GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Pthreads (POSIX Threads)

Dr. Gen Kawamura
ATLAS Experiment, A. Quadt

II. Physikalisches Institut, Georg-August-Universität Göttingen

Practical Course on Parallel Computing Apr., 2018

Bundesministerium
für Bildung
und Forschung

BMBF-Forschungsschwerpunkt
ATLAS-EXPERIMENT
Physik bei höchsten Energien mit dem ATLAS-Experiment am LHC
FSP 103
ATLAS

Table of Contents

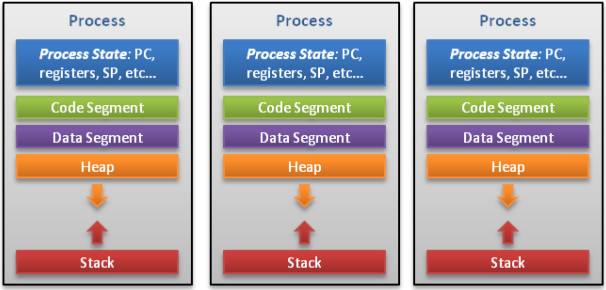
GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Excursion: Processes

- Common operating systems on common hardware handle a lot of *processes* at once
- Each one with it's own set of *virtual memory*
- All processes are strictly separated for security, simplicity and compatibility reasons
- Each *core* can only run one process at a time
- At different intervals the *scheduler* stops the process, changes the memory-Content of the CPU core (aka *the registers*) and starts another one.

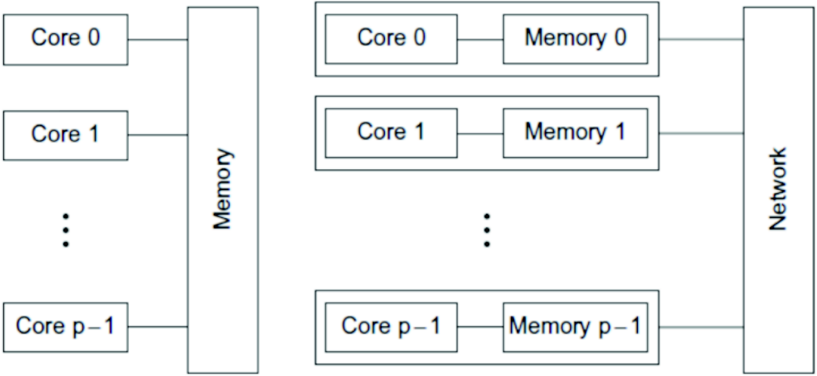
Excursion: Processes #2

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN



Memory structure of a process on a common Operating System/ CPU architecture

Multiple Core / Shared Memory Systems



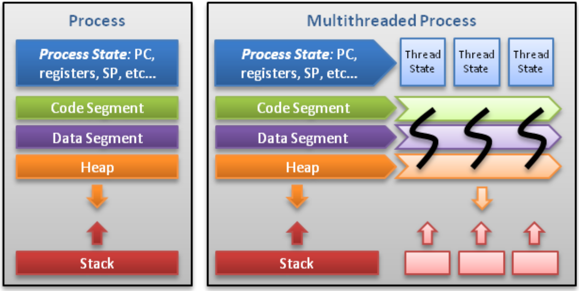
a) shared memory b) distributed memory

Multiple Core / Shared Memory Systems #2

- Run one program per CPU core simultaneously
  - Start one process which sets up the environment
  - and spawns one worker-process per core via fork(3)
- Idea: Communicate via inter-process (shared) memory
- But . . .
  - Memory from different processes strictly separated (on common OS)
  - How to deal with simultaneous access on same memory page?

Threads as a lightweight alternative

- Idea for SMP machines: separate the state and stack but share the heap
- *Lightweight Process* or *Thread*



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

Very short Operating System intermezzo

- How does the OS scheduler handle threads?
  - 1:1 i.e. one thread is one *job* in the scheduler. The case for most recent OS
  - n:1 or n:m i.e. multiple threads are mapped to one *job*. In this case the library or even the threads have to schedule themselves
- One might compare the 2 models above with preemptive vs. cooperative scheduling or with kernel- vs. user-(space)-threads.

How to *create* Threads

- Linux: clone (2) system call; implemented in the NPTL-lib (and glibc)
- Mac OS X: NSThread class (from Cocoa)
- Windows: CreateThread() library call
- POSIX Threads: *pthreads*
  - most Unix': Linux, Mac OS X, Solaris, BSDs...
  - even on Windows
- And many abstractions like boost, QT, glib etc.

