HPCSE - I

«Introduction to multithreading»

Panos Hadjidoukas
Outline

• Processes and Threads
• POSIX Threads API
  • Thread management
  • Synchronization with mutexes
• Deadlock and thread safety
Terminology

- Parallelism in Hardware:
  - multiple cores and memory

- Parallelism in Software:
  - **process**: execution sequence within the OS, a running program
  - **thread**: can execution sequence within a process, all threads of the same process share the application data (memory)

```c
int a[1000];

int main(int argc, char** argv)
{
    for(int i = 0; i < 500; i++) a[i] = 1;
    for(int i = 500; i < 1000; i++) a[i] = 2;

    return 0;
}
```
Processes

• A process consists of the following:
  • Address space: text segment (code), data segment, heap and stack
  • Information maintained by the operating system (process state, priority, resources, statistics)
• Process state: snapshot where the above information has specific values
  • Memory state: state of the address space
  • Processor state: register values
Process Switching

• Before execution, the processor state of a process must be loaded first to the specific processor
• During execution, the processor state of the process changes
• Context switching: a running process stops and another one starts (or resumes)
  – The processor state of the current process is stored
  – The processor state of the next process is loaded
Process Memory

• Each process has its own (private) memory space
  – A process cannot access the memory of another process
  – This provides basic safety in a multi-user environment

• Communication between processes is important
  – When the cooperate to solve a single problem

• Operating systems implement several mechanisms for interprocess communication
  – signals, files, pipes, sockets
  – shared memory
Process Memory Layout

- Command line arguments and environment variables
- Stack
- Heap
- Uninitialized data
- Initialized data
- Text (code)
Memory Organization (C/C++)

- **Text segment**
  - Instruction executed by the processor
  - Can be shared between multiple processes
  - Read-only segment

- **Initialized data segment**
  - Global variables with initial value:
    ```
    double Pi = 3.1415;
    static char message[] = "hello world!";
    ```
Memory Organization (C/C++)

• Uninitialized data segment
  – Global variables without initial value
    ```
    int result;
    double Matrix[512][512];
    ```
  – The operating system initializes these variables to zero before the execution of the program

• Stack
  – Local variables, function parameters

• Heap
  – Dynamic memory management (`malloc`, `new`, …)
Threads

• Thread: an independent stream of instructions that can be scheduled to run as such by the operating system
  • execution sequence within the process
• A process can create multiple threads
  • each thread executes a specific user-defined function
  • main() is the first (primary) thread
• Threads
  • share the memory space of the process they belong to
  • have their own state and some private memory (stack)
  • are cheap to create but difficult to use correctly
  • can run on different processors
Processes and Threads

Traditional Process

- Stack
- Registers
- Heap
- Data
- Code

Single execution flow

Multithreaded Process

- Stack
- Registers
- Stack
- Registers
- Stack
- Registers
- Stack
- Registers
- Heap
- Data
- Code

Multiple execution flows
Spawning and Joining Threads

• During execution of a multithreaded program threads get spawned and joined dynamically
General View

Applications

Operating System (OS)

SCHEDULER

User thread

OS thread

Processor/Core
POSIX Threads (Pthreads)

- Standardized C language threads programming interface
  - [http://pubs.opengroup.org/onlinepubs/9699919799/](http://pubs.opengroup.org/onlinepubs/9699919799/)

- Header file:
  
  ```
  #include <pthread.h>
  ```

- Compilation
  
  ```
  $ gcc -pthread -o hello hello.c
  ```

- Execution
  
  ```
  $ ./hello
  ```
void *func(void *arg)
{
    /* define local data */
    /* function code */
    return (void *)&result;
}

main()
{
    pthread_t tid;
    int exit_value;
    pthread_create (&tid, NULL, func, NULL);
    pthread_join (tid, &exit_value);
}

equivalently:
    pthread_exit(&result);
int pthread_create (pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*routine)(void *), void *arg);

- thread: unique identifier for the new thread returned by the subroutine
- attr: used to set thread attributes. If NULL, the default values are used.
- routine: the C routine that the thread will execute once it is created.
- arg: single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
- if there are no errors, it returns 0
#include <pthread.h>

pthread_t tid;
extern void *func(void *arg);
void *arg;

int res = pthread_create(&tid, NULL, func, arg);
struct data {
  int i;
  float f;
}

void *routine(void *arg) {
  struct data *d = (struct data *) arg;
  int local_i = d->i;
  d->f = 5.0;
  return NULL;
}

int main() {
  pthread_t tid;
  struct data main_data;
  main_data.i = 6;

  pthread_create(&tid, NULL, routine, (void *) &main_data);
  //...
}
Thread Joining

```c
int pthread_join (pthread_t thread,
                  void **status);
```

