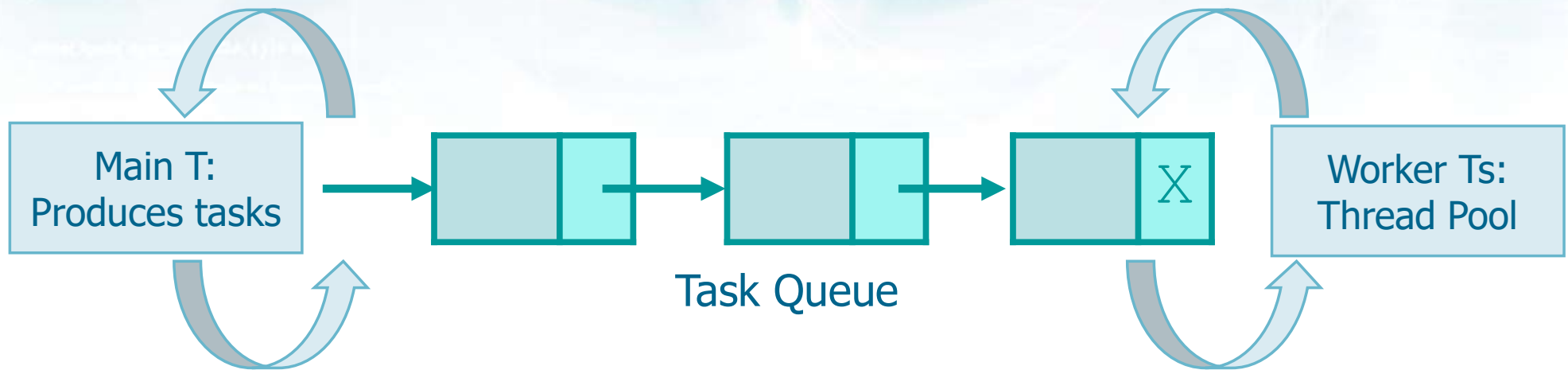
Task Queue

❖ **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

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 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

Initialization

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

Producer

```
Producer () {
  int val;
  while (TRUE) {
    produce (&val);
    wait (empty);
    wait (MEp);
    enqueue (val);
    signal (MEp);
    signal (full);
  }
}
```

Consumer

```
Consumer () {
  int val;
  while (TRUE) {
    wait (full);
    wait (MEc);
    dequeue (&val);
    signal (MEc);
    signal (empty);
    consume (val);
  }
}
```

C Solution II

```
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
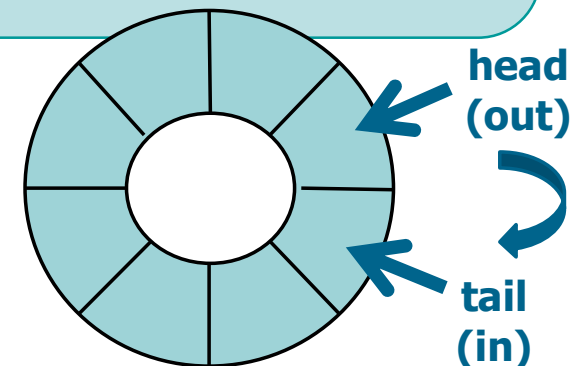nC = Number of tasks consumed

Task array (pointer to strings)

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

Mutual exclusion for producers and consumers

Empty and full task queue

Circular buffer

head
(out)

tail
(in)

C Solution III

Initialization

```
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);
```

Stop the pool after N tasks

Create the worker threads

Wait all threads

C Solution IV

Producer

```c
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

Protect consumers

Protect queue

See complete solution for details

```c
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);
}
```

C++ Solution I

(Complex) C++ Version (Partial)

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

```cpp
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;
};
```

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

```cpp
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

C++ Solution III

❖ The infinite loop function

➢ A loop waiting for the task queue to open up

```cpp
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();
  }
}
```

Workers wait on a condition variable

Terminate worker thread

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

C++ Solution IV

❖ 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

```
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

```
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

C++ Solution VI

❖ Stop the pool

```
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