Introduction to Threading

Jon Macey
jmacey@bournemouth.ac.uk

https://github.com/NCCA/Threads
https://nccastaff.bournemouth.ac.uk/jmacey/Lectures/threads/?home=/jmacey/AProg#/Introduction

Introduction

- Concurrent execution of multiple programs has been around since 1960's CTSS
- This was done by interrupting execution of one program and giving the CPU to another.
- This is triggered by
- Regular Hardware interrupts generated by the clock
- Irregular interrupts (e.g. hardware needing attention)
- A call to the OS (e.g. a request to perform I/O)

Time Sharing

- Time sharing allows for efficient use of computational resources but it cannot speed things up for each individual process.
- Time sharing can actually slow down a program as it limits the CPU time the program can use (OS dispatcher dictates this)

What is a thread?

- A thread is an independent stream of instructions scheduled by the operating system to be executed
- It may be easier to think of this as a procedure that run independently from the main program
- This is tied in closely to the Unix process model and in most operating systems this is fairly similar

What is a thread?

- A more precise definition is

  "that it is an execution path, a sequence of instructions, that is managed separately by the operating system scheduler as a unit.
- There can be multiple threads per unit.

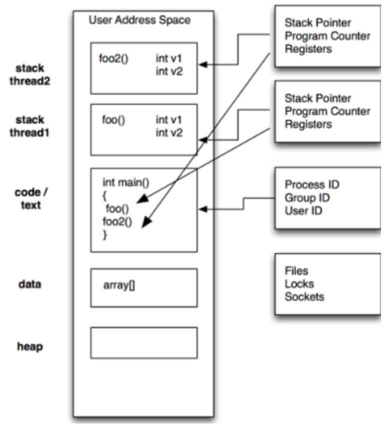
What are Threads good for?

- **Improved performance** breaks up load, places them on multiple cores.
- **Background tasks** Interactive tasks can use threads to allow things to process whilst main task is being executed (GUI / Qt)
- **Asynchronous processing** Sending a request to a server over a network produces latency, threads can be used to wait thus freeing CPU for other tasks.
- **Improving program structure** Games are a good example, we have different tasks that require different times. AI, Render Update Sound etc.

Unix Processes

- A process is created by the operating system and copies a large amount of the environment when it is created
- It will be allocated ID's as well as it's own registers heap etc

- We can create a simple version of this using the C fork()routine



fork

- The fork() function shall create a new process.
- The new process (child process) shall be an exact copy of the calling process (parent process) except
    - It will have it's own id's and environment
    - it's own copy of any file descriptors
    - locks are not inherited
- more information can be found in the man pages

fork.cpp

```
include <cstdlib>
#include <iostream>
#include <unistd.h>

void error()
{
  std::cout<<"Error \n";
}

void child()
{
  std::cout<<"In child\n";
}

void parent()
{
  std::cout<<"In Parent\n";
}

int main()
{

std::cout<<"Started main about to fork\n";
```

```
pid_t pid = fork();

switch (pid)
{
  case -1:
    /* an error occurred, i.e. no child process created */
    error();
  case 0:
    /* a return value of 0 means we're in the child process */
    child();
    break;  // or _exit()
  default:
    /* we're in the parent; pid is the child's process id */
    parent();
}
```

fork

- The typical usage for fork is writing unix daemons, these are programs that start and detach from the parent console (becomes a background process)
- After the fork we create an infinite loop which will do the daemon processing
- It is important that the sleep command is called else we may thrash the OS

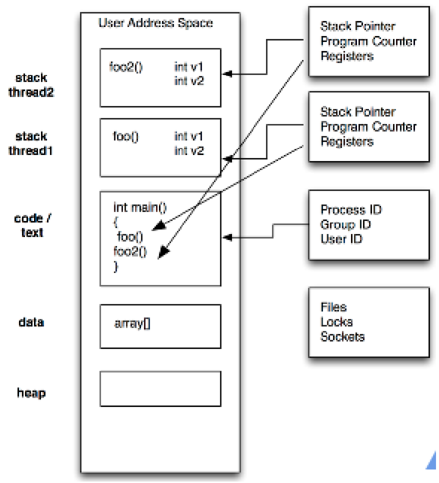
deamon.cpp

```
#include <iostream>
#include <cstdlib>
#include <unistd.h>

int daemonInit()
{
// initialise the daemon using the standard fork
// for a good example see Advanced Programming
// in the UNIX Environment by Stevens
  pid_t pid;
  if ((pid = fork()) < 0)
  {
    return -1;
  }
  else if (pid != 0)
  {
    exit(EXIT_SUCCESS);
  }
            // create a new session
  setsid();

  return 0;
}
```

```
int main()
{
        daemonInit();
        while(1)
        {
                std::cout<<"ping\n";
   sleep(2);
        }
}
```

Threads

- Threads live in the same process resources as the normal unix model
- However they are scheduled by the operating system to run in their own space as they have their own
    - Stack pointer
    - Registers
    - Scheduling properties
    - Set of pending and blocked signals
    - Thread specific data.



