2. Übung IBN

Maurice Donner Ise Glade

May 17, 2022

Aufgabe 1

The script can be found in the `.zip` folder and is called `order.sh`. The folder also contains a directory with some test images, which have different timestamps. Use `./order.sh -h` to print the help information.

Aufgabe 2

a. The `files_struct` contains pointers to up to **256** `file` data structures. In todays systems, the maximum number of files a process is able to use is 1048576 (as on Kernel version 5.10.0). Other sources give different numbers which depend on the Kernel version.
This limit can however be changed by simply writing to the file
`/proc/sys/fs/file-max`.

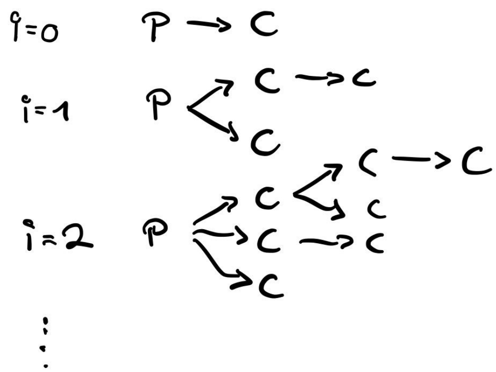
b. The struct `files_struct` is referenced on **line 1073**. Its source code looks like the following:

```
struct files_struct {
        atomic_t    count;              /* structure's usage count */
        spinlock_t  file_lock;          /* lock protecting this structure */
        int         max_fds;            /* maximum number of file objects */
        int         max_fdset;          /* maximum number of file descriptors */
        int         next_fd;            /* next file descriptor number */
        struct file **fd;               /* array of all file objects */
        fd_set      *close_on_exec;     /* file descriptors to close on exec() */
        fd_set      *open_fds;           /* pointer to open file descriptors */
        fd_set      close_on_exec_init; /* initial files to close on exec() */
        fd_set      open_fds_init;      /* initial set of file descriptors */
        struct file *fd_array[NR_OPEN_DEFAULT]; /* default array of file objects */
};
```

The `files_struct` contains all per-process information about open files and file descriptors.

Aufgabe 3

This program creates exactly 2^n processes, where n is the command line argument. For $n = 10$ this is equal to 1024 processes.
The reason for this is that with each new iteration, the variable `i` is inherited by the child process, leading to a tree-like structure that is visualised below



Aufgabe 4

a. The `bash` command `bs` gives a snapshot of the currently running processes. Options such as -aux lets the user get a full list of all processes, even those not associated with a terminal. **b.** There exist five states of processes in Linux:

- Running or Runnable (R)
- Uninterruptible Sleep (D)
  This is the *waiting-* state from the lecture. The process might be waiting for I/O, or other resources.
- Interruptable Sleep (S)
  Similar to `D`, but with the additional option to be woken up by a signal.
- Stopped (T) This is the interrupted state, where a process has recieved the `SIGSTOP` signal, and can be turned back into the runnable state by recieving a `SIGCONT` signal.
- Zombie (Z) This state is usually used by child processes, that have finished their execution, and are waiting for the parent process to terminate them.

Aufgabe 5

a. All we need to do to synchronize the threads is to join each thread after it finished running:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define NUM_THREADS 5

void *TaskCode (void *argument) {
    int tid;
    tid = *((int *) argument);
    printf ("It's me, dude! I am number %d!\n", tid);
    return NULL;
    }

    int main (int argc, char *argv[]) {
    pthread_t threads [NUM_THREADS];

    int thread_args [NUM_THREADS];
    int rc, i;

    for (i=0; i<NUM_THREADS; ++i) {
        /* create all threads */
        thread_args[i] = i;
        printf("In main: creating thread %d\n", i);
        rc = pthread_create(&threads[i],NULL,TaskCode,(void *) &thread_args[i]);
        assert(0 == rc);
        /* Wait for thread i to complete */
        rc = pthread_join(threads[i], NULL);
        assert(0 == rc);
    }

    /* wait for all threads to complete */
    for (i=0; i<NUM_THREADS; ++i) {
    rc = pthread_join (threads[i], NULL);
    assert (0 == rc);
    }

    exit(EXIT_SUCCESS);
}
```

New waiting condition

Note however, that this completely negates the purpose of creating the threads in the first place! The standard output will look like this:

```
In main: creating thread 0
It's me, dude! I am number 0!
In main: creating thread 1
It's me, dude! I am number 1!
In main: creating thread 2
It's me, dude! I am number 2!
In main: creating thread 3
It's me, dude! I am number 3!
In main: creating thread 4
It's me, dude! I am number 4!
```

b. For `NUM_THREADS` $>$ 200000 the execution of the program starts taking longer than 9 seconds. **c.** Running `TaskCode` in a single sequential program 200000 times takes about 0.6s.
One of the reasons for the drastic decrease in runtime is the printing to stdout. Printing, that a thread has been created and printing its thread id, already slows down the program by approximately a factor of 2 (on the used system). However, this still doesn't explain the 5 seconds the program still needs, which might be spent creating and terminating threads. Creating threads could therefore be a relatively slow process, but only if the runtime of the task they perform is in the order of thread creation itself.

The system used for this task is the following:

```
OS: Debian GNU/Linux 11 (bullseye) x86_64
Host: 42915CG ThinkPad X220
Kernel: 5.10.0-9-amd64
Uptime: 18 days, 10 hours, 15 mins
Packages: 2882 (dpkg)
Shell: bash 5.1.4
Resolution: 1920x1080
Terminal: /dev/pts/5
CPU: Intel i5-2520M (4) @ 3.200GHz
GPU: Intel 2nd Generation Core Processor Family
Memory: 2780MiB / 7839MiB
```

Aufgabe 6

- The first level is rather easy, as the global variable `flag` is set true, and both threads can enter the critical region immediately.
- The second level can be solved by first increasing the right counter three times (three-headed dragon), enter the critical region, and then the left counter another two times (five-headed dragon). Since it is the same counter variable that is changed, both threads can enter the critical region.
- The third level can be solved, by expanding the operation of increasing the variable `first`. If both threads store its value at the same time into the variable `temp`, it will never be increased above 1, therefore fulfilling the condition and breaking the program.

Aufgabe 7

a.

- `TSL`: The thread copies the lock value and sets it to something $\neq 0$.
- `CMP` It then checks if the value copied from lock is 0. If no, it may proceed.
- `RET` It then leaves the routine `enter_region` and enters the critical area.

b.

- `TSL`: The thread copies the lock value and sets it to something $\neq 0$.
- `CMP` It then checks if the value copied from lock is 0. If yes, another thread previously requested the lock, and our first thread has to wait. It enters the wait loop.
- `JNE` The thread jumps back to the start of `enter_region` and starts over from step 1: `TSL`.
- This happens another time, before the value copied from the register is 0. The third time, it then may proceed.
- `RET` The thread leaves the routine `enter_region` and enters the critical area.

Below, the processes are quickly sketched:

1. 2.

- enter_region:
  - **TSL** RX, LOCK | kopiere Sperrvariable, sperre mit != 0
  - CMP RX, #0 | war die Sperrvariable 0?
  - JNE enter_region | wenn nicht 0, war gesperrt => Schleife
  - RET | Rücksprung, d.h. k.R. wird nun betreten

no

Schleife

yes yes no