System and Device Programming

Thread Synchronization

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- For advanced thread synchronization it is possible to use the following strategies
  - Semaphore throttles
  - Barriers
  - Thread pools

Use to limit the number of running threads in specific code sections

Synchronization point for multiple threads

Used with "lots" of threads, when thread context switching is very time consuming

Semaphore Throttles

- Scenario
  - **N** worker Ts contend for a shared resource
    - They may use a CS, a mutex or a semaphore
  - Performance degradation is severe when **N** increases and contention is **high**
- Target
  - Improve performance
  - Retain the simplicity of the original approach
- "Semaphore throttles"
  - Use a semaphore to **fix** the **maximum** amount of running Ts

Semaphore Throttles

- The boss T
  - Creates a semaphore
  - Sets the maximum value to a “reasonable number”
    - Example: 4, 8, 16
    - Its value depends on the number of core or processors
    - It is a tunable value
- Worker Ts must get a semaphore unit before working
  - Wait on the semaphore throttles
  - Then, wait on the CS or mutex or semaphore, etc. (to access critical section areas)

Semaphore Throttles

Worker loop

```
while (TRUE) {
  wait (sem);
  ...
  wait (mutex);
  ...
  signal (mutex);
  ...
  signal (sem);
}
```

Semaphore Throttles

Standard Critical Section

If the max count for hSem is 1 (at most 1 worker T), hMutex is useless

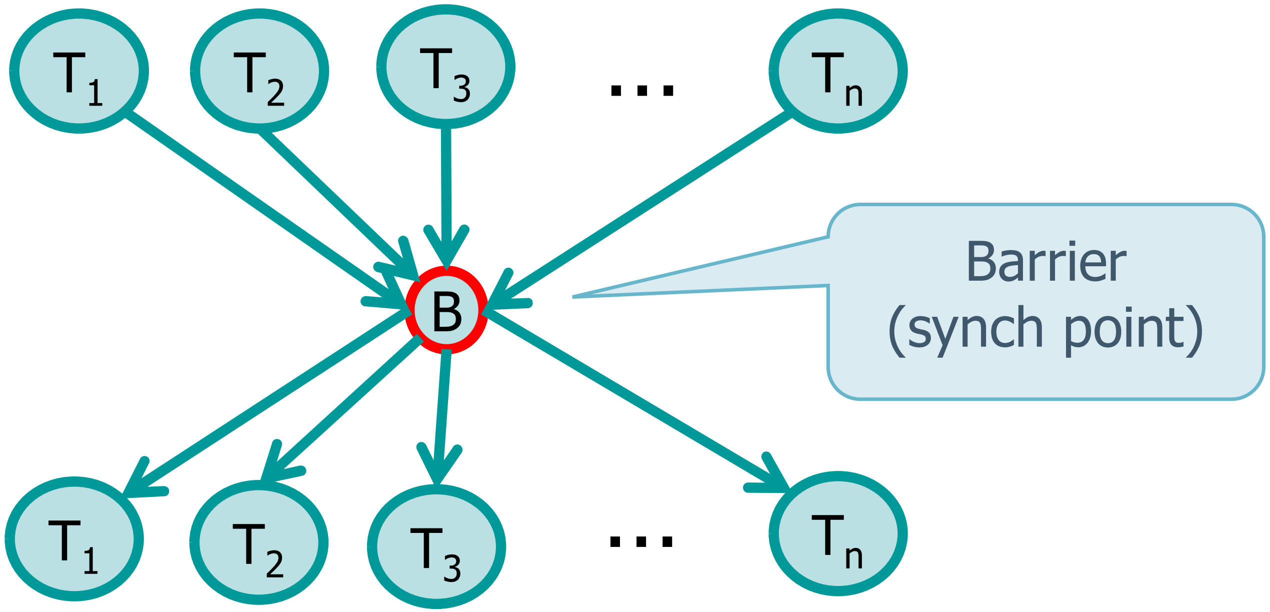
Semaphore Throttles

- Variations
  - Some workers may acquire multiple units
    - The idea is that workers than use more resources wait more on the throttles
  - Caution
    - Pay attention to deadlock risks
- The boss T may tune dynamically the worker Ts behavior
  - Decreases or increases the number of active workers
    - By waiting or releasing semaphore units
    - Anyhow, the maximum number of Ts allowed is set once and only once at initialization