Terminology

- Resource : usually memory (variables etc) but anything the program can access
- Critical Section : code that accesses a shared resource
- Semaphore : controls access to a shared resource
- Condition : a construct used to synthesise access to a resource
- Mutex : Mutual Exclusion (semaphores and Conditions are both examples of these)

Mutex

- Short for Mutual Exclusion
- Only one thread can lock (own) a mutex at one time.
- This means critical sections can be locked so only one thread can access at any one time
- This is good for accessing data that is critical to the processing of the thread

Mutex

- Typical process is :
  - create a mutex
  - many threads may try to access
  - only one succeeds and then performs an action
  - once done thread unlocks so others may access
  - repeat (with different threads)
- Two approaches, either thread not accessing blocks until free or can try and not block

Condition Variables

- Used to wait on a condition (for example a thread to finish)
- We use a condition variable in conjunction with a mutex and a predicate (bool) to see if we can access the resource
- We then wait on the condition variable until it is set then access the resource.
- It is a signalling type approach to accessing the data.

pthreads

- Short for "POSIX" threads, the standard unix library for multi-threading under unix style operating systems
- A C library with wrappers to other languages
- Is a standard IEEE POSIX 1003.1c standard (1995).
- Most OS have an implementation of PThreads (can be a wrapper around own code)
- Many higher level API's also use pthreads below a managed layer

pthread functions

- Can be split into three main areas
  - creating, joining and destroying threads
  - mutexes (locking and un-locking access)
  - conditional variables

pthread1.cpp

- need to use the -lpthread flag when compiling on some systems

```
#include <iostream>
#include <cstdlib>
#include <array>
#include <pthread.h>

void *threadFunc(void *arg)
{
  for(int i=0; i<10; ++i)
  {
    std::cout<<"thread func "<<i<<' ';
    std::cout.flush();
  }
  std::cout<<'\n';
  return 0;
```

```
}

int main()
{
  std::array<pthread_t,8> threadID;
  for(auto &t : threadID)
  {
    pthread_create(&t,0,threadFunc,0);
  }

  // now join
  for(auto &t : threadID)
  {
    std::cout<<"*************************\n";
    pthread_join(t,0);
  }
  std::cout<<"############################\n";
}
```

pthread_create

- thread is the id of the thread created
- attr is a list of attributes (0 = default)
- start_routine : the function to run
- arg the arguments to the function

```
#include <pthread.h>

       int pthread_create(pthread_t *thread,
                          const pthread_attr_t *attr,
                          void *(*start_routine) (void *),
                          void *arg);
```

attributes

- To set attributes for the thread the pthread_attr_init function is called
- This allows you to set the way the thread may be joined
- The initial state such as stack size etc
- For most cases the default version can be used

thread function

- The thread function is in the form
- We can pass arguments to the function using a structure and re-cast it within the thread function (examples later)

```
void *foo(void *)
```

pthread_join

- The pthread_join() function suspends execution of the calling thread until the target thread terminates.
- If successful, the pthread_join() function returns zero
- otherwise error values are returned.

std::cout

- std::cout is not thread safe (as we will see in the next example)
- printf behaves better but is also not guaranteed to be safe.
- We may need to use other thread safe logging libs to make it work properly
- I will use printf for now as it makes it clearer what is happening in the examples

pthread2.cpp

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <array>

struct argStruct
{
  int arg1;
  char arg2;
};


void *threadFunc(void *arg)
{
  struct argStruct *args = (argStruct *)arg;
  /*      std::cout<<"thread func \n";
  std::cout<<"Arg 1 "<<args->arg1<<'\n';
  std::cout<<"Arg 2 "<<args->arg2<<'\n';
  */
  printf("thread function %d %c \n",args->arg1,args->arg2);
  return 0;
}


int main()
{
  std::array<pthread_t,4> threadID;
  struct argStruct args;

  for(int i=0; i<4; ++i)
  {
    args.arg1=i;
    args.arg2='a'+i;
    pthread_create(&threadID[i],0,threadFunc,(void *)&args);
  }
  // now join
```

```
  for(auto &t : threadID)
  {
    pthread_join(t,0);
  }
}
```

pthread3.cpp

```
#include <iostream>
#include <cstdlib>
#include <array>
#include <pthread.h>


struct argStruct
{
  int arg1;
  char arg2;
};


void *threadFunc(void *arg)
{
  struct argStruct *args = (argStruct *)arg;
  for(int i=0; i<100000; ++i)
  {
    printf("thread function %d %c \n",args->arg1,args->arg2);
  }
  return 0;
}

int main()
{
  std::array<pthread_t,4> threadID;
  std::array<struct argStruct,4> args;

  for(int i=0; i<4; ++i)
  {
    args[i].arg1=i;
    args[i].arg2='a'+i;
    pthread_create(&threadID[i],0,threadFunc,(void *)&args[i]);
  }
  // now join

  for(auto &t : threadID)
  {
    pthread_join(t,0);
```

```
  }
}
```

pthread4.cpp

```
#include <iostream>
#include <cstdlib>
#include <array>
#include <pthread.h>
struct argStruct
{
  int arg1;
  char arg2;
};
void *threadFunc(void *arg)
{
  struct argStruct *args = (argStruct *)arg;
  printf("thread function %d %c \n",args->arg1,args->arg2);
  int ret=args->arg1+10;
  pthread_exit(reinterpret_cast<void *>(ret));
}

