Concurrency Design Patterns

Using code for this lab session

All source code is provided in the archive `threadpool.tar.gz`. To open this archive, issue the following command "`tar zxvf threadpool.tar.gz`" this will create the directory `threadpool` with all source code.
A makefile is provided to compile all source code. Issue "`make help`" for more details.
When compiling, you may see in the console: `foo.c:9: note: #pragma message: TODO - Exercise X`. This indicates a place where you need to correct the source code for a given exercise.

Note: the code provided for this lab session has been tested for Linux OS only. It relies on mechanisms from Unix and POSIX libraries that are not available on Mac OS X or Windows.

Assignements

You are tasked to perform all exercises, and write an implementation report. This implementation report is made of two files: a text file describing what you did, and the annotated source code.
Hard deadline: one week after the day of the lab session.

1 The Thread pool design pattern

In this lab[1], we consider the implementation of the "ThreadPool"design pattern. This pattern is at the heart of many server implementation.

1.1 Introduction

A concurrency pattern is a category of design pattern, used in software engineering, to identify methods that a computer program uses to handle multi-threaded tasks.
These patterns help software architects to create and enhance an interface between objects, synchronize shared memory between threads, make data thread-safe, monitor progress and manage threads and events, etc.

The Thread Pool pattern (see figure 1) is a design pattern, used in software engineering to organise the processing of a large number of queued tasks through a smaller/limited number of threads. Results can also be queued. When a thread finishes its task it requests another. If none are available, the thread can wait or terminate. This pattern is associated to various analysis techniques, in particular to evaluate the performance of a system where incoming requests are represented by a general law λ, and the processing time of each server process U_i, one can derive the average time to process a request μ.

In the following, we will consider the implementation of this pattern, and its tuning for simple Web servers.

[1] Part of this lab is based on materials from `https://en.wikipedia.org/wiki/Thread_pool_pattern` and `http://social.technet.microsoft.com/wiki/contents/articles/13245.thread-pool-design-pattern.aspx`

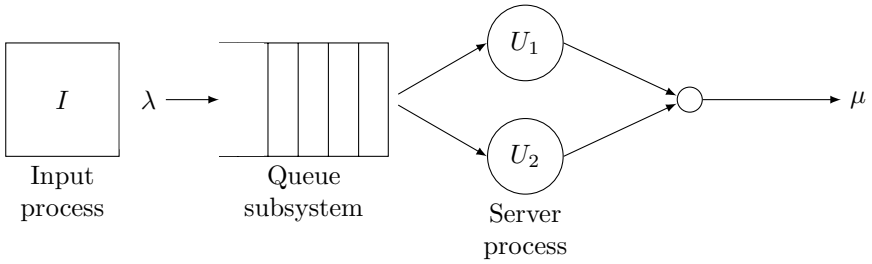

Figure 1: An abstract view of a threadpool

1.2 Performance considerations

The size of the thread pool is a tuning parameter that allows fine tuning of resource usage and responsiveness, for best results. Pooling threads helps improve performance by saving time creating the thread, however, too many idle threads can consume resources. The number of threads is based on percentage used of CPU, queued requests, and/or the number of processors in the system.
Better performance is the main gain with this pattern. Juggling the various factors that determine performance becomes more important as computers leverage multiple processor cores.
The number of threads used is a parameter that can be tuned to provide the best performance. Additionally, the number of threads can be dynamic based on the number of waiting tasks. For example, a web server can add threads if numerous web page requests come in and can remove threads when those requests taper down. The cost of having a larger thread pool is increased resource usage. The algorithm used to determine when to create or destroy threads will have an impact on the overall performance:

- create too many threads, and resources are wasted and time also wasted; creating any unused threads
- destroy too many threads and more time will be spent later creating them again
- creating threads too slowly might result in poor client performance (long wait times)
- destroying threads too slowly may starve other processes of resources

In [**?**], the authors discuss in depth the dimensionning of a thread pool with respect to the cost (in time) to create threads.

In the following, you are tasked to

1. get familiar with a POSIX-based implementation of the Thread Pool pattern, and use it for a basic multi-threaded web server, serving on basic page;
2. gather metrics on the execution of the threadpool;
3. extend this impementation to support dynamic management of threads: creation, deletion based on some metrics.

2 Implementation of the Threadpool design pattern

In the files `threadpool.[h|c]`, you'll find a POSIX-compliant implementation of the threadpool design pattern. The file `test_threadpool.c` provides a basic example to use this threadpool. It relies on a minimal API to

- create a threadpool, using the `pthread_pool_init()` function;

- assign jobs to the threadpool, with the `pthread_pool_exec()` function. this function takes as parameter a pointer to the pool, a pointer to the function to be executed and its additional arguments. If there is no need for arguments, `NULL` can be used;
- shutdown of the thread pool can be done by passing a specific job to the pool `pthread_pool_destroy()`, and the associated thread pool.

Exercise 1 (Creating a thread pool based Web Server) *In this exercise, we want to experiment with the thread pool API and rewrite the web server you did in previous exercise using this API.*[2]
To use the thread pool, one needs to break function into basic functions representing jobs to be processed. In the case of a web server, one can distinguish:

- *reception of requests from the pool of sockets;*
- *processing of the requests.*

From the previous lab, in which you implemented a web server,

1. *propose two functions that will process incoming requests and dispatch them to the threadpool. Provide the general diagram for request processing, identifyings the queues (O/S level or software levels) and the tasks involved in the processing of one request;*
2. *using the thread pool API, create a multi-threaded Web Server. We will consider the case where the server never stops. The main entrypoint of this server should be in `tp_web_1.c`.*

Exercise 2 (From thread pool to Leader/Followers) *In the previous exercise, we actually use two threads to process a request. In the case of basic static webpage, this is a waste of resources.*
*The Leader/Followers pattern [**?**] builds on the thread pool model in a different way. As it is described by its authors:*

> *"The Leader/Followers architectural pattern provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources"*

The pattern works as follows: structure a pool of threads to share a set of event sources efficiently by taking turns demultiplexing events and synchronously dispatching the events to application services that process them

- *allow one leader thread to wait for an event*
- *other follower threads can queue up waiting for their turn*
- *after detecting an event, then the leader promotes a follower to leader. It then plays the role of a processing thread*

Using the thread pool API and the code from the previous exercise, create a multi-threaded Web Server. We will consider the case where the server never stops. The main entrypoint of this server should be in `lf_web.c`.

Exercise 3 (Benchmarking) *In this exercise, we want to compare relative performances of patterns. We consider servers use a thread pool of size* 10.
Web servers receive requests in random patterns, yet there exist some micro-patterns: a client may spawn multiple requests to fetch HTML pages, CSS (style) and pictures; some client may perform periodic polling to fetch updates; Your objective is to implement a client that will exercise your servers. We are interested in the following scenarios:

[2] **Note:** You do not need to wander in the core of `threadpool.c` for this exercise.

- *A set of periodic clients, each client triggering* 20 *connections every* 300 *ms;*
- *A set of aperiodic clients, triggering* 20 *connections, then suspend for a random amount of time from* 100*ms to* 3*s;*

The performance is usually measured in terms of:

- *Number of requests that can be served per second (depending on the type of request, etc.);*
- *Latency response time in milliseconds for each new connection or request.*

Note: You may use the following function to implement a delay of a few milliseconds:

```
#include<time.h>
void mysleep_ms(int milisec)
{
    struct timespec res;
    res.tv_sec = milisec/1000;
    res.tv_nsec = (milisec*1000000) % 1000000000;
    clock_nanosleep(CLOCK_MONOTONIC, 0, &res, NULL);
}
```