PA193 - Semminar on concurrency

Miroslav Jaroš

6th April 2021

Quick quiz

- What is concurrency?
- Thread x Process difference
- Who provides threads?

Quick quiz

- What is concurrency?
  - Concurrency is the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.
- Thread x Process difference
  - Thread is minimal runnable unit for OS that can be deployed on processor, unlike process it does not own virtual memory, but uses virtual memory of parent process.
  - Every process has at least one thread, which is main for execution.
- Who provides threads?
  - Threads are provided by OS, although many languages has their own implementation due to optimization.

Threads

UNIX

- Pthread library part of POSIX
- `#include <pthread.h>`

WINDOWS

- Defined in WIN32 API
- `#include <windows.h>`

MULTI PLATFORM

- Since C11 and C++11 standards are threads part of standard library
- Qt Framework
- Boost library

LANGUAGES

- **GO**: gorutines
- **ERLANG**: processes
- **ADA**: tasks
- **Java**: `java.lang.Thread`
- **Python**: thread and threading module

Pthread library

- Part of POSIX library
- Provides basic interface for Thread management and mutual exclusion techniques
- All types and functions are prefixed with pthread string
- Needs to be compiled with -pthread argument, to link pthread library into binary
- All types and functions are described in pthread.h header file

Pthread library

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

- Creates new thread, which will start execution in start_routine function
- **thread** in/out attribute, after pthread_create is called, it's set to threads identifier
- **attr** thread attributes, typically passed NULL
- **start_routine** entry point of newly created thread
- **arg** arguments passed to start_routine
- `man 3 pthread_create`

```
int pthread_join(pthread_t thread, void **retval);
```

- Waits for thread to end execution and collect return value
- **thread** thread identifier, set by pthread_create
- **retval** if not set to NULL pthread_join will store start_routine return value in it.
- `man 3 pthread_join`

Helgrind

- Part of valgrind tool for dynamic analysis
- Designed to find bugs in threaded code
- Executed similarly to memcheck
- `valgrind --tool=helgrind ./your_code`
- Your code should be compiled with debugging symbols **"gcc -g ...."**
- http://valgrind.org/docs/manual/hg-manual.html

CRITICAL SECTION

- Point of code where shared resource is manipulated.
- Must be executed exclusively - only one thread at time
- **Even read operations must be exclusive**
  - Context switch can happen in the middle of read operation
  - Then data can be inconsistent
- Goal is to make critical section as small as possible
- Use mutual exclusion to achieve exclusivity

Mutual exclusions

- Posix defines several methods of mutual exclusion
- Mutex - **Mu**tual **Ex**clusion
- Condition variable
- Semaphore

Mutex

- Object which can be in two states, locked and unlocked
- When thread wants to enter critical section, it locks mutex
- When other thread tries to lock mutex, the execution will be stopped and will wait until mutex is unlocked by blocking thread
- When thread is leaving critical section, it unlocks the mutex
- `man 3 pthread_mutex_lock`

```c
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Condition Variable

- Critical section can be entered after condition is met
- Typically in producer-consumer applications, where consumer needs to wait for producer
- Consumer locks mutex, but finds, that it cannot enter critical section
- It calls `pthread_cond_wait` and sleeps, mutex is unlocked
- When producer creates new resource, it calls `pthread_cond_signal`
- All threads waiting for condition are waked and tries to obtain lock, check condition and if its met, they enter critical section with locked mutex
- `man 3 pthread_cond_init`

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
           const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
           pthread_mutex_t *restrict mutex);
```

Semaphores

- Integer value that identifies how many resources are consumable
- Every time the new resource is created, or released the counter is increased
- Every time resource is consumed the counter is decreased.
- Thread that tries to use resource sleeps until resource is allocated
- This allows multiple threads enter critical section when there are enough resources
- Sometimes it needs to be used with mutex, due to possible inconsistencies.
- `man 3 sem_init`

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

Tasks

Work on any UNIX computer

1. Create program which will increase one variable to 10000 from 3 different threads
2. Increase number of threads to 100 and wait for problems to appear
3. Try to find problems with helgrind
4. Add locking to your program
5. Try helgrind to find possible race conditions
6. Modify your code to create deadlock
7. Test it with helgrind
8. Fix your code, so deadlock won't happen.

Tasks II.

1. Create a program, where one thread is putting random numbers into the array and several others are verifying if those numbers are primes.
2. The time needed to produce the numbers should be unpredictable (use random numbers again) so the worker threads will need to wait for new elements to appear
3. Use an appropriate technique for mutual exclusion to avoid busy waiting.
4. Try helgrind to navigate you through the “hell” of the problems.

Assignment

- Simple thread pool
- You are provided with a simple queue and worker interface
- Deadline: 2021-04-13 24:00 CEST (22:00 UTC) - soft deadline with penalisation as usual

Goals:

- Make the queue thread-safe - enqueue and dequeue can happen from different threads
- Do not busy-wait for the elements to appear in the queue. The pop function must be blocking - use an appropriate technique to achieve this.
- Modify worker, so that it will in worker_init spawn several threads, which will wait for jobs to appear in the queue
- You can add any attribute to structures to achieve thread-safety
- You should write your own tests for queue and worker to prove their safety

Assignment

Report:

- As a part of the assignment, you will submit a report
- There you'll describe how did you achieve thread safety
- What bugs you have found during development with helgrind
- If you can't or won't make any bugs during development, then try to make some in tests and describe them in the report as well.
- You should report at least 5 different errors found by helgrind, how those bugs were made, and how you fixed them

Conclusion

- **Don't be afraid of threads**
- **Use threads in your applications**
- You should keep in mind dangers that concurrency can create in your code
- Always try to make critical sections as minimal as possible
- Use mutexes, semaphores and other tools to avoid race conditions, deadlocks and other possible issues
- Check your code with helgrind, it can save you many hours of debugging

Conclusion

- Write your code with concurrency in mind, you might not want to write concurrent library, but someone will eventually try to use it with threads
- Many frameworks and libraries uses threads, even though you don't know it
- Last but not least: **Test your code!**
  - Multi threaded applications are hard to debug, you need to be sure, that particular function/method is doing what it should do!