int main()
{
  std::array<pthread_t,4> threadID;
  std::array<struct argStruct,4> args;

  for(int i=0; i<4; ++i)
  {
    args[i].arg1=i;
    args[i].arg2='a'+i;
    pthread_create(&threadID[i],0,threadFunc,(void *)&args[i]);
  }
  // now join

  int ret;
  for(auto &t : threadID)
  {
    printf("join\n");
    pthread_join(t,(void **)&ret);
    printf("return %d\n",ret);
  }
}
```

Race Conditions

- Race conditions or Race Hazards are when two threads are tying to access the same resource at the same time

- The following example has a single shared memory block with three threads trying to access the same data
- This produces a race hazard

racehazard.cpp

```
#include <iostream>
#include <cstdlib>
#include <memory>
#include <array>
#include <unistd.h>
#include <pthread.h>

std::unique_ptr<char []>sharedMem;
constexpr int SIZE=20;

void *starFillerThread(void *arg)
{
  while(1)
  {
  printf("Star Filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='*';
  sleep(2);
          }
}

void *hashFillerThread(void *arg)
{
  while(1)
  {
  printf("hash filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='#';
  sleep(2);
  }
}


void *consumerThread(void *arg)
{
  while(1)
  {
  printf("Consumer\n");
  for(int i=0; i<SIZE; ++i)
    printf("%c",sharedMem[i]);
  printf("\n");
```

```
  sleep(2);
  }
}

int main()
{
  sharedMem.reset(  new char[SIZE]);
  std::array<pthread_t,3> threadID;

  pthread_create(&threadID[0],0,starFillerThread,0);
  pthread_create(&threadID[1],0,hashFillerThread,0);
  pthread_create(&threadID[2],0,consumerThread,0);

  for(auto &t : threadID)
    pthread_join(t,0);
}
```

pthread_mutex_t

- mutex are created using the data type pthread_mutex_t
- To create a mutex we use pthread_mutex_init passing in the mutex and any attributes needed for the creation
- We then use the lock and unlock functions to use the mutex in the code.
- The following example show this (but is still flawed)

mutex1.cpp

```
#include <iostream>
#include <cstdlib>
#include <memory>
#include <array>
#include <unistd.h>
#include <pthread.h>

std::unique_ptr<char []>sharedMem;
constexpr int SIZE=20;


pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

void *starFillerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  printf("Star Filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='*';
  pthread_mutex_unlock(&mutex);
  sleep(2);
```

```
  }
}

void *hashFillerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  printf("hash filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='#';
  pthread_mutex_unlock (&mutex);
  sleep(2);
  }
}
```

```
void *consumerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  printf("Consumer\n");
  for(int i=0; i<SIZE; ++i)
    printf("%c",sharedMem[i]);
  pthread_mutex_unlock (&mutex);
  printf("\n");
  sleep(2);
  }
}

int main()
{
  sharedMem.reset(  new char[SIZE]);
  std::array<pthread_t,3> threadID;

  pthread_mutex_init(&mutex, 0);
  pthread_create(&threadID[0],0,starFillerThread,0);
  pthread_create(&threadID[1],0,hashFillerThread,0);
  pthread_create(&threadID[2],0,consumerThread,0);

  for(auto &t : threadID)
    pthread_join(t,0);
}
```

Problems

- As you can see whilst the data is now being filled in one go the program still has problems

- The threads do not wait for each other and the sequence is out of order
- We need to use conditional waits to make this work correctly.

pthread_cond_t

- This type is used for the conditional wait signals.
- It works in a similar way to the mutex values, first we initialise the variable then call
- pthread_cond_wait with the conditional variable as well as a locked mutex
- The thread will then wait until the condition is met, we must also signal to say we are done.

[conwait.cpp]

```
#include <iostream>
#include <cstdlib>
#include <memory>
#include <array>
#include <pthread.h>
#include <unistd.h>

std::unique_ptr<char []>sharedMem;
constexpr int SIZE=20;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitConsume=PTHREAD_COND_INITIALIZER;


void *starFillerThread(void *arg)
{
 while(1)
 {
 pthread_mutex_lock (&mutex);
 pthread_cond_wait(&waitConsume,&mutex);
 printf("Star Filler\n");
 for(int i=0; i<SIZE; ++i)
   sharedMem[i]='*';
 pthread_mutex_unlock(&mutex);
 sleep(2);
 }
}

void *hashFillerThread(void *arg)
{
 while(1)
 {
 pthread_mutex_lock (&mutex);
 pthread_cond_wait(&waitConsume,&mutex);
 printf("hash filler\n");
 for(int i=0; i<SIZE; ++i)
   sharedMem[i]='#';
 pthread_mutex_unlock (&mutex);
```

