Thread Pool in C

John

Introduction

When I was writing Poddown I needed a thread pool and I needed one that is cross platform. Since it's a lightweight app I didn't want to include a big third party threading library. So I wrote my own.

Why a Thread Pool

Creating threads can be quite expensive. Typically each thread is going to do essentially the same thing so it's good to keep reusing them. Threads are actually quite heavy and creating or destroying threads takes time away from what you're trying to accomplish.

Another benefit of thread pools is they keep the system from becoming overloaded. They allow a limit to the number of threads. Tasks are queued and only run when a thread is available.

In Poddown, it downloads multiple podcasts simultaneously. Obviously, it uses threads but I was worried about having way to many threads started at one time. If there are 250 podcasts that need to be downloaded, that's a lot of threads and a lot of data trying to be pulled down over my anemic internet connection. The thread pool allows me to easily have a configurable limit for the number of podcasts being downloaded in parallel.

Requirements

A, good, thread pool keeps a set number of threads running and waiting to do something. The pool could be designed to scale up with the amount of work you need to do but I prefer specifying a fixed number of threads. A good way to choose this number is to use the number of cores/processors on the system + 1.

At the heart of a thread pool are threads. I'm going to use pthreads but don't worry! I wrote a simple wrapper for Windows that provides a pthread API on Windows. It's a very slim wrapper so it's basically drop in and doesn't have any external requirements. I'm using this with Poddown by the way.

```
#ifndef __TPOOL_H__
#define __TPOOL_H__

#include <stdbool.h>
#include <stddef.h>

struct tpool;
typedef struct tpool tpool_t;

typedef void (*thread_func_t)(void *arg);

tpool_t *tpool_create(size_t num);
void tpool_destroy(tpool_t *tm);

bool tpool_add_work(tpool_t *tm, thread_func_t func, void *arg);
void tpool_wait(tpool_t *tm);

#endif /* __TPOOL_H__ */
```

The two functions of importance are `tpool_add_work` which adds work to the queue for processing and

`tpool_wait` which blocks until all work has been completed.

Object data

```
struct tpool_work {
    thread_func_t      func;
    void              *arg;
    struct tpool_work *next;
};
typedef struct tpool_work tpool_work_t;
```

The work queue is a simple linked list which stores the function to call and its arguments. A result queue could be implemented in the same way. However, I prefer to have the results handled by the work function because it gives a bit more flexibility. For example, not all work needs a result and not all results need thread synchronization.

```
struct tpool {
    tpool_work_t    *work_first;
    tpool_work_t    *work_last;
    pthread_mutex_t  work_mutex;
    pthread_cond_t   work_cond;
    pthread_cond_t   working_cond;
    size_t           working_cnt;
    size_t           thread_cnt;
    bool             stop;
};
```

Since the work queue is implemented as a linked list `work_first` and `work_last` are used to push and pop work objects. There is a single `mutex` (`work_mutex`) which is used for all locking.

We need two conditionals, `work_cond` signals the threads that there is work to be processed and `working_cond` signals when there are no threads processing. We combine these with `working_cnt` to know how many threads are actively processing work. We're not checking if the threads running because they'll always be running as part of the pool. They're just not using resources if they're not processing any work.

Finally, we track how many threads are alive using `thread_cnt`. This helps us prevent running threads from being destroyed prematurely. This is used in conjunction with `stop` which actually stops the threads.

Implementation

Work Data Object

```
static tpool_work_t *tpool_work_create(thread_func_t func, void *arg)
{
    tpool_work_t *work;

    if (func == NULL)
        return NULL;

    work       = malloc(sizeof(*work));
    work->func = func;
    work->arg  = arg;
    work->next = NULL;
    return work;
}

static void tpool_work_destroy(tpool_work_t *work)
{
    if (work == NULL)
        return;
    free(work);
}
```

Simple helpers for creating and destroying work objects.

```
static tpool_work_t *tpool_work_get(tpool_t *tm)
{
    tpool_work_t *work;

    if (tm == NULL)
        return NULL;

    work = tm->work_first;
    if (work == NULL)
        return NULL;

    if (work->next == NULL) {
        tm->work_first = NULL;
        tm->work_last  = NULL;
    } else {
        tm->work_first = work->next;
    }

    return work;
}
```

Work will need to be pulled from the queue at some point to be processed. Since the queue is a linked list this handles not only pulling an object from the list but also maintaining the list `work_first` and `work_last` references for us.

