System-Level Programming

32 Concurrent Threads – Practical Considerations

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

http://sys.cs.fau.de/lehre/ss24
Example: POSIX Threads (pthread)

- Standardized programming interface: **pthread library** (IEEE-POSIX-Standard P1003.4a)
- pthread-interface (basic functions):
  - `pthread_create`: create a new thread
  - `pthread_exit`: thread can terminate itself
  - `pthread_join`: wait for the end of a thread
  ...:
- Functions are combined in the pthread-library
  
  ```
gcc ... -pthread ...
  ```
**pthread Interface**

- **Thread creation**
  ```c
  #include <pthread.h>
  
  int pthread_create(pthread_t *tid, const pthread_attr_t *attr, 
                     void *(*func)(void *), void *param);
  ```

  - **Parameters**
    - **tid**: Pointer to a variable that will store the ID of the thread.
    - **attr**: Pointer to attributes (e.g., size of the stack) for the thread.
      - `NULL` if standard attributes are chosen.
    - **func, param**: The newly created thread will execute the function `func` with parameter `param`.
    - The returned value usually is 0. In case of an error, an error code (similar to `errno`) is returned.
## pthread Interface

** Terminating a thread (on return from inside `func` or): 

```c
#include <pthread.h>

void pthread_exit(void *retval);
```

The thread is terminated and `retval` is returned (see `pthread_join`).

** Waiting for a thread and checking the `pthread_exit`-status: 

```c
#include <pthread.h>

int pthread_join(pthread_t tid, void **retvalp);
```

Waits for the thread with given thread ID `tid` and returns its return value via `retvalp`. The returned value is 0. In case of an error, an error code (similar to `errno`) is returned.
**pthread Example**

Example (matrix-vector multiplication; \( \vec{c} = A \vec{b} \)):

```c
double a[100][100], b[100], c[100];

static void *mult(void *ci) {
    int i = (double *) ci - c;
    double sum = 0.0;
    for (int j = 0; j < 100; j++) {
        sum += a[i][j] * b[j];
    }
    c[i] = sum;
    return NULL;
}

int main(void) {
    pthread_t tid[100];
    for (int i = 0; i < 100; i++) {
        pthread_create(&tid[i], NULL, mult, &c[i]);
    }
    for (int i = 0; i < 100; i++) {
        pthread_join(tid[i], NULL);
    }
}
```
Coordination via mutex (mutual exclusion) variables

Creation of mutex variables

```c
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

**lock operation**

```c
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *m);
```

**unlock operation**

```c
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *m);
```
pthread Example

Mutex example:

```c
volatile int counter = 0;
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);

/* Thread 1 */
pthread_mutex_lock(&m);
    counter++;
pthread_mutex_unlock(&m);
...

/* Thread 2 */
pthread_mutex_lock(&m);
    printf("counter = %d\n", counter);
    counter = 0;
pthread_mutex_unlock(&m);
...```

Synchronization via condition variables.
- It is used for waiting for termination (sleep)
- A termination is signaled (wakeup)
- Creation of a condition variable

```c
pthread_cond_t c;
pthread_cond_init(&c, NULL);
```

- Waiting for a condition

```c
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

- Signaling of a condition

```c
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *c);
int pthread_cond_broadcast(pthread_cond_t *c);
```

`pthread_cond_signal` wakes up one thread, `pthread_cond_broadcast` wakes up all threads waiting for the condition.
Example: counting semaphore

```c
pthread_mutex_t m;
pthread_cond_t c;
pthread_mutex_init(&m, NULL);
pthread_cond_init(&c, NULL);

void P(volatile int *s) {
    pthread_mutex_lock(&m);
    while (*s == 0) {
        pthread_cond_wait(&c, &m);
    }
    *s -= 1;
    pthread_mutex_unlock(&m);
}

void V(volatile int *s) {
    pthread_mutex_lock(&m);
    *s += 1;
    pthread_cond_broadcast(&c);
    pthread_mutex_unlock(&m);
}
```
Thread concept, coordination and synchronization are integrated in Java.

Creation of threads via a thread class; example:

```java
class MyClass implements Runnable {
    public void run() {
        System.out.println("Hello!");
    }
}
...
MyClass o = new MyClass();  // create object
Thread t1 = new Thread(o);  // create thread to run in o
    t1.start();  // start thread
Thread t2 = new Thread(o);  // create second thread
    t2.start();  // start second thread
```
Coordination and synchronization can take place in Java with the help of any object.

- **Coordination via `synchronized` blocks**

```java
synchronized(obj) {
    ...
}
```

Such a block calls a **lock** for the given object `obj` at the beginning and then executes the given instructions. Before leaving the block, the corresponding **unlock** is called.

- **Synchronization via `wait`, `notify` and `notifyAll`**

  - `obj.wait()`: Waits for the signal of a termination on the given object `obj`.
  - `obj.notify()`: Signals the termination on the given object `obj` to a **single** waiting thread.
  - `obj.notifyAll()`: Signals the termination on the given object `obj` to **all** waiting threads.
Example coordination and synchronization:

```java
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }

    public void P() {
        synchronized(this) {
            while (s == 0)
                this.wait();
            s--;
        }
    }

    public void V() {
        synchronized(this) {
            s++;
            this.notifyAll();
        }
    }
}
```

Analogue to the pthread example...
Simplified notation (corresponds to the “monitor” concept):

```java
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }
    public synchronized void P() {
        while (s == 0) {
            wait();
        }
        s--;
    }
    public synchronized void V() {
        s++;
        notifyAll();
    }
}
```