```
 sleep(2);
 }
}
```

```
void *consumerThread(void *arg)
{
 while(1)
 {
 pthread_mutex_lock (&mutex);
 printf("Consumer\n");
 for(int i=0; i<SIZE; ++i)
   printf("%c",sharedMem[i]);
 pthread_mutex_unlock (&mutex);
 pthread_cond_signal(&waitConsume);
 printf("\n");
 sleep(2);
 }
}

int main()
{
 sharedMem.reset(  new char[SIZE]);
 std::array<pthread_t,3> threadID;

 pthread_mutex_init(&mutex, 0);
 pthread_cond_init(&waitConsume,0);

 pthread_create(&threadID[0],0,starFillerThread,0);
 pthread_create(&threadID[1],0,hashFillerThread,0);
 pthread_create(&threadID[2],0,consumerThread,0);

 for(auto &t : threadID)
   pthread_join(t,0);

}
```

Condition Variables

- Used to wait on a condition (for example a thread to finish)
- We use a condition variable in conjunction with a mutex and a predicate (bool) to see if we can access the resource
- We then wait on the condition variable until it is set then access the resource.
- It is a signalling type approach to accessing the data.

pthread3.cpp

```
#include <iostream>
#include <cstdlib>
#include <array>
#include <pthread.h>


struct argStruct
{
  int arg1;
  char arg2;
};


void *threadFunc(void *arg)
{
  struct argStruct *args = (argStruct *)arg;
  for(int i=0; i<100000; ++i)
  {
    printf("thread function %d %c \n",args->arg1,args->arg2);
  }
  return 0;
}

int main()
{
  std::array<pthread_t,4> threadID;
  std::array<struct argStruct,4> args;

  for(int i=0; i<4; ++i)
  {
    args[i].arg1=i;
    args[i].arg2='a'+i;
    pthread_create(&threadID[i],0,threadFunc,(void *)&args[i]);
  }
  // now join

  for(auto &t : threadID)
  {
    pthread_join(t,0);
  }
}
```

pthread4.cpp

```
Full Screen#include <iostream>
#include <cstdlib>
#include <array>
```

```
#include <pthread.h>

struct argStruct
{
  int arg1;
  char arg2;
};


void *threadFunc(void *arg)
{
  struct argStruct *args = (argStruct *)arg;
  printf("thread function %d %c \n",args->arg1,args->arg2);
  int ret=args->arg1+10;
  pthread_exit(reinterpret_cast<void *>(ret));
}


int main()
{
  std::array<pthread_t,4> threadID;
  std::array<struct argStruct,4> args;

  for(int i=0; i<4; ++i)
  {
    args[i].arg1=i;
    args[i].arg2='a'+i;
    pthread_create(&threadID[i],0,threadFunc,(void *)&args[i]);
  }
  // now join

  int ret;
  for(auto &t : threadID)
  {
    printf("join\n");
    pthread_join(t,(void **)&ret);
    printf("return %d\n",ret);
  }
}
```

pthread_cond_t

- This type is used for the conditional wait signals.
- It works in a similar way to the mutex values, first we initialise the variable then call
- pthread_cond_wait with the conditional variable as well as a locked mutex
- The thread will then wait until the condition is met, we must also signal to say we are done.

conwait.cpp

```
#include <iostream>
#include <cstdlib>
#include <memory>
#include <array>
#include <pthread.h>
#include <unistd.h>

std::unique_ptr<char []>sharedMem;
constexpr int SIZE=20;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitConsume=PTHREAD_COND_INITIALIZER;


void *starFillerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  pthread_cond_wait(&waitConsume,&mutex);
  printf("Star Filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='*';
  pthread_mutex_unlock(&mutex);
  sleep(2);
  }
}

void *hashFillerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  pthread_cond_wait(&waitConsume,&mutex);
  printf("hash filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='#';
  pthread_mutex_unlock (&mutex);
  sleep(2);
  }
}


void *consumerThread(void *arg)
{
  while(1)
  {
```

```
  pthread_mutex_lock (&mutex);
  printf("Consumer\n");
  for(int i=0; i<SIZE; ++i)
    printf("%c",sharedMem[i]);
  pthread_mutex_unlock (&mutex);
  pthread_cond_signal(&waitConsume);
  printf("\n");
  sleep(2);
  }
}

int main()
{
  sharedMem.reset(  new char[SIZE]);
  std::array<pthread_t,3> threadID;

  pthread_mutex_init(&mutex, 0);
  pthread_cond_init(&waitConsume,0);

  pthread_create(&threadID[0],0,starFillerThread,0);
  pthread_create(&threadID[1],0,hashFillerThread,0);
  pthread_create(&threadID[2],0,consumerThread,0);

  for(auto &t : threadID)
    pthread_join(t,0);

}
```

Problems

- As you can see there are still problems with this, whilst the program locks and fills correctly we still have some synchronisation issues.
- The way to overcome this is using two signals
- They will all be blocked by default then we will unblock one to allow the system to start
- Then filler and consumer will show when they are ready

conwait2.cpp

```
#include <iostream>
#include <cstdlib>
#include <memory>
#include <array>
#include <pthread.h>

std::unique_ptr<char []>sharedMem;
constexpr int SIZE=20;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitFill=PTHREAD_COND_INITIALIZER;
pthread_cond_t waitConsume=PTHREAD_COND_INITIALIZER;
```