The Worker Function

```
static void *tpool_worker(void *arg)
{
    tpool_t      *tm = arg;
    tpool_work_t *work;

    while (1) {
        pthread_mutex_lock(&(tm->work_mutex));

        while (tm->work_first == NULL && !tm->stop)
            pthread_cond_wait(&(tm->work_cond), &(tm->work_mutex));

        if (tm->stop)
            break;

        work = tpool_work_get(tm);
        tm->working_cnt++;
        pthread_mutex_unlock(&(tm->work_mutex));

        if (work != NULL) {
            work->func(work->arg);
            tpool_work_destroy(work);
        }

        pthread_mutex_lock(&(tm->work_mutex));
        tm->working_cnt--;
        if (!tm->stop && tm->working_cnt == 0 && tm->work_first == NULL)
            pthread_cond_signal(&(tm->working_cond));
        pthread_mutex_unlock(&(tm->work_mutex));
    }

    tm->thread_cnt--;
    pthread_cond_signal(&(tm->working_cond));
    pthread_mutex_unlock(&(tm->work_mutex));
    return NULL;
}
```

This is the heart and soul of the pool and is where work is handled. At a high level this waits for work and

processes it.

```
    while (1) {
```

This will keep the tread running and as long as it doesn't exit it can be used.

```
        pthread_mutex_lock(&(tm->work_mutex));
```

The first thing that happens is locking the `mutex` so we can be sure nothing else manipulates the pool's members.

```
        while (tm->work_first == NULL && !tm->stop)
            pthread_cond_wait(&(tm->work_cond), &(tm->work_mutex));
```

Check if there is any work available for processing and we are still running. We'll wait in a conditional until we're signaled and run our check again. Remember the conditional automatically unlocks the `mutex` so others can acquire the lock. When the conditional is signaled it will relock the `mutex` automatically so we don't have to. We're looping here instead of using an `if` statement to handle spurious wakeups.

```
        if (tm->stop)
            break;
```

Now we check if the pool has requested that all threads stop running and exit. We want to keep holding the lock so we can modify a few things outside of the run loop. The `mutex` will be unlocked before the thread exists. The stop check is all the way up here because we want to stop before pulling any work.

```
        work = tpool_work_get(tm);
        tm->working_cnt++;
        pthread_mutex_unlock(&(tm->work_mutex));
```

Once the thread was signaled there is work, we'll pull some from the queue and increment `working_cnt` so the pool knows a thread is processing. The `mutex` is unlocked so other threads can pull and process work. We want the work processing to happen in parallel it doesn't make sense to hold the lock while work is processing. The lock is only there to synchronize pulling work from the queue.

```
        if (work != NULL) {
            work->func(work->arg);
            tpool_work_destroy(work);
        }
```

If there was work, process it and destroy the work object. It is possible that there was no work at this point so there isn't anything that needs to be done. For example, lets say there is one piece of work and 4 threads. All threads are signaled there is work. Each one will unblock one at a time and pull the work, so the first thread will acquire the lock pull the work, lock and start processing. The next three will unblock and pull nothing from the queue because it's empty.

```
        pthread_mutex_lock(&(tm->work_mutex));
        tm->working_cnt--;
        if (!tm->stop && tm->working_cnt == 0 && tm->work_first == NULL)
            pthread_cond_signal(&(tm->working_cond));
        pthread_mutex_unlock(&(tm->work_mutex));
```

Finally, once the work has been processed (or not if there wasn't any) the `mutex` will be locked again and `working_cnt` is decreased because the work is done. If there are no threads working and there are no items in the queue a signal will be sent to inform the wait function (in case someone's waiting) to wake up.

