NTNU | Norwegian University of Science and Technology

Operating Systems

Lecture overview and Q&A Session 4 – 7.2.2022

Michael Engel

Lectures 5 and 6

Threads

- Overhead of process creation
- Lightweight processes – threads and fibers
- Threads in Linux and Windows
- Duff's Device

Concurrency: Mutual Exclusion and Synchronization

- Synchronization problems – race conditions
- Critical sections
- Locks – examples: bakery algorithm, atomic operations
- Semaphores
- Monitors

Threads – Overhead of process creation

- Copying the address space when forking takes a lot of time
    - Fast process creation when immediately calling exec
    - Modern solution: copy on write

- Other approach to implement parallel activities: ***Threads***
- Difference processes ↔ threads
    - Processes have separate address spaces
        - Ensured by copy on write (read-only pages can be shared)
        - Threads of a process share a single address space
    - Threads have separate execution paths
        - Each thread still needs a separate ***stack***

Threads in Linux and Windows

- Windows:
  - Process: provides environment and address space for threads
    - But has no execution context in itself!
  - A Win32 process always contains at least one thread
  - Thread: unit executing code
- Linux:
  - processes without threads are the traditional Unix model
  - Linux implements POSIX threads using the pthreads library
  - all threads and processes are internally managed as tasks
    - scheduler does not differentiate between those

Lightweight processes – threads vs. fibers

- user-level threads, green threads or featherweight processes
- Implemented on application level only
  - operating system doesn't know about them
  - thus, scheduling affects the whole process
- Advantages:
  - Extremely fast context switch – No switch to kernel mode
  - Every application can choose best suited library
- Disadvantages:
  - Blocking a single fiber leads to blocking the whole process (since the OS doesn't know about fibers)
  - No speed advantage from multiprocessor systems

Duff’s Device

- A bad hack that was used in production (in the 1970s…)
- Basic idea (*code fixed to compile on modern Unix*):
    - reduce loop overhead by ***unrolling***
    - abuse the C compiler by jumping into the middle of a loop
    - This worked because the compiler (used to) generate a jump back to the start of the loop when compiling the `while` instruction
- Please don’t write code like this…

```
send(short *to, short *from, int count)
{
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

number of unrolled do-loop iterations

first iteration jumps here for count = 3,11,19,…

jumps to start of loop at end of do-loop

Concurrency – Synchronization problems – race conditions

- Remember – threads share code and data
  - Access to shared data by two or more threads is error-prone
- ***Race condition***
  - multiple processes access shared data concurrently and at least one of the processes manipulates the data
  - the resulting value of the shared data is dependent on the order of access by the processes
  - result is therefore ***not predictable*** and can also be ***incorrect*** in case of overlapping accesses!
- ***Synchronization*** required to ensure safe concurrent access
  - creates an order for the activities of concurrent processes

Critical sections

- In the case of a race condition, N processes compete for the access to shared data
- The code fragments accessing these critical data are called ***critical sections***
- **Problem**
  - We need to ensure that only a single process can be in the critical section at the same time
- Solution: ***Lock variables*** with operations ***wait*** and ***signal***

```
Semaphore lock; /* = 1: use semaphore as lock variable */
void enqueue (struct list *list, struct element *item) {
  item->next  = NULL;
  wait (&lock);          // try to obtain the lock

  *list->tail = item;            // this is the
  list->tail  = &item->next;     // critical section!