```
void *starFillerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  pthread_cond_wait(&waitConsume,&mutex);
  printf("Star Filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='*';
  pthread_mutex_unlock(&mutex);
  pthread_cond_signal(&waitFill);

  }
}

void *hashFillerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  pthread_cond_wait(&waitConsume,&mutex);
  printf("hash filler\n");
  for(int i=0; i<SIZE; ++i)
    sharedMem[i]='#';
  pthread_mutex_unlock (&mutex);
  pthread_cond_signal(&waitFill);
  }
}

void *consumerThread(void *arg)
{
  while(1)
  {
  pthread_mutex_lock (&mutex);
  pthread_cond_signal(&waitConsume);
  pthread_cond_wait(&waitFill,&mutex);

  printf("Consumer\n");
  for(int i=0; i<SIZE; ++i)
    printf("%c",sharedMem[i]);
  pthread_mutex_unlock (&mutex);
  printf("\n");
  }
}
int main()
{
  sharedMem.reset(  new char[SIZE]);
```

```
  std::array<pthread_t,3> threadID;
  pthread_mutex_init(&mutex, 0);
  pthread_cond_init(&waitConsume,0);
  pthread_cond_init(&waitFill,0);

  pthread_create(&threadID[0],0,starFillerThread,0);
  pthread_create(&threadID[1],0,hashFillerThread,0);
  pthread_create(&threadID[2],0,consumerThread,0);

  for(auto &t : threadID)
    pthread_join(t,0);
}
```

c++ 11 threads

- C++ 11 has native, cross platform threading support
- No longer need to use pthreads (but still valid and used)
- Not fully supported in all compilers
- Huge new area but here are some simple examples to get started

thread1.cpp

```
#include <iostream>
#include <thread>
#include <cstdlib>

void hello()
{
  std::cout<<"hello from thread\n";
}

int main()
{
  auto nThreads=std::thread::hardware_concurrency();
  std::cout<<"num threads "<<nThreads<<'\n';
  std::thread t(hello);
  t.join();

  return EXIT_SUCCESS;
}
```

thread1.cpp

- Each thread has an id so we can tell them apart
- We can call the std::this_thread::get_id() to get the id
- As with most C++ 11 we can also use auto for ease as shown in the following example

Thread id’s

- Each thread has an id so we can tell them apart

- We can call the std::this_thread::get_id() to get the id
- As with most C++ 11 we can also use auto for ease as shown in the following example

multithread.cpp

```
#include <thread>
#include <iostream>
#include <vector>
#include <cstdlib>

std::mutex g_print;
void task()
{
   g_print.lock();
   std::cout << "task id=" << std::this_thread::get_id() << '\n';
   g_print.unlock();
}

int main()
{
   std::vector<std::thread> threads(5);

   for(auto &t : threads)
   {
      t=std::thread(task);
   }

   for(auto& thread : threads)
   {
      thread.join();
   }

   return EXIT_SUCCESS;
}
```

Still get race hazards

```
#include <iostream>
#include <random>
#include <thread>
#include <chrono>

int g_counter=0;

void run(int runs)
{
  std::cout<<"Thread " << std::this_thread::get_id()<<" is running\n";
  for(int i=0; i<runs; ++i)
```

```
  {
    std::this_thread::sleep_for(std::chrono::milliseconds(rand()%4));
    g_counter++;
  }
}


int main(int argc, char **argv)
{
  int N = atoi(argv[1]);
  int runs= atoi(argv[2]);
  std::thread *t[N];
  for(int i=0; i<N; ++i)
  {
    t[i]=new std::thread(run,runs);
  }
  for(int i=0; i<N; ++i)
  {
    t[i]->join();
  }
  std::cout<<g_counter<<'\n';

}
```

Lockguard

```
#include <iostream>
#include <random>
#include <thread>
#include <mutex>
#include <chrono>

int g_counter=0;
std::mutex gcountermutex;  // protects counter
void run(int runs)
{
  std::lock_guard<std::mutex> lock(gcountermutex);
  std::cout<<"Thread " << std::this_thread::get_id()<<" is running\n";
  for(int i=0; i<runs; ++i)
  {
    std::this_thread::sleep_for(std::chrono::milliseconds(rand()%4));
    g_counter++;
  }
}


int main(int argc, char **argv)
{
```

```
  int N = atoi(argv[1]);
  int runs= atoi(argv[2]);
  std::thread *t[N];
  for(int i=0; i<N; ++i)
  {
    t[i]=new std::thread(run,runs);
  }
  for(int i=0; i<N; ++i)
  {
    t[i]->join();
  }
  std::cout<<g_counter<<'\n';
}
```

NCCA Logger Lib

- I have created a simple Logger system for both C++ and C++ 11 that allows logging to both file and console at the same time
- This uses std::lock_guard to lock our thread a bit like the previous example
- It also allows for console colour output and writing to log files.

NCCA Logger Lib

- Simple singleton interface using PIMPL idiom for clean API

```
#ifndef LOGGER_H_
#define LOGGER_H_
#include <memory>
#include <cstdarg>
#include <iostream>
#include <ostream>
#include <fstream>