- `pthread_join()` blocks the calling thread until the specified thread terminates.
- The value returned by the thread function is stored in the memory location specified by `status`.
- If there are no errors, it returns 0.
#include <pthread.h>

pthread_t tid;
int result;

pthread_join(tid, (void *)&result);

pthread_join(tid, NULL);
void *work(void *arg)
{
    pthread_t me = pthread_self();
    printf("Hello world from thread %ld!\n", (long)me);
    return NULL;
}

int main(int argc, char **argv)
{
    long i = 1;
    pthread_t thread;

    printf("main thread %ld!\n", (long)pthread_self());

    pthread_create(&thread, NULL, work, (void *)i);
    pthread_join(thread, NULL);

    printf("Child ended, exiting\n");
    return 0;
}
void *func(void *arg)
{
    sleep(1);
    return NULL;
}

int main(int argc, char * argv[])
{
    pthread_t id[4];

    for (long i = 0; i < 4; i++)  {
        pthread_create(&id[i], NULL, func, NULL);
    }

    for (long i = 0; i < 4; i++)  {
        pthread_join(id[i], NULL);
    }

    return 0;
}
```c
void * func(void * arg)
{
    long sec = (long) arg + 1;
    sleep((long) sec);
    return arg;  /* pthread_exit(arg); */
}

int main(int argc, char * argv[])
{
    pthread_t id[4];
    long result;

    for (long i = 0; i < 4; i++) {
        pthread_create(&id[i], NULL, func, (void *) i);
    }

    for (long i = 0; i < 4; i++) {
        pthread_join(id[i], (void *) &result);
        /* result == i */;
    }

    return 0;
}
```

Fix: 

```c
    long sec = (long) (*arg) + 1;
```
Synchronization

- Consider the following code
  - `next_ticket` is a global variable initialized to 0
  - `ticket` is a local variable, private to each thread

  \[
  \text{ticket} = \text{next\_ticket}++; \quad /* 0 \Rightarrow 1 */
  \]

- In the general case, this is equivalent to the following:

  \[
  \text{ticket} = \text{temp} = \text{next\_ticket}; \quad /* 0 */
  \]
  \[
  ++\text{temp}; \quad /* 1 */
  \]
  \[
  \text{next\_ticket} = \text{temp}; \quad /* 1 */
  \]
Execution with 2 Threads

Thread 0

\[
\begin{align*}
tkt &= \text{tmp} = n\_tkt; \quad (0) \\
++\text{tmp}; &\quad (1) \\
n\_tkt &= \text{tmp}; \quad (1)
\end{align*}
\]

time

Thread 1

\[
\begin{align*}
tkt &= \text{tmp} = n\_tkt; \quad (1) \\
++\text{tmp}; &\quad (2) \\
n\_tkt &= \text{tmp}; \quad (2)
\end{align*}
\]
Another Possible Case

Thread 0

\[ tkt = tmp = n_{tkt}; \quad (0) \]
\[ ++tmp; \quad (1) \]
\[ n_{tkt} = tmp; \quad (1) \]

Thread 1

\[ tkt = tmp = n_{tkt}; \quad (0) \]
\[ ++tmp; \quad (1) \]
\[ n_{tkt} = tmp; \quad (1) \]

What we observe here is a race condition
The update of the shared variable is a critical section
and must be protected
Synchronization - Mutexes

- POSIX Threads include several synchronization mechanisms:
  - Mutexes, Condition variables, Semaphores, Reader-Writer locks, Spinlocks, Barriers

- A Mutex (Mutual Exclusion) is a mechanism that allows multiple threads to synchronize their access to shared resources (e.g. variables):
  - A mutex has two states: locked and unlocked
  - Only one thread can lock the mutex
  - Once a mutex is locked, any other thread that tries to lock that mutex will suspend its execution, until the thread unlocks the mutex
  - At that point, one of the waiting threads acquires the mutex and continues its execution
Mutex Management

• Declaration and static initialization of a mutex:
  
  ```c
  pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
  ```

• Declaration and dynamic initialization of a mutex:
  