We will use POSIX Threads (pthreads)

- Set of (c-) library functions in pthread.h
- Abstracts the underlying OS
- Provides very basic functionality but everything needed to start
- If using frameworks like QT one should use their implementations.
- For C++ one can use the language inherent std::thread class (since C++11)
- C version 11 also has standard threads.h but this is not widely implemented

In the following

- ...we will have a close look at some of pthreads features
- ...we will learn about general concepts of multi-threading

Table of Contents

General Concepts

- Not all features are available on all systems
- You only deal with functions
- If you want to change data objects you have to (!) use special functions

Working with pthreads

```
#include <pthread.h>
pthread_t thread;
// Create attribute object
pthread_attr_t attr;
// Initialize it
pthread_attr_init(&attr);
// Change it
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
// use it
r = pthread_create(&thread, &attr, [...]);
```

Most basic program

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Basis

```c
//compile with   gcc -pthread basic.c
#include <stdio.h>
#include <pthread.h>
void *hello()
{
    printf("Hello World.\n");
    pthread_exit(NULL);
}
main () {
    pthread_t thread;
    pthread_create(&thread, NULL, hello, NULL);
    pthread_exit(0);
}
```

Create, Exit and Cancel Threads

- A thread is terminated if
  - the function ends
  - it calls *pthread_exit*(int return_value)
  - it gets killed with *pthread_cancel*(thread_id)
  - main() ends without waiting (it might wait with *pthread_exit*)
  - the process is terminated/ killed by the OS
- *pthread_exit* does not clean after itself – you have to free() memory, close files etc.

A short word on scheduling

- On Linux you can change the scheduling parameters via setpriority (2), pthread_setaffinity_np (3) or sched_setaffinity (2).
- This might be important for binding on a specific core on NUMA machines.

Passing arguments

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

With arguments

```
void *answer(void *value)
{
    long number = (long) value;
    printf("The␣answer␣is␣%ld.\n", number);
    pthread_exit(NULL);
}
int main () {
    long value = 42;
    pthread_t thread;
    pthread_create(&thread, NULL, answer, (void *) value);
    pthread_exit(0);
}
```

Passing many arguments

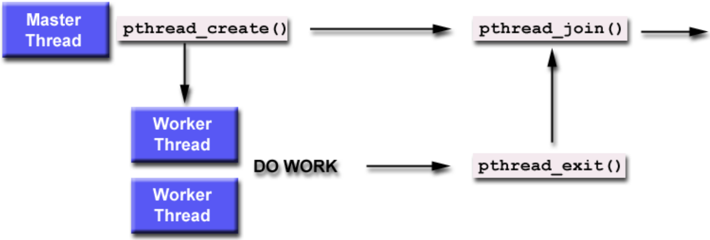
GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

With arguments

Read hello_arg2.c

Wait for another thread

- *pthread_join*(threadid, status) – waits for thread *threadid* and writes the *pthread_exit* return code into *status*
- A thread can only be joined by exactly one other thread
- Example later!



- Global C variables are global over thread boundaries
- Memory in heap (*malloc*) is global over threads boundaries
- Variables in stack are *n*on-global

Shared Data with a pointer

```
int global_variable=42;
int main() {
    int non_global_variable=23;
    int *pointer_to_global_data = malloc(sizeof(int));
    [...]
}
```

Basic Example for shared data

Basic Example

```
int answer=42;
void *hello()
{
    printf("The␣answer␣is␣again␣%d\n",answer);
}
int main () {
    pthread_t thread;
    pthread_create(&thread, NULL, hello, NULL);
    pthread_create(&thread, NULL, hello, NULL);
    pthread_exit(0);
}
```

Locking Data

- pthread offers a native Mutex implementation
- A Mutex shields a part of code. Once you are in this code no one else can go into it.
- Syntacticly a Mutex is a set of functions and a data object

```
mutex_t mymutex;
void thread_1() {
[...]
    lock(mymutex);          // as long as mymutex is locked
        do_something();  // no one else can lock it
    unlock(mymutex);
[...]
}
void thread_2() {
[...]
    lock(mymutex);          // so thread_2 might have to wait
        do_something_else();
    unlock(mymutex);
[...] }
```

pthread Mutex functions

GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

- *pthread_mutex_t* – mutex data structure
- *pthread_mutex_init*(mutex, NULL) – initialize mutex variable
- *pthread_mutex_destroy*(mutex) – destroy it
- *pthread_mutex_lock*(mutex) – lock the mutex; will block and stop the thread until mutex is available
- *pthread_mutex_unlock*(mutex) – can only be called by the mutex-owning thread
- *pthread_mutex_trylock*(mutex) – lock the mutex but does not block; might return with a *impossible-to-lock* error code

Example for the use of a mutex

dotprod_serial.c
dotprod_mutex.c

Some considerations

- A mutex does no magic! The one and only function is to block until the owner unlocks it.
- Think of it as a gentleman's agreement.
- The scheduling is non-deterministic! Any thread might get the lock first! Beware of deadlocks!!