/// @brief logger class for writing to stream and file.
namespace nccalog
{
  enum class Colours{NORMAL,RED,GREEN ,YELLOW,BLUE,MAGENTA,CYAN,WHITE,RESET};
  enum class TimeFormat{TIME,TIMEDATE,TIMEDATEDAY};
  class NCCALogger
  {
    private:
      NCCALogger();
      ~NCCALogger();
      NCCALogger(const NCCALogger &)=delete;
      NCCALogger & operator=(const NCCALogger &)=delete;
    public :
      static NCCALogger &instance();
      void logMessage(const char* fmt, ...);
      void logError(const char* fmt, ...);
```

```
    void logWarning(const char* fmt, ...);
    void enableLogToFile();
    void disableLogToFile();
    void enableLogToConsole();
    void disableLogToConsole();
    void enableLogToFileAndConsole();
    void disableLogToFileAndConsole();
    void setLogFile(const std::string &_fname);
    void setColour(Colours _c);
    void enableLineNumbers();
    void disableLineNumbers();
    void enableTimeStamp();
    void disableTimeStamp();
    void disableColours();
    void enableColours();
    void setLineNumberPad(unsigned int i);
    void setTimeFormat(TimeFormat _f);

   private :
    class Impl;
    std::unique_ptr<Impl> m_impl;


  };

} // end namespace

#endif
```

example

```
#include <thread>
#include <iostream>
#include <vector>
#include <cstdlib>
#include "Logger.h"


void task()
{
  nccalog::NCCALogger::instance().setColour(nccalog::Colours::RED);
  for(int i=0; i<4; ++i)
  nccalog::NCCALogger::instance().logMessage(  "task %x\n" ,std::this_thread::get_id() );
}

int main()
{
  std::vector<std::thread> threads;
```

```
 for(int i = 0; i < 5; ++i)
 {
  nccalog::NCCALogger::instance().setColour(nccalog::Colours::CYAN);
  nccalog::NCCALogger::instance().logWarning("creating thread %d\n",i);
  threads.push_back(std::thread(task));
 }
 int i=0;
 for(auto& thread : threads)
 {
  nccalog::NCCALogger::instance().setColour(nccalog::Colours::YELLOW);
  nccalog::NCCALogger::instance().logWarning("Joining thread %d\n",i++);
  thread.join();
 }

 return EXIT_SUCCESS;
}
```

building

```
#!/bin/bash
clang++ -std=c++11 $1  -g -I../LoggerC++11 -L../LoggerC++11 -lNCCALogger -
I/usr/local/include
```

Threading Functions

- We can use std::bind to bind functions to threads
- Note the use of std::mem_fun to bind the join method.

```
#include <thread>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <algorithm>
#include <chrono>
#include <functional>

#include "Logger.h"

void foo(const std::string  &a, const std::string &b)
{
 while(1)
 {
  nccalog::NCCALogger::instance().setColour(nccalog::Colours::RED);

  nccalog::NCCALogger::instance().
  logMessage("foo(str,str) ID %d value %s %s \n"
```

```
    ,std::this_thread::get_id(),a.c_str(),b.c_str());
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
  }
}
void foo(int a)
{
  while(1)
  {
    nccalog::NCCALogger::instance().setColour(nccalog::Colours::YELLOW);
    nccalog::NCCALogger::instance().logMessage("foo(int) ID %d value %d \n"
    ,std::this_thread::get_id(),a);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
  }
}

void foo(double a)
{
  while(1)
  {
    nccalog::NCCALogger::instance().setColour(nccalog::Colours::BLUE);

    nccalog::NCCALogger::instance().logMessage("foo(double) ID %d value %f\n"
    ,std::this_thread::get_id(),a);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
  }
}


int main()
{
  std::vector<std::thread> threads;
  threads.reserve(4);
  nccalog::NCCALogger::instance().setColour(nccalog::Colours::CYAN);
  nccalog::NCCALogger::instance().logWarning("creating thread String Function\n");


  auto funca = std::bind<void(int)>(foo,1);
  threads.emplace_back(funca);

  auto funcb = std::bind<void(double)>(foo,0.002);
  threads.emplace_back(funcb);

  std::string a="hello";
  std::string b=" c++ 11 threads";
  auto funcs = std::bind<void(const std::string &,const std::string &)>(foo,a,b);
  threads.emplace_back(funcs);
```

```
  using namespace std::placeholders;  // for _1, _2, _3...
  auto funcs2 = std::bind<void(const std::string &,const std::string &)>(foo,_1,_2);
  threads.emplace_back(funcs2,"placeholders","are cool");
  std::for_each(std::begin(threads),std::end(threads),std::mem_fn(&std::thread::join));
  return EXIT_SUCCESS;
}
```

Threading Class Methods

```
#include <thread>
#include <iostream>
#include <vector>
#include <memory>
#include <cstdlib>
#include <string>
#include <functional>

