Data Intensive Systems
Software Technology
EEMCS
Delft University of Technology

Operating Systems (CSE2430)
Lab 2: Multithreading

K. Atasu, J. Decouchant, Davis Kažemaks

Deadline: 28 february 2025 - 12:30

This lab contains several exercises about threads, processes and inter-process communication. The objectives of this session are to:

- Learn what a thread is and how you can create one
- Learn how to change the execution priority of a process
- Learn the difference between sleeping and busy waiting
- Get some experience with multithreading and what it entails
- Learn what thread-synchronization entails

Background

Pipes in Linux

Piping in a Linux system connects the standard output of one application to the standard input of another application. When you use `printf()`, you would normally print to the terminal, but when using pipes you can send the data to another application. This application can read that data using `scanf()`. For example, let `numberGen` be a program that outputs some number every second, and let `num2str` be a program that receives numbers as input and prints them as a string:

```
[me@myComputer me]$ ./numberGen
3
5
1
[me@myComputer me]$ ./numberGen | ./num2str
 > three
 > five
 > one
```

Pipes are often used when a program outputs a lot of data. For example, `ps -aux` prints information on all currently running processes, which does not fit on a single screen. You can use `less` to make it possible to scroll through the output with the arrow keys. Another option would be to use `grep` to select only the processes you are interested in: `ps -aux | grep root` would only display lines containing "root". When that is still too much, it is also possible to use `ps -aux | grep root | less`.

If you want to use a text file as the input of your program, you can use the application `cat`, which prints the contents of a file to the standard output. For example:

```
[me@myComputer me]$ cat numbers.txt
7
11
0
[me@myComputer me]$ cat numbers.txt | ./num2str
 > seven
 > eleven
 > zero
```

If you want to write the standard output to a file you can use the > operator. For example, `cat file1.txt > file2.txt` copies the contents of `file1.txt` to `file2.txt`.
Apart from the standard output stream, there is also a standard error stream. For example, `make` prints its errors to the error stream. To write the output from an error stream to a file, use `2>`. You can also use `&>` to write both the standard and the error streams to the same file. For example, `make all &> makeOutput.txt` would save all output from make to the file `makeOutput.txt`.

Using pipes in C

For assignment 2.5, you will have to use `pipe()` to communicate between the parent and its children. The principles that were described previously apply here.

`int pipe(int pipefd[2])` will create a unidirectional channel. `pipefd` will keep the file descriptors used to identify the read and write end of the pipe, `pipefd[0]` for reading from the pipe, and `pipefd[1]` for writing to the pipe. To read and write to a pipe, you can use:
`ssize_t write(int fd, const void buf[.count], size_t count)`
`ssize_t read(int fd, void buf[.count], size_t count)`

`read()` by default is blocking until at least 1 byte can be read, while `write()` is non-blocking and is buffered by the kernel until read. The buffer by default has a **limited capacity** of 65536 bytes, so be sure not to overflow when writing to it. If the pipe is empty and all write channels are closed, `read()` will return EOF. Here is a small example of simple communication between a parent and a child:

```
// create a pipe for child and parent
int pipefd[2];

if (pipe(pipefd) == -1) {
    printf("Unable to create pipe \n");
    exit(1);
}
...

// Parent
close(pipefd[1]); // close the write end of pipe
char buf = 0;
while (read(pipefd[0], &buf, 1) > 0) {
```

```
    printf("%c", buf);
}
printf("\n");

close(pipefd[0]); // close the read end of pipe
...

// Child
close(pipefd[0]); // close the read end of pipe
char* msg = "hello world";
write(pipefd[1], msg, sizeof(char) * 12);
// Alternative: sending one by one
for(int i = 0; i < 12; i++) {
    write(pipefd[1], &msg[i], sizeof(char));
}

close(pipefd[1]); // close the write end of pipe
```

Pointers in C

Here a quick recap of pointers will be given. Pointers store memory locations. They can be used to change the value of a variable indirectly. To do this the reference(&) and dereference(*) operators are used.

```
int x;
int* p = &x; // create a pointer that points to x
             // p now stores the memory location of x
*p = 5;      // x now is equal to 5
int* p2 = x; // p2 now stores the value of x
             // This is something you generally don't want to do
             // The compiler will throw a warning saying that
             // an integer is converted to a pointer without a cast
*p2 = 6;     // BAD: the memory at location 5 is now set to 6
             // this will probably cause a segfault
```

Pointers can point to a variable that goes out of scope. When this happens the behavior of the pointer is undefined: the memory might still be at its last used value, the memory might be used by a different part of your program, or the memory might not be assigned to your program anymore and a segfault will occur.