Set it to be "large enough"

Barriers

- Barriers can be used to coordinate multiple threads working in parallel
  - A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there



Barriers

- Barriers generalize the pthread_join function
  - pthread_join acts as a barrier to allow one thread to wait until another thread than this
  - Barriers allow an arbitrary number of threads to wait until all of the threads have completed processing
  - The threads don't have to exit, as they can continue working after all threads have reached the barrier

pthread_barrier_init

```
int pthread_barrier_init (
  pthread_barrier_t *restrict barrier,
  const pthread_barrierattr_t *restrict attr,
  unsigned int count
);
```

Always include pthread.h

- We can use the pthread_barrier_init function to initialize a barrier
  - The **count** argument to specify the number of threads that must reach the barrier before all of the threads will be allowed to continue

pthread_barrier_init

- We use the attr argument to specify the attributes of the barrier object
  - The default attribute is NULL

```
int pthread_barrier_init (
  pthread_barrier_t *restrict barrier,
  const pthread_barrierattr_t *restrict attr,
  unsigned int count
);
```

pthread_barrier_wait

```
int pthread_barrier_wait (
  pthread_barrier_t *barrier
);
```

- We use the pthread_barrier_wait function to indicate that a thread is done with its work and it is ready to wait for all the other threads to catch up

pthread_barrier_destroy

```
int pthread_barrier_destroy (
  pthread_barrier_t *barrier
);
```

- We can use the pthread_barrier_destroy function to deinitialize a barrier
  - Any resource allocated for the barrier will be freed

Example

Use a barrier to synchronize 2+1 threads

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_barrier_t bar;

void *f() {
  printf ("1\n");
  pthread_barrier_wait(&bar);
  printf ("2\n");
}
```

Example

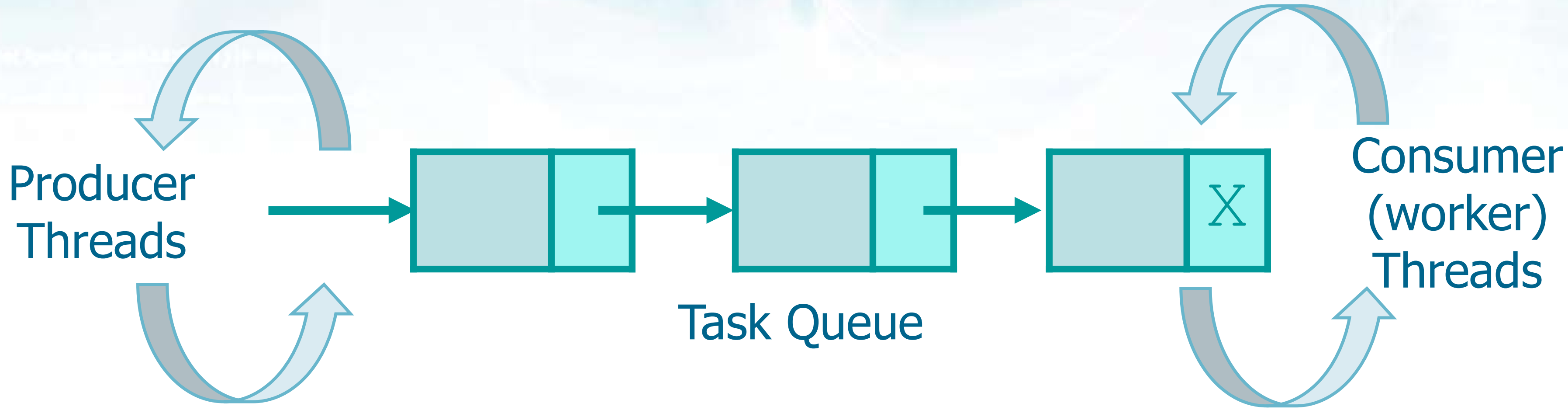
```
int main () {
  pthread_t th1, th2;

