System-Level Programming

31 Concurrent Threads

Peter Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

Multi Processor Systems

In multiprocessor systems, physically parallel execution is possible

but

process creation, termination and, switching are expensive!

For **practical applications**, we therefore should take into account:

- only few processes should be created/terminated
- *never create more processes than there are physical processors*

or

instead of expensive processes use more lightweight, simple **threads**

Threads in a Process

Solution: **multiple** threads in **one** execution environment

- Each **thread** has for its own execution
  - individual program counter
  - individual set of registers
  - individual stack (for local variables)
- **Shared execution environment** provides a set of resources
  - memory mapping
  - permissions
  - open files
  - root and working directory
  - ...

Threads in a Process (2)

- The concept of a process is split up into one **execution environment** and one or more **threads**
- A classical UNIX process is a thread in an execution environment

Threads in a Process (3)

- *Creation/termination of a thread* are less expensive compared to creating/terminating a process (less individual resources required)
- *Switching between threads* inside one process is also cheaper than switching between processes
  - only the registers and the program counter have to be changed (similar to a function call)
  - memory mapping does not have to be changed (cached content remains valid!)

Coordination / Synchronization

- Threads work concurrent/parallel and have shared memory
  ⇒ all problems occurring when dealing with signals and interrupts and accessing shared data also exist
- *Differences* between threads and ISRs/signal handling functions:
  - "main thread" of an application and an ISR/signal handling function are unequal in their behavior
    - ISRs/signal handlers function interrupts the main thread but ISRs/signals are not interrupted by themselves
  - two threads are equal
    - a thread can always be interrupted in favor of an other thread by the scheduler or be run in parallel to another one

⇒ It is insufficient to block signals!

Coordination / Synchronization (2)

- Basic problems
  - mutual exclusion (**coordination**)
    Example:
    A thread wants to read a set of data and prevent other threads from changing the data in this time.
  - mutual waiting (**synchronization**)
    Example:
    A thread waits for an other thread so that they can combine partial results that each thread has computed.

Coordination / Synchronization (3)

- Example of complex problem with coordination and synchronization
  - **Bounded buffer**
  - Threads write data into a buffer, others remove data from it; critical situations:
    - access to the buffer
    - buffer empty/full

Inserting an element:

- wait until there is free space
- wait until no other thread reads/writes from/to the buffer
- write into the buffer
- send signal that there is a new element in the buffer

Removing an element:

- wait until an element is in the buffer
- wait until no other thread reads/writes
- read from the buffer
- send signal that there is free space in the buffer

Mutual Exclusion

- Simple implementation with **mutex** variables

```
volatile int m = 0; /* 0: free; 1: locked */
volatile int counter = 0;
```

```
...         /* Thread 1 */
lock(&m);
counter++;
unlock(&m);
...
```

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

Only the thread that called `lock` is allowed to call `unlock`!

- Realization (only conceptual!)

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

```
void unlock(volatile int *m) {
    *m = 0;
}
```

`lock` (and `unlock`) have to be **executed atomically**!

Counting Semaphores

- A `semaphore` (greek. character carrier) is a data structure with two instructions (refer *Dijkstra*):
  - `P-operation` (*proberen; passeren; wait; down*)

```c
void P(volatile int *s) {
    while (*s <= 0) {
        /* Wait/sleep... */
    }
    *s -= 1;
}
```

  - `V-operation` (*verhogen; vrijgeven; signal; up*)

```c
void V(volatile int *s) {
    *s += 1;
    /* Wakeup... */
}
```

`P` and `V` have to be executed **atomically**!

`P` and `V` do not have to be called from the same thread.

Bounded Buffer (2)

Bounded integer buffer example:

```
#define N 1000
volatile int mutex = 0;
volatile int alloc = 0, free = N;
volatile int head = 0, tail = 0;
volatile int buf[N];
```

Inserting element:

```
void put(int x) {

    P(&free);
    lock(&mutex);
    buf[head] = x;
    head = (head + 1) % N;
    unlock(&mutex);
    V(&alloc);

}
```

Removing element:

```
int get(void) {
    int x;
    P(&alloc);
    lock(&mutex);
    x = buf[tail];
    tail = (tail + 1) % N;
    unlock(&mutex);
    V(&free);
    return x;
}
```

Spin Lock vs. Sleeping Lock

- **Spin lock**
  - active waiting until mutex variable is free ($= 0$)
  - conceptually similar to polling
  - thread stays in the state *running*

  Problem: when there is only one processor available, computation time is wasted until the scheduler schedules a switch

  - only another running thread can free the mutex variable
- **Sleeping Lock**
  - passive waiting
  - thread changes state to *blocked*
  - when `unlock` occurs, the blocked thread changes to the state *ready*

  Problem: for really short critical sections the expenses for blocking/waking up and switching are disproportionately expensive

Implementation Spin Lock

- Main problem: atomicity of mutex request and setting

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

critical section

- Solution: special *machine instructions* that enable to atomically request and modify a cell in the main memory
  - *test-and-set*, *compare-and-swap*, *load-link/store-conditional*, ...

Implementation Sleeping Lock

- Two problems:

1. Conflict with a second **lock** operation:
   Atomicity of mutex request and setting

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

critical section 1 (covers `while (*m == 1) { sleep(); } *m = 1;`)

2. Conflict with second **unlock**: *lost-wakeup* problem

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

critical section 2 (covers `while (*m == 1) { sleep();`)

- Scenarios:

1. switching of processes during a **lock** operation
2. actually parallel running **lock**- and/or **unlock** operations

Implementation Sleeping Lock (2)

- Solution scenario (1):
  prevent process switches
  - process switches are functions of the OS kernel
    - takes place in the context of system calls (e. g., `exit`)
    - or in the context of an interrupt handler (e. g., time-slice expiration interrupt)

    ⇒ **`lock`/`unlock`** are implemented in the OS kernel; OS kernel has preemption avoidance

```c
void lock(volatile int *m)
{
    enter_OS();
    cli();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    sei();
    leave_OS();
}
```

```c
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    *m = 0;
    wakeup_waiting_threads();
    sei();
    leave_OS();
}
```

Implementation Sleeping Lock (3)

- Solution scenario (2):
  Prevent parallel execution on another processor

```
void lock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    spin_unlock();
    sei();
    leave_OS();
}
```

```
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    *m = 0;
    wakeup_waiting_threads();
    spin_unlock();
    sei();
    leave_OS();
}
```

- `P()` and `V()` similar