  signal (&lock);        // release the lock
}
```

Locks

Different approaches to implement locks:

- **Bakery algorithm**
  - Assign waiting number to process that wants to enter a critical section
  - Admission to critical section in order of waiting numbers
    - Slow, problematic for multicore systems
- **Atomic operations**
  - read/modify/write a memory location in a single cycle
  - cannot be interrupted by other processes or cores
  - requires hardware support – special machine instruction
- **Interrupt control**
  - Disable interrupt before, enable after critical section
  - Large overhead, not useful on multicores

Semaphores

Semaphore:
"*a non-negative integer number*" with two atomic operations

- acquire using "p"/"down"/"**wait**" (different names)
  - if the semaphore has the value 0, the process calling p is blocked
  - otherwise, the semaphore value is decremented and the critical section can be entered
- release using "v"/"up"/"**signal**"
  - if a process waiting for the semaphore (due to a previous call to p), it is unblocked
  - otherwise, the semaphore is incremented by 1
- Semaphores are an **operating system abstraction** to exchange synchronization signals between concurrent processes
  - Complex use patterns, e.g. different reader/writer problems

Monitors

- A **monitor** is an abstract data type with implicit synchronization properties
  - multilateral synchronization at the interface to the monitor
    - mutual exclusion of the execution of all monitor methods
  - unilateral synchronization inside of the monitors using condition variables
    - wait blocks a process until a signal or condition occurs and implicitly releases the monitor again
    - signal indicates that a signal or condition has occurred and unblocks (exactly one or all) processes blocking on this event
- Monitors require support by the programming language

Q&As – Processes

- Why is it important that a parent process needs to check upon (with wait()) a child process that has terminated?

From the Linux `wait(2)` manpage:

- In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.
- A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child.
- As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by `init(1)`, [...]; `init(1)` automatically performs a wait to remove the zombies.

Q&As – Processes

- What is an exit status for a process?

- The exit status is an integer number. 0 exit status means the command was successful without any errors. A non-zero (1-255 values) exit status means command was a failure.

```
int main(int argc, char **argv) {
      …
      exit(42);
}
```

explicit call to `exit` (➛ syscall) never returns

implicit call to `exit` when main returns (startup code in crt0)

```
int main(int argc, char **argv) {
      …
      return 42;
}
```

A program doesn't start at main! …instead, there is startup code in crt0 that is automatically linked and which calls main, which has an `int` return type.

See lecture 8 for some details on crt0 (C runtime zero) and program startup.

Q&As – Processes

- How can I use the exit status?

- If a program is started from the shell, the shell variable $? contains the exit status value of the last executed command:

```
$ ls
foo.c
$ rm farg.c; echo $?
1
$ touch farg.c; rm farg.c; echo $?
0
```

`farg.c` does not exist, trying to delete it fails ➛ `$? = 1`

We create `farg.c` first, then try to delete it ➛ works ➛ `$? = 0`

Q&As – Processes

- How can I use the exit status?
- If a child process is created using fork (the shell also does this, of course), the exit value can be obtained using `wait(2)`:

```
int status;
if (fork() > 0) {
  pid = wait(&status);
  printf("End of proc %d\n", pid);
  if (WIFEXITED(status)) {
    printf("Process exit(%d).\n",
    WEXITSTATUS(status));
  }
}
```

See example code qa4_wait.c on the web page

parent process `waits` for termination of child, passes pointer to status variable

If process exited normally (other reason could be killed by a signal – `WIFSIGNALED`)

Extract the exit code from the status variable using `WEXITSTATUS`

```
#define __WEXITSTATUS(status)      (((status) & 0xff00) >> 8)
#define __WTERMSIG(status)         ((status) & 0x7f)
#define __WIFEXITED(status)        (__WTERMSIG(status) == 0)
```

`/usr/include/x86_64-linux-gnu/bits/waitstatus.h`

Q&As – Threads

- Hva er forskjellen mellom kernel level og user level threads, og når bruker man hva?
- Kernel threads are scheduled by the kernel (D'oh!), i.e. each thread has an entry in a kernel table and can thus be scheduled. Linux implements kernel threads as processes that share an address space. In tools like `htop`, the threads show up with separate process ids:

```
108395 me       20   0  4232  3584  2948 S  0.0  0.3  0:00.01 /bin/bash
108524 me       20   0 19024   636   552 S  0.0  0.1  0:00.00 ./qa4_pthreads
108525 me       20   0 19024   636   552 S  0.0  0.1  0:00.00 ./qa4_pthreads
108523 me       20   0 19024   636   552 S  0.0  0.1  0:00.00 ./qa4_pthreads
```

See example code qa4_pthreads.c linked on the course web page

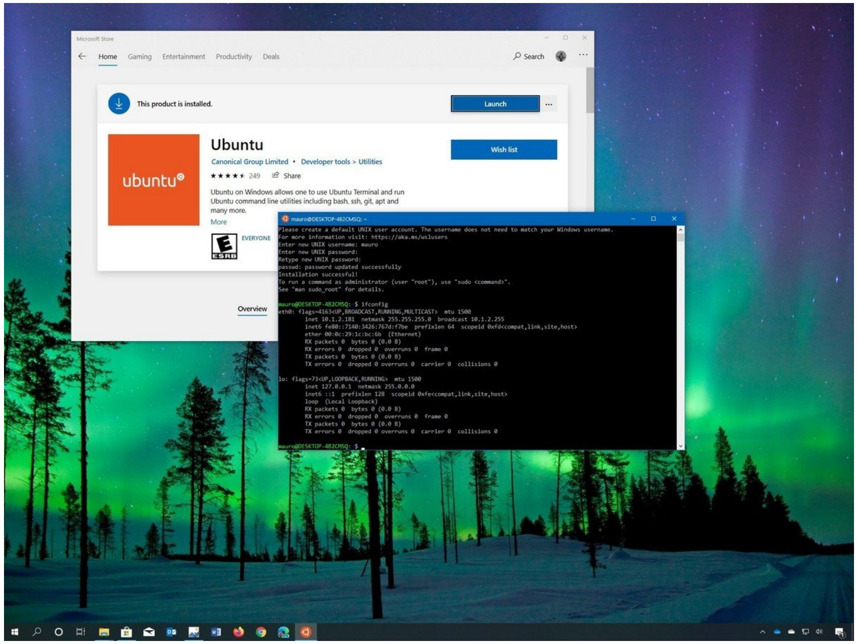
- User mode threads have a lower switching overhead since it's just a "goto" (jump)
- Multi-threaded (User-level) applications cannot take advantage of multiprocessing. Why?
- The OS does not know multiple threads exist inside of a process. A process is (without kernel threads) always assigned to a single core, switching between user threads is an operation like any other process operation

Q&As – Threads

- Let's say I have an app (a process) with two threads, where one has the responsibility to upload real time data to a server and the other for something else. As a gamer, will we need to implement user level threads? Or how is it done?
- There are libraries for user mode as well as kernel mode threads in Linux/Unix. The commonly used library is called `pthreads` (POSIX threads, see the previous example)
- Pthreads could be implemented as user-mode threads – it's just a portable specification how to use OS-specific threads. However, all implementations I have seen use kernel threads
  - *We will supply a pthreads tutorial later this week!*
- The original threading library in Java, GreenThreads, was a user-level threading implementation
  - You could create user-level threads (see the protothreads example from lecture 5), e.g. using the `setjmp/longjmp` calls
  - Linux has additional support, see `setcontext(2)` and `makecontext(3)`

Q&As – Windows?

- How to compile and run code if you have a Windows machine?
- The Windows Subsystem for Linux (WSL) is a virtual machine* that allows to run a Linux system from within Windows 10/11
- Different Linux distributions
  - If you are new to Linux, try Ubuntu
  - You can use your preferred Windows editor
- See details at https://docs.microsoft.com/en-us/windows/wsl/install



* This is true for the current version WSL2, the previous WSL1 used a Linux system call emulation on top of the Windows kernel.

Q&As – Compiling code?

- How to compile and run code on Linux/macOS/WSL?
- Simply on the command line (shell):

The prompt "$" is printed by the shell

```
$ gcc -o prog prog.c
$ ./prog # run it!
```

Call the gcc compiler to compile "prog.c" and link an executable "prog"

- What if you have multiple source code files?

```
$ gcc -o prog file1.c file2.c # or
$ gcc -c file1.c  # create object file file1.o
$ gcc -c file2.c  # create object file file2.o
$ gcc -o prog file1.o file2.o # link executable
```

- On (Ubuntu) Linux (also in WSL) you need the build-essentials package:
  `sudo apt install build-essential`
- On macOS, install Xcode from the App Store or from Terminal.app:
  `xcode-select --install`
- For more complex programs use Makefiles – https://makefiletutorial.com

Overview Theoretical Exercise 2

It's all about parallel execution, semaphores and deadlocks

Why?

- parallel programming is hard and error-prone
- we do not teach it in the first semesters
  - …but almost all computers have multiple cores today
- operating systems implement and require parallel activities
  - e.g. to share data between an interrupt handler and the OS kernel

The forum question

- We are currently discussing setting up a Discourse server
  - open source solution (https://github.com/discourse/discourse)
  - the maths department seems to use it as well as TDT4120…

- Still work in progress – sorry…