  pthread_barrier_init (&bar, NULL, 3);

  pthread_create (&th1, NULL, f, NULL);
  pthread_create (&th2, NULL, f, NULL);

  pthread_barrier_wait (&bar);

  pthread_join (th1, NULL);
  pthread_join (th2, NULL);

  pthread_barrier_destroy(&bar);
  return 0;
}
```

2+1
threads

Thread Pools

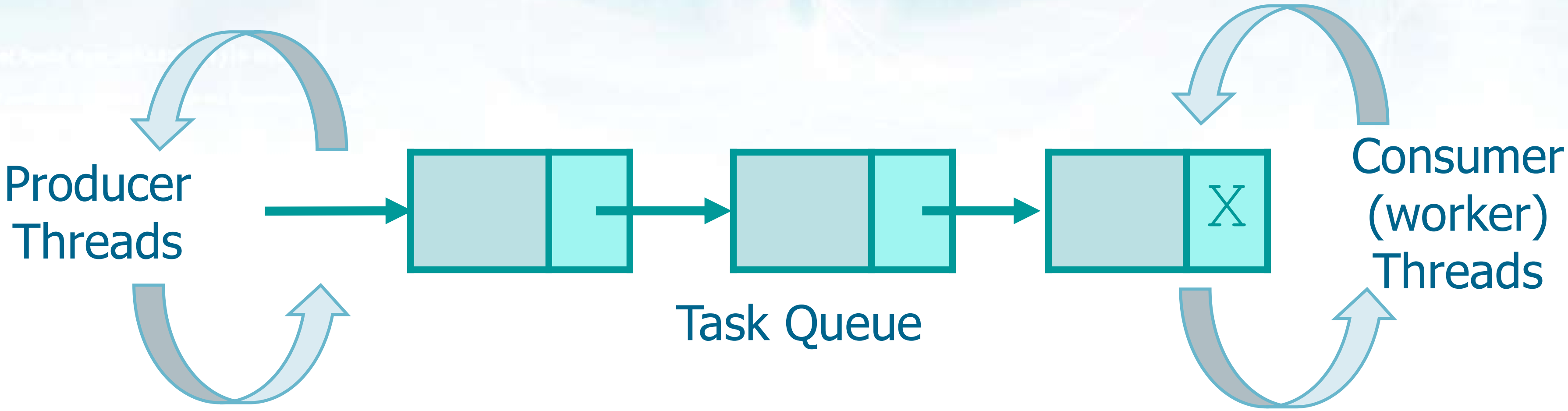
- Creating and destroying a thread and its associated resources can be an expensive process in terms of time
- A **thread pool** is a design pattern for achieving concurrency and reducing overheads
  - They are also called a **replicated workers** or **worker-crew model**

Thread Pools



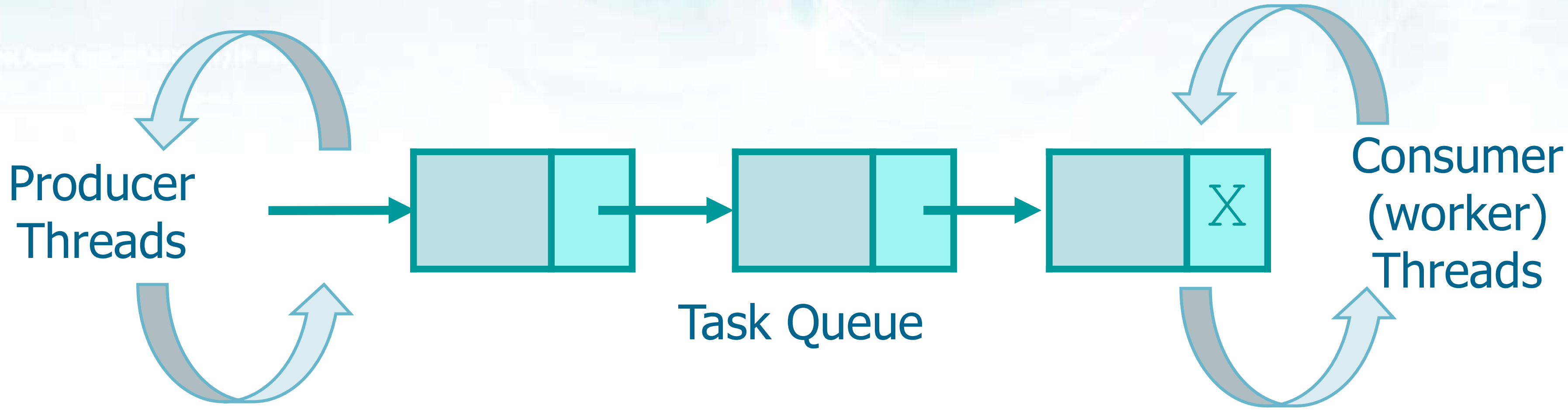
- A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution
  - One or more threads generate the task
  - Tasks are enqueue in a (FIFO) queue
    - Dynamic list, circular array, etc.
  - Tasks are solved by worker threads in the thread pool

Thread Pools



- The user
  - Initializes
    - A "thread pool" (or task) queue
    - The "working theads"
  - Creates "work objects" (or "tasks")
  - Inserts tasks into the queue
- Worker threads
  - Get tasks from the queue

Thread Pools



- The size of a thread pool is the number of threads kept in reserve for executing tasks
  - It is usually a tunable parameter of the application, adjusted to optimize program performance
  - Deciding the optimal thread pool size is crucial to optimize performance

Thread Pools

- Performance
  - One benefit of a thread pool over creating a new thread for each task is that thread creation and destruction overhead is restricted to the initial creation of the pool
    - This may result in better performance and better system stability
  - An excessive number of threads in reserve may waste memory and context-switching between the runnable threads invokes performance penalties

Thread Pools

- There are languages / environments in which thread pools have an explicit support
  - In windows API
  - C++
  - Etc.
- In the general case tasks have a callback function which is run when the thread runs

Exercise

- Use the producer-and-consumer paradigm to implement a thread pool
  - Producers create tasks and insert them into the queue
    - Each task is a randomly generated string on random size
  - Consumers get tasks from the queue and solve them
    - Each solution cornesponds to capitalize the string and display it on standard output

Please, see C file for the complete solution

Solution