A word on semaphores

- a semaphore is similar to a mutex but it *counts* the number of lock holders
- pthread does not offer a native semaphore type!
- but *semaphore.h* does.
- c.f. sem_init (3)

Signaling and Condition Variables

- With a condition variable we can signal another thread about an event, e.g. that we are finished doing something.

```
physics() {
    calculate_gravity();
    signal_ready(physics);
    [...] }
artifical_intelligence() {
    look_around();
    move_oponents();
    signal_ready(ai):
    [...] }
game(){
    for_each_timestep {
        [...]
        wait_for(physics);
        wait_for(ai);
        paint_graphics();
        }
}
```

Conditions with pthreads

- *pthread_cond_t* – data structure
- *pthread_cond_init*(condition, NULL)
- *pthread_cond_destroy*(condition)
- *pthread_cond_wait*(condition, mutex) – wait for condition *condition*
- *pthread_cond_signal*(condition) – signal to one thread only that the *condition* is fullfilled
- *pthread_cond_broadcast*(condition) – signal to EVERYBODY that the *condition* is fullfilled

Using conditions with pthreads

- You always need an additional mutex to shield the condition!
- *cond_wait*(cond, mutex)
  - should be called after *mutex* is locked by the same thread!
  - *unlocks mutex* and blocks
  - waits for the *cond* to be signaled
  - unblocks and immediately *locks mutex*
  - You have to unlock the *mutex* afterwards!
- The thread might wake up from *cond_wait* although the condition is not fulfilled! $\rightarrow$ You should put it inside a *while*-loop.
- If there is the smallest possibility that more then one thread waits for a condition – use *cond_broadcast*!

Example for Conditions

condvar.c

Barriers

- A barrier is a construct to synchronize threads
- All threads that arrive at a barrier have to wait until everybody else is there
- When initializing we have to specify the maximum number of threads that have to wait

```
mybarrier = barrier(9); // we have 8 planets and one sun

calc_planet_position (my_planet) {
    while (true) {
        calc_force_on_my_planet(all_planet_positions);
        move_my_planet();

        //wait for the other planets to finish
        barrier_wait(mybarrier);
    }
}
```

Barriers in pthread

- `pthread_barrier_t` – data type
- *pthread_barrier_init*(barrier, attr,number) – initialize new barrier which stops *number* of threads
- *pthread_barrier_wait*(barrier) – blocks until *number* threads called this function
- *pthread_barrier_destroy*(barrier)

Table of Contents

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Performance Considerations

- *Lock granularity* – How coarse or fine are your mutexes? Do they lock a whole structure or fields of a structure? The more fine-grained, the more concurrency you can gain.
- *Lock frequency* - Are you locking (too) often? Locking at unnecessary times? Reduce such occurrences to fully exploit concurrency and reduce synchronization overhead.
- *Critical sections* - You should minimize critical sections i.e. section that can only be entered by one thread at a time.
- *Worker thread pool* - If you are using a Boss/Worker thread model, make sure you pre-allocate your threads instead of creating threads on demand.
- *Too many threads?* - At what point are there too many threads? Can it severely impact and degrade performance?

bug6.c
bug6_correct.c

Example for a Deadlock

Thread 1

```
pthread_mutex_lock(&m1);
/* use resource 1 */

pthread_mutex_lock(&m2);

/* use resources1 and 2 */


pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Thread 2

```
pthread_mutex_lock(&m2);
/* use resource 2 */

pthread_mutex_lock(&m1);

/* use resources 1 and 2 */


pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

Table of Contents

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Conclusion

- Threads are a nice way to parallelize problems on a shared memory architecture
- pthreads offer an OS abstraction for threads, mutexe and signal handling (conditions and barriers)
- Avoid race conditions but also avoid deadlocks.

Literature

- pthreads (7) man page and pthread_* (3) man pages
- `http://pages.cs.wisc.edu/~travitch/pthreads_primer.html`
- `http://randu.org/tutorials/threads/`
- `https://computing.llnl.gov/tutorials/pthreads/` (also has a reference)
- Practical Course on Parallel Computing - Sose2015 (The most content/codes were from this material)

Questions