To avoid this you can dynamically allocate memory using `malloc(<size in bytes>)`. To obtain the right size you can use the `sizeof` keyword. This way, you can allocate memory that remains allocated until it is deallocated using `free`. Unlike Java, there is no garbage collector in C, so you have to deallocate memory yourself. However, when a program is finished executing, all memory that is still allocated will be deallocated automatically. Nevertheless it is good practice to always deallocate all dynamically allocated memory, because it makes your code more reusable and it makes clear what the lifetime of the memory is.

```
int *p, *p2; // creates two pointers (each one should have a separate *)

if(1) {      // always execute block below
    int x = 5;
    p = &x;   // p points to local variable x
    p2 = malloc(sizeof(int)); // allocates memory for one int
    *p2 = 6;
}
printf("%d\n", *p);  // BAD: output not defined, because x went out of scope
printf("%d\n", *p2); // prints 6
free(p2);            // deallocates memory
```

It is also possible to allocate blocks of memory. The syntax for pointers to blocks of memory is the same as the syntax of arrays. Actually an array in C is nothing more than a pointer to a block of memory that will be deallocated when the array goes out of scope. Arrays can be implicitly cast to pointers or be send to a function expecting a pointer.

```
int arr[] = {1, 2, 3};
int *p = arr;
int *p2 = malloc(sizeof(int)*3); // dynamically allocate array of 3 ints

// these three commands will do the same:
arr[1] = 4;
p[1] = 4;
*(p+1) = 4;

free(p2);  // always deallocate your memory!
```

Pointers and dynamic memory allocation are powerful tools, but they can be confusing. When using pointers always keep in mind the scope of your memory. You should only use dynamically allocated memory when the size or the lifetime of your memory block are variable during runtime. When both the size and the lifetime of your memory block are known at compile time, you should use normal variables and arrays instead.

Pthreads

`Pthreads` is a set of interfaces that are used to perform multithreading in C (more details `man pthreads`). You will have to use this interface for the majority of the subassignments. Here is a list of functions that you will encounter:

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

int pthread_join(pthread_t thread, void **retval);
// Assignment 2
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
   const pthread_barrierattr_t *restrict attr, unsigned count);
```

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

For more details on how these functions work, run `man <function_name>`

Assignments

For the assignments all code has to be written in C. Only the code has to be submitted to Weblab (more details on Weblab per assignment).

Some code is provided for assignments to enforce structure or avoid spending time implementing parts not relevant to current study goals. These materials are available on Brightspace and will be provided for each assignment. To ease the building process a `Makefile` is provided. To build an assignment, run `make <assignment>` where `<assignment>` is the name of the source file without '.c'. For example, to compile `2_2.c`, run `make 2_2`.

Please read the assignments carefully and make sure your code exactly implements the given descriptions. We will assess both the functionality and the clarity of your code and comments. **Weblab spec tests do not guarantee that your solution is fully correct.**

You have to research how to solve these assignments on your own: you can look for and read the description of the functions to use and study examples that you will find online. For example, type `man pthread_create` in a terminal to learn more about the function that creates a thread. To learn more details about the manual: `man man`.

2.1

For this assignment, you will be creating a simple program that continuously listens to user input on the main thread (or a new thread), while offloading computations to other threads in parallel. Your tasks are:

1. Have a thread that continuously listens to user input.
2. Once the user inputs an integer between 0-99, a new thread should be spawned to perform a computation.
3. Your program should support up to 10 computation threads running in parallel.
4. For this sub-assignment, the computation is simple: the thread should `sleep` for 1 second (simulating workload), and then increment the number supplied by the user.
5. After the computation is done, the thread should print the output and `fflush` it to `stdout`.
6. The program should continue listening to user input while performing the computation.
7. If the user inputs a negative number, the program should wait for all the threads to finish and then terminate.

Ensure that your program works as expected when running it and manually supplying input, or piping input into the program. For example `echo -e "10\n4\n-1" | ./a.out` should output 11 and 5 after 1 second, and then terminate.

2.2

For this assignment, you will be working with more concrete computations and synchronization mechanisms. Now instead of spawning a thread for each user input, you will have 3 worker threads that use a queue to receive input from the user or other threads.
In this assignment, queue implementation, arguments struct, and worker functions are provided in the materials provided on Brightspace, and **have to** be used in this assignment. You can import them into your program by including `"queue.h"` and `"util.h"`.
The objectives are:

1. Before listening to user input, create 3 threads:
   (a) Thread 1 takes an input and output queue as arguments. It applies `function_1` to the argument polled from the input queue and then pushes it to the output queue.
   (b) Thread 2 takes an input and output queue as arguments. It applies `function_2` to the argument polled from the input queue and then pushes it to the output queue.
   (c) Thread 3 takes 2 input queues as arguments. The input queues for thread 3 are the output queues of threads 1 and 2. Once both working threads signal that they are done with a single item, poll from both of the queues, multiply the outputs, and apply `function_3` to the product. Print and `fflush` the result to `stdout`.
2. Verify the user input is between 1 and 100, and submit the integer to both queues of thread 1 and thread 2.
3. To avoid busy waiting on thread 3 (we assume it has a small workload), use `pthread_barrier_t` to synchronize between all 3 working threads. Thread 3 should only start execution after both threads 1 and 2 have finished 1 item from the queue and pushed their results to the output queues. Threads 1 and 2 should wait for each other before continuing execution.
4. Thread 1 and thread 2 should busy wait for items to appear in the queue, and only exit once user inputs a negative number. You are allowed to avoid busy looping if you can figure out how without compromising other requirements.
5. The program should continue listening to user input while performing the computation.
6. If the user inputs a negative number, the program should wait for all the threads to finish the tasks, and then terminate. **Hint**: The provided functions never produce negative numbers. You can use this information to terminate working threads.

Ensure that your program works as expected when running it and manually supplying input, or piping input into the program.

2.3

In this assignment, we will be observing the scheduling priority (niceness) impact on working processes. Your tasks are:

1. Create a program that asks the user to input a single positive number.
2. `fork()` the program twice to create 3 processes total. Set each niceness value of 0, 10, and 19 on each of these processes, such that the parent process exits after all child processes have finished.
3. All processes perform `function_1` from the previous assignment on the argument. After completing, they should print the output together with their niceness value, and `fflush` the result to `stdout`.
4. To ensure that the program uses a single core while executing, use `sched_setaffinity()` and limit it to the first core of the system. (See `man cpu_set` for more information)

You are **not** allowed to use any other form of synchronization between processes such as `wait` or

`sleep`. Pay attention to the ordering of the outputs of your program.

2.4

A sequential implementation of the quicksort algorithm is given to you in `quicksort.c`. Your task is to modify this `quicksort()` function to make it multi-threaded. You can validate your results by calling the `checkFn()` provided in `quicksort.c`
In this assignment, `quicksort` base implementation, and `thread_count` variable are provided in the materials, and **have to** be used in this assignment. **Do not change the signatures and functionalities of the provided functions**.

1. In `quicksort()`, for both partitions created by the pivot point, make a thread that will sort the subarray in parallel.
2. Your multi-threaded `quicksort()` function should switch to the original serial quicksort implementation when the subarray size becomes smaller than 10,000.
    - Note that even though the input array size may initially be larger than 10,000, the arrays processed by the child threads will become smaller, and serial execution is more efficient than parallel execution for small arrays. Furthermore, switching to serial execution for small arrays helps limit the number of threads spawned.
3. Modify the multi-threaded `quicksort()` function to count the total number of threads created by your multi-threaded quicksort implementation **atomically**. You should use the provided variable `thread_count`.
    - You will need to introduce a new shared variable across threads. To avoid race conditions (which will be discussed more in the next assignment) use a mutex, which can be locked and unlocked respectively using the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. You can initalize it globally with `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

Optional The strategy of spawning a new thread for each recursive call may require an excessively large number of threads, which can exceed the maximum number of threads that can be executed by your system. This approach can also lead to unnecessary performance overheads. An alternative solution is to create a thread pool with a fixed number of threads that use a shared task queue to process the recursive calls to quicksort. This method is readily supported by thread-level parallelism frameworks such as OpenMP, but can also be implemented using pthreads. Feel free to give it a try!

2.5

In this exercise, your task is to develop a parallel sorting implementation using multiple **processes**. You can validate your results by calling the `checkFn()` provided in `quicksort.c`. In this assignment, `quicksort` base implementation is provided in the materials, and have to be used in this assignment. **Do not change the signatures and functionalities of the provided functions.**
Your task is to implement a simplified version of parallel quicksort that uses child processes and pipes. Implementing recursive algorithms using pipes can be a little tricky, therefore we ask you to implement the algorithm as such:

1. For the first level of recursion, instead of creating a thread per partition, create a child process by using a `fork()`.
    - For these first partitions created by the parent, apply the serial quicksort implementation. You do not need to recurse further.
2. The parent should partition the array before calling `fork()`.
3. If the array size has less than 10 entries, apply serial quicksort without splitting.

- Note that pipe buffers have **limited capacity**, so ensure that your writes do not overflow. The child processes should sort the sub-arrays they receive and return them sorted to their parent process utilizing a `pipe()`.

4. The parent process should recombine the full sorted array.

2.5b

This assignment is optional, and constitues a continuation of assignment 2.5. Try implementing a fully recursive version of parallel `quicksort`, similar to the one you implemented with threads:

1. Instead of creating a thread per partition, create a child process by using a `fork()`.
2. The parent should still partition the array before calling `fork()`.
3. Change the base case for switching to serial quicksort implementation when the subarray size becomes 10 times less than the original **or** less than 10 entries.
    - There is a limit set to how many pipes can be created by a user, hence the threshold must be more dynamic. Again, keep in mind that pipe buffers have limited capacity. Child processes should still return the sorted sub-arrays to their parent process using `pipe()`.
4. The parent process should recombine the full sorted array.

If you want to go further

- Develop a multi-process implementation of `quicksort`, where the user specifies the number of processes to use.
- Use other inter-process communication mechanisms (e.g., network sockets).