Since we know that only one thread can hold the lock, then only the last thread to finish it's work can decrement `working_cnt` to zero. This way we know there aren't any threads working.

```
    tm->thread_cnt--;
    pthread_cond_signal(&(tm->working_cond));
```

```
    pthread_mutex_unlock(&(tm->work_mutex));
    return NULL;
```

At the top of the run loop we break out on stop and this is where it leads us. We decrement the thread count because this thread is stopping.

Then we signal `tpool_wait` that a thread has exited. We need to do this here because `tpool_wait` will wait for all threads to exit when stopping. If we're here it's because `tpool_destroy` was called and it's waiting for `tpool_wait` to exit. Once all threads have exited `tpool_wait` will return allowing `tpool_destroy` to finish.

We unlock the `mutex` last because everything is protected by it, and we're guaranteed `tpool_wait` won't wake up before the thread finishes it's cleanup. The signal above only runs after we unlock.

Pool create and destroy

```
tpool_t *tpool_create(size_t num)
{
    tpool_t   *tm;
    pthread_t  thread;
    size_t     i;

    if (num == 0)
        num = 2;

    tm             = calloc(1, sizeof(*tm));
    tm->thread_cnt = num;

    pthread_mutex_init(&(tm->work_mutex), NULL);
    pthread_cond_init(&(tm->work_cond), NULL);
    pthread_cond_init(&(tm->working_cond), NULL);

    tm->work_first = NULL;
    tm->work_last  = NULL;

    for (i=0; i<num; i++) {
        pthread_create(&thread, NULL, tpool_worker, tm);
        pthread_detach(thread);
    }

    return tm;
}
```

When creating the pool the default is two threads if zero was specified. Otherwise, the caller specified number will be used. It's a good idea for us to have a default but two might be a little low. This goes back to the idea of using the number of core/processors + 1 as the default. If you're using the my thread wrapper it has a cross platform function, `pcthread_get_num_procs`, that will give this to you.

```
    for (i=0; i<num; i++) {
        pthread_create(&thread, NULL, tpool_worker, tm);
        pthread_detach(thread);
    }
```

The requested number of threads are started and `tpool_worker` is specified as the thread function. The threads are detached so they will cleanup on exit. There is no need to store the thread ids because they will never be accessed directly. If we wanted to implement some kind of force exit instead of having to wait then we'd need to track the ids.

```
void tpool_destroy(tpool_t *tm)
{
    tpool_work_t *work;
    tpool_work_t *work2;
```

```
    if (tm == NULL)
        return;

    pthread_mutex_lock(&(tm->work_mutex));
    work = tm->work_first;
    while (work != NULL) {
        work2 = work->next;
        tpool_work_destroy(work);
        work = work2;
    }
    tm->work_first = NULL;
    tm->stop = true;
    pthread_cond_broadcast(&(tm->work_cond));
    pthread_mutex_unlock(&(tm->work_mutex));

    tpool_wait(tm);

    pthread_mutex_destroy(&(tm->work_mutex));
    pthread_cond_destroy(&(tm->work_cond));
    pthread_cond_destroy(&(tm->working_cond));

    free(tm);
}
```

Once all outstanding processing finishes the pool object is destroyed.

```
    pthread_mutex_lock(&(tm->work_mutex));
    work = tm->work_first;
    while (work != NULL) {
        work2 = work->next;
        tpool_work_destroy(work);
        work = work2;
    }
    tm->work_first = NULL;
```

We are throwing away all pending work but the caller really should have dealt with this situation since they're the one that called `destroy`. Typically, destroy will only be called when all work is done and nothing is processing. However, it's possible someone is trying to force processing to stop. In which case there will be work queued and we need to get clean it up.