#include "Logger.h"


class Foo
{
  public :
    Foo(int id):m_id(id){}
    void foo(const std::string  &a, const std::string &b)
    {
    while(1)
      nccalog::NCCALogger::instance().logMessage("foo(str,str) %d ID %d value %s %s \n"
      ,m_id,std::this_thread::get_id(),a.c_str(),b.c_str());
    }

  void foo(int a)
  {
    while(1)
      nccalog::NCCALogger::instance().logMessage("foo(int) %d ID %d value %d \n"
      ,m_id,std::this_thread::get_id(),a);
  }

  void foo(double a)
  {
    while(1)
      nccalog::NCCALogger::instance().logMessage("foo(double) %d ID %d value %f\n"
      ,m_id,std::this_thread::get_id(),a);
  }
  private :
    int m_id;

};
```

```
int main()
{
  std::vector<std::thread> threads;
  threads.reserve(6);
  nccalog::NCCALogger::instance().setColour(nccalog::Colours::CYAN);
  nccalog::NCCALogger::instance().logWarning("creating thread String Function\n");
  std::shared_ptr<Foo> pFoo(new Foo(10));
  Foo b(20);


  auto funca = std::bind( static_cast<void (Foo::*)( int )>(&Foo::foo),b,2);
  threads.emplace_back(funca);

  auto funcb = std::bind( static_cast<void (Foo::*)( int )>(&Foo::foo),pFoo.get(),99);
  threads.emplace_back(funcb);

  auto funcc = std::bind( static_cast<void (Foo::*)( double )>(&Foo::foo),b,2.23);
  threads.emplace_back(funcc);

  auto funcd = std::bind( static_cast<void (Foo::*)( double )>(&Foo::foo),pFoo,9.9);
  threads.emplace_back(funcd);

  std::string sa="hello";
  std::string sb=" c++ 11 threads";
  auto funce = std::bind( static_cast<void (Foo::*)( const std::string &,const std::string & )>
  (&Foo::foo),b,sa,sb);
  threads.emplace_back(funce);
  auto funcf = std::bind( static_cast<void (Foo::*)( const std::string &,const std::string & )>
  (&Foo::foo),pFoo.get(),sa,sb);
  threads.emplace_back(funcf);

  std::for_each(std::begin(threads),std::end(threads),std::mem_fn(&std::thread::join));

  return EXIT_SUCCESS;
}
```

Using std::ref

- this example binds a reference to obtain a return value using std::ref

```
#include <thread>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <functional>

#include "Logger.h"
```

```
class Foo
{
public :
  Foo(int id):m_id(id){}

  void mutate(int &io_b)
  {
    io_b+=m_id;
  }

private :
  int m_id;

};

int main()
{
  std::vector<std::thread> threads;
  threads.reserve(2);
  nccalog::NCCALogger::instance().setColour(nccalog::Colours::CYAN);
  nccalog::NCCALogger::instance().logWarning("creating thread String Function\n");
  Foo *pFoo=new Foo(50);
  Foo b(99);

  int value1=10;
  int value2=20;
  auto funca = std::bind( static_cast<void (Foo::*)( int &)>(&Foo::mutate),b,std::ref(value1));
  threads.emplace_back(funca);

  auto funcb = std::bind( static_cast<void (Foo::*)( int
&)>(&Foo::mutate),pFoo,std::ref(value2));
  threads.emplace_back(funcb);


  nccalog::NCCALogger::instance().setColour(nccalog::Colours::YELLOW);
  nccalog::NCCALogger::instance().logWarning("Joining threads \n");
  std::for_each(std::begin(threads),std::end(threads),std::mem_fn(&std::thread::join));
  nccalog::NCCALogger::instance().logError("Value a %d Value b %d \n",value1,value2);

return EXIT_SUCCESS;
}
```

Threading classes

- We can also overload the function call operator and use this as the thread function
- This is sometimes easier to implement if we need a simple call for a class

- This is more difficult if we need arguments and the previous bind method is easier to implement.

example

```
#include <thread>
#include <iostream>
#include <cstdlib>

class Task
{
public :

  Task(){m_id=99;}
  Task(int _t) : m_id(_t){;}

  void operator()() const
  {
    std::cout<<"class operator called "<<m_id<<'\n';
  }

  private :
    int m_id;
};

int main()
{
  Task t;
  unsigned long const nThreads=std::thread::hardware_concurrency();
  std::cout<<"num threads "<<nThreads<<"\n";
  std::thread thread( (Task(2)));
  thread.join();

  return EXIT_SUCCESS;
}
```

std::future std::promise

- The class template std::future provides a mechanism to access the result of asynchronous operations
- An asynchronous operation (created via std::async, std::packaged_task, or std::promise) can provide a std::future object to the creator of that asynchronous operation.

future1.cpp