  ```c
  pthread_mutex_t mymutex;
  pthread_mutex_init(&mymutex, NULL);
  ```

• Locking (acquiring) the mutex:
  
  ```c
  pthread_mutex_lock(&mymutex);
  ```

• Unlocking (releasing) the mutex after the critical section:
  
  ```c
  pthread_mutex_unlock(&mymutex);
  ```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
double global_sum = 0.0;

void *work(void *arg)
{
    ...
    pthread_mutex_lock(&mutex);
    global_sum += local_sum;
    pthread_mutex_unlock(&mutex);
    ...
}
Check and Lock

int pthread_mutex_trylock(pthread_mutex_t *m);

• Allows a thread to try to lock a mutex
• If the mutex is available then the thread locks the mutex
• If the mutex is locked then the function informs the user by returning a special value (EBUSY)
• This approach allows for implementations of spinlocks

while (pthread_mutex_trylock(&mutex) == EBUSY)
    /*sched_yield()*/;
long num_steps = 100000;
double step;

int main()
{
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (int i=0; i <num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }

    pi = step * sum;

    return 0;
}
#include <pthread.h>
#define NUM_THREADS 2
pthread_t thread[NUM_THREADS];
pthread_mutex_t Mutex;
long num_steps = 100000;
double step;
double global_sum = 0.0;

void *Pi(void *arg) {
    long start;
double x, sum = 0.0;

    start = (long) (*(int *) arg);
    step = 1.0/(double) num_steps;

    for(long i=start; i<num_steps; i+=NUM_THREADS)
    {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pthread_mutex_lock (&Mutex);
    global_sum += sum;
    pthread_mutex_unlock(&Mutex);

    return 0;
}

int main ()
{
    double pi;

    int Arg[NUM_THREADS];

    for(int i=0; i<NUM_THREADS; i++)
        Arg[i] = i;

    pthread_mutex_init(&Mutex, NULL);
    for (int i=0; i<NUM_THREADS; i++)
        pthread_create(&thread[i], NULL, Pi, &Arg[i]);
    for (int i=0; i<NUM_THREADS; i++)
        pthread_join(thread[i], NULL);
    pi = global_sum * step;

    return 0;
}
Deadlocks

- Deadlock can occur when multiple mutexes are not locked in the same order.
- The threads cannot continue their execution:

  Thread 0
  ```c
  pthread_mutex_lock(&mut1);
  pthread_mutex_lock(&mut2);
  pthread_mutex_lock(&mut2);
  pthread_mutex_lock(&mut1);
  ```

  Thread 1
  ```c
  pthread_mutex_lock(&mut2);
  pthread_mutex_lock(&mut1);
  pthread_mutex_lock(&mut1);
  pthread_mutex_lock(&mut1);
  ```

- Deadlock can also occur if a mutex is locked twice (recursively) by the same thread

  ```c
  pthread_mutex_lock(&mut1);
  pthread_mutex_lock(&mut1);
  ```
Thread Safety

- A function is thread-safe if it can be called safely at the same time by multiple threads
- Functions that use static variables are not thread-safe
  - `rand()`, `drand48()`

Solutions
- A mutex inside the function provides an easy but inefficient solution
  - already applied to the previous functions
- Static variables must be replaced by thread private variables that are passed to the function as arguments
Example 1

- rand_r: thread-safe version of rand()
  - randp is assigned a number from 0 and RAND_MAX
  - returns 0 on success

```c
#include <pthread.h>
#include <stdlib.h>

int rand_r(int *ranp){
    static pthread_mutex_t = PTHREAD_MUTEX_INITIALIZER;
    int error;
    if (error = pthread_mutex_lock(&lock))
        return error;
    *ranp = rand();
    return pthread_mutex_unlock(&lock);
}
```
Example 2

• `drand48()` vs `erand48()`
  - "return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0 , 1.0]"

```
srand48(10); // initialization
double xi = drand48();
double yi = drand48();

unsigned short buf[3];// random stream
buf[0] = 0; buf[1] = 0; buf[2] = 10; // initialization
double xi = erand48(buf);
double yi = erand48(buf);
```
References

• Advanced Programming in the Unix Environment, W. Richard Stevens
• Programming with POSIX Threads, David R. Butenhof
  • www.openmp.org
• POSIX threads tutorial at LLNL, Blaise Barney
  • https://computing.llnl.gov/tutorials/pthreads/