This only empties the queue and since it's in the `work_mutex` we have full access to the queue. This won't interfere with anything currently processing because the threads have pulled off the work they're working on. Any threads trying to pull new work will we've finished clearing out any pending work.

```
    tm->stop = true;
    pthread_cond_broadcast(&(tm->work_cond));
    pthread_mutex_unlock(&(tm->work_mutex));
```

Once we've cleaned up the queue we'll tell the threads they need to stop.

```
    tpool_wait(tm);
```

Some threads may have already been running and are currently processing so we need to wait for them to finish. This is where a force exit parameter could be implemented which would kill all threads instead of waiting on them to finish but this is a bad idea. For example, if you're writing to a database you could corrupt it.

Even though we could add force quit now and never use it we probably don't want to have it at all because people might not realize they shouldn't use it. Wait, why would we even implement something that we won't use and we don't want people to use? Maybe that's why we didn't do it in the first place.

Adding work to the queue

```
bool tpool_add_work(tpool_t *tm, thread_func_t func, void *arg)
{
    tpool_work_t *work;

    if (tm == NULL)
        return false;

    work = tpool_work_create(func, arg);
    if (work == NULL)
        return false;

    pthread_mutex_lock(&(tm->work_mutex));
    if (tm->work_first == NULL) {
        tm->work_first = work;
        tm->work_last  = tm->work_first;
    } else {
        tm->work_last->next = work;
        tm->work_last       = work;
    }

    pthread_cond_broadcast(&(tm->work_cond));
    pthread_mutex_unlock(&(tm->work_mutex));

    return true;
}
```

Adding to the work queue consists of creating a work object, locking the `mutex` and adding the object to the liked list.

Waiting for processing to complete

```
void tpool_wait(tpool_t *tm)
{
    if (tm == NULL)
        return;

    pthread_mutex_lock(&(tm->work_mutex));
    while (1) {
        if (tm->work_first != NULL || (!tm->stop && tm->working_cnt != 0) || (tm->stop && tm->th
            pthread_cond_wait(&(tm->working_cond), &(tm->work_mutex));
        } else {
            break;
        }
    }
    pthread_mutex_unlock(&(tm->work_mutex));
}
```

This is a blocking function that will only return when there is no work. The `mutex` is locked and we wait in a conditional if there are any threads processing, or f there is still work to do, or if the threads are stopping and not all have exited. The retry is a safety measure in case of spurious wake ups. Once there is nothing processing, return so the caller can continue.

Testing

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

#include "tpool.h"

static const size_t num_threads = 4;
static const size_t num_items   = 100;

void worker(void *arg)
```

```
{
    int *val = arg;
    int  old = *val;

    *val += 1000;
    printf("tid=%p, old=%d, val=%d\n", pthread_self(), old, *val);

    if (*val%2)
        usleep(100000);
}

int main(int argc, char **argv)
{
    tpool_t *tm;
    int     *vals;
    size_t   i;

    tm   = tpool_create(num_threads);
    vals = calloc(num_items, sizeof(*vals));

    for (i=0; i<num_items; i++) {
        vals[i] = i;
        tpool_add_work(tm, worker, vals+i);
    }

    tpool_wait(tm);

    for (i=0; i<num_items; i++) {
        printf("%d\n", vals[i]);
    }

    free(vals);
    tpool_destroy(tm);
    return 0;
}
```

For fun we'll use 4 thread and a 100 work items.

Here we have the worker function the pool will call. This will take an `int` pointer add 1000 and print the values before and after. The thread id will also be printed to show the four threads being used. Finally, the `usleep` is used to try and get the threads to run in a non-linear order. A fast enough system will have each thread finish one after another making it appear as if threads aren't being used (that's a good thing).

All main is doing is creating an array of 100 `int`s and add them as work to the queue. Then we just need to wait for all processing to finish and print the result.

Further Exploration

This implementation uses a linked list so it can queue an infinite amount of work. Another option would be to set a fixed size queue and have `tpool_add_work` block if there are no open slots. If you're worried about memory usage you should think about this.