```
#include <thread>
#include <future>
```

```
#include <iostream>

void func(std::promise<int> && p)
{
  p.set_value(99);
}


int main()
{
  std::promise<int> p;
  auto f = p.get_future();
  std::thread t(&func, std::move(p));
  t.join();
  int I = f.get();
  std::cout<<i<<'\n';
}
```

std::async

- will be deprecated in C++17

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <algorithm>
#include <chrono>
#include <future>
#include <numeric>


const static unsigned int size=100000000;

int sumVect(const std::vector<int>& v)
{
  std::cout<<"sumVect\n";
  int sum=0;
  for(auto i : v)
    sum += i;
  return sum;
}

int sumVectLambda(const std::vector<int>& v)
{
  std::cout<<"sumLambda\n";
  int sum=0;
  for_each(std::begin(v),std::end(v), [&sum](int x) {sum += x; });
  return sum;
}
```

```
int main()
{
  std::vector <int> data(size);
  std::iota(std::begin(data),std::end(data),0);

  auto res1 = std::async(sumVect,data);
  auto res2 = std::async(sumVectLambda,data);
  std::cout<<"start timer\n";
  auto t0 = std::chrono::high_resolution_clock::now();
  auto a=res1.get();
  auto b=res2.get();
  auto t1 = std::chrono::high_resolution_clock::now();
  std::chrono::milliseconds totalMs =
std::chrono::duration_cast<std::chrono::milliseconds>(t1 - t0);
  std::cout<<"using standard vector "<<a<<" "<<b<<" took "<<totalMs.count()<<" Ms \n";
  return EXIT_SUCCESS;
}
```

std::atomic

- Each instantiation and full specialisation of the std::atomic template defines an atomic type.
- Objects of atomic types are the only C++ objects that are free from data races; that is, if one thread writes to an atomic object while another thread reads from it, the behaviour is well-defined.

std::atomic

```
#include <thread>
#include <atomic>
#include <iostream>
#include <vector>

class Counter
{
  public :

    Counter() =default;

    void increment(){ ++m_value;}

    void decrement(){ --m_value;}

    int get()
    {
      return m_value.load();
    }

  public :
    std::atomic<int> m_value={0};
```

```
};

int main()
{
  Counter counter;

  std::vector<std::thread> threads;
  for(int i = 0; i < 10; ++i)
  {
    threads.push_back(std::thread([&counter]()
    {
      for(int i = 0; i < 500; ++i)
      {
      counter.increment();
      }
    }
    ));
  }

  std::for_each(std::begin(threads),std::end(threads),std::mem_fn(&std::thread::join));

  std::cout << counter.get() << '\n';

  return 0;
}
```

thread_local

- The thread_local keyword is only allowed for objects declared at namespace scope, objects declared at block scope, and static data members.
- It indicates that the object has thread storage duration.
- The object is allocated when the thread begins and deallocated when the thread ends.
- Each thread has its own instance of the object.

threadLocal1.cpp

```
#include <thread>
#include <iostream>
#include <cstdlib>

thread_local int i=0;

void foo(int*p)
{
  *p=42;
}

int main()
{
  i=9;
```

```
  std::thread t(foo,&i);
  t.join();
  std::cout<<i<<'\n';
}
```

threadLocal2.cpp

```
#include <thread>
#include <iostream>
#include <cstdlib>
#include <vector>
#include <algorithm>
#include "Logger.h"

class Counter
{
  public :
    void increment() { ++m_count; }
    ~Counter()
    {
      nccalog::NCCALogger::instance().logWarning("Thread %d called %d times
\n",std::this_thread::get_id() ,m_count        );
    }
  private :
    unsigned int m_count = 0;
};

thread_local Counter c;

void threadTask()
{
  c.increment();
}

int main()
{
  std::vector<std::thread> threads;
  for(int i=0; i<10; ++i)
    threads.push_back(std::thread(threadTask));

  for_each(std::begin(threads),std::end(threads),std::mem_fn(&std::thread::join));
}
```

References

- Gerassimos Barlas. 2014. Multicore and GPU Programming: An Integrated Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

- Peter Pacheco. 2011. An Introduction to Parallel Programming (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

1. Introduction to Threading
2. Introduction
3. Time Sharing
4. What is a thread?
5. What is a thread?
6. What are Threads good for?
7. Unix Processes
8. *Slide 7*
9. fork
10. fork.cpp
11. fork
12. deamon.cpp
13. Threads
14. *Slide 13*
15. Terminology
16. Mutex
17. Mutex
18. Condition Variables
19. pthreads
20. pthread functions
21. pthread1.cpp
22. pthread_create
23. attributes
24. thread function
25. pthread_join
26. std::cout
27. pthread2.cpp
28. pthread3.cpp
29. pthread4.cpp
30. Race Conditions
31. racehazard.cpp
32. pthread_mutex_t
33. mutex1.cpp
34. Problems
35. pthread_cond_t
36. conwait.cpp
37. Problems
38. conwait2.cpp
39. Threading in Qt
40. QThread
41. QThread

42. QThread
43. QThread
44. Synchronising Threads
45. QMutex
46. QReadWriteLock
47. QSemaphore
48. QWaitCondition
49. c++ 11 threads
50. thread1.cpp
51. Thread id’s
52. multithread.cpp
53. Still get race hazards
54. Lockguard
55. NCCA Logger Lib
56. NCCA Logger Lib
57. example
58. building
59. Threading Functions
60. Threading Class Methods
61. Using std::ref
62. Threading classes
63. example
64. std::future std::promise
65. future1.cpp
66. std::async
67. std::atomic
68. std::atomic
69. thread_local
70. threadLocal1.cpp
71. threadLocal2.cpp
72. Watch this
73. References