```
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 tasks
nP = Number of tasks produced
nC = Number of tasks consumed

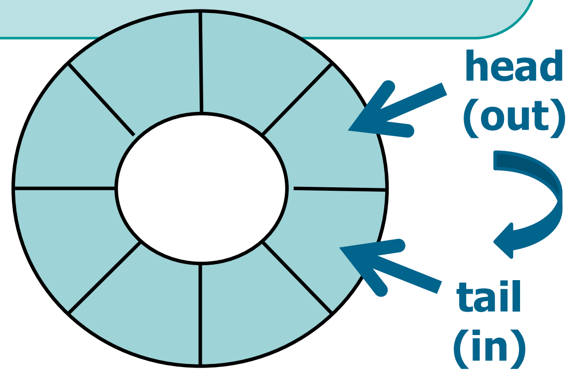
Task array (pointer to strings)

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



Solution

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);
```

Solution

```
static void *producer (void *arg) {
  thread_t *p; int goon = 1;
  p = (thread_t *) arg;
  while (goon == 1) {
    waitRandomTime (3);
    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);
}
```

Producer

Solution

```
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);
}
```

Consumer

Conclusions

- Thread throttle
  - Used to limit the number of threads running on the more expensive program sections
- Barriers
  - Used to coordinate multiple threads working in parallel
    - You want all threads to wait until everyone has arrived at a certain point
    - A simple sempahore would do the exact opposite, i.e., each thred would keep running and the last one will go to sleep

Conclusions

- Thread pools
  - Used to limit the cost of re-creating threads over and over again