DM510 Operating Systems
Spring 2026

Exercise Sheet 3

Complete before tutorial on Thursday, February 26th

Learning goals

Be able to

- apply Amdahl's Law and interpret it
- compare processes and threads
- understand role of user and kernel threads
- apply scheduling algorithms seen in lecture
- calculate scheduling performance metrics from lecture on given schedule
- use openmp for parallelisation
- understand and use simple macros in C

Chapter 4

Exercise 1. Name three scenarios where multithreading is useful.

Exercise 2. Using Amdahl's Law, calculate the speedup gain for the following applications

- 40% parallel with (a) eight processing cores and (b) sixteen processing cores
- 67% percent parallel with (a) two processing cores and (b) four processing cores
- 90% percent parallel with (a) four processing cores and (b) eight processing cores

Exercise 3. In the lecture we have seen a speedup gain of roughly $5\times$ for an application that is (estimated) 90% parallel on a system with 12 processing cores. Consider Amdahl's Law and interpret the difference.

Exercise 4. Determine if the following problems exhibit task or data parallelism

- Using a separate thread to generate a thumbnail for each photo in a collection
- Transposing a matrix in parallel
- A network application where one thread reads from the network

Exercise 5. Which of the following components of process state are shared across threads in a multithreaded process?

- Register values
- Heap memory
- Global variables
- Stack memory

Exercise 6. How does creating a thread differ creating a process?

Exercise 7. Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

- The number of kernel threads allocated to the program is less than the number of processing cores.
- The number of kernel threads allocated to the program is equal to the number of processing cores.
- The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads

Chapter 5

Exercise 8. Explain the difference between preemptive and non-preemptive scheduling. Name advantages for both variants.

Exercise 9. Which of the following scheduling algorithms could result in starvation?

- First-come, first-served
- Shortest job first
- Round robin
- Priority

Exercise 10. Suppose that CPU bursts of the following processes arrive for execution at the times (in ms) indicated. Each process will terminate after its CPU burst. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average response time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P1 and P2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling and is of course difficult to implement.

Exercise 11. The following processes each have one CPU burst of the given duration (in ms) that is triggered at the given arrival time. Each process terminates after its CPU burst has finished. They are being scheduled using a preemptive, priority-based, round-robin scheduling algorithm.

Process	Arrival Time	Burst Time	Priority
P1	0	20	5
P2	25	25	30
P3	30	25	30
P4	60	15	10
P5	100	10	40
P6	105	10	35

Each process is assigned a numerical priority, with a lower number indicating a higher relative priority. The scheduler will execute the highest priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

OpenMP

Exercise 12. Write a program that first creates a large array, for example, using `int array = malloc(1024*1024*64*sizeof(int));` and writes the numbers $0, 1, 2, 3, 4, \ldots$ into the array. Then add together all entries and outputs the result. Exploit parallelism via openmp with `#pragma omp parallel for` (and parameter `-fopenmp` when compiling). Check whether the result is the same as the sequential program and discuss whether your code is safe from race conditions. Test the speedup of your code. Since initialization of the array may take a significant amount of time, you might want to consider repeating the summation multiple times to get a better estimate of the speedup. Interpret your results.

Macros

Macros are widely used in C. Consider the example:

```
#define MACRONAME value
```

The preprocessor of the compiler scans the source file for the token `MACRONAME` and every occurrence is replaced by `value`. It is convention to use capital letters for macros. Macros can also take arguments. For example:

```
#define TWICE(x) 2 * x
```

With this, the term `TWICE(5)` would be replaced by `2 * 5`. Using `##` we can even glue a parameter to some other token: Take for example:

```
#define PRINT(msg, type) print ## type(msg)
```

Here, `PRINT("hello world", f)` would be replaced by `printf("hello world")` and `PRINT("hello world", k)` would be replaced by `printk("hello world")`.
`#ifdef MACRONAME ... #endif` and `#ifndef MACRONAME ... #endif` check if there exists (or does not exist) a macro under this name and otherwise the preprocessor will delete everything in between. A macro can also be defined with an empty value.

Exercise 13. In header files you often see the structure

```
#ifndef __HEADERNAME_H__
#define __HEADERNAME_H__
...
#endif
```

Explain the purpose of this.

Exercise 14. Write a macro `SQUARE(x)` that returns the square of `x`. Does you macro work also on `SQAURE(2+2)`? If not, how can you fix this?
Instead of a macro, you could also implement a square function in a library. Name an advantage of either variant.

Exercise 15. When implementing the system call in the programming project, you need to use the macro `SYSCALL_DEFINE1`. This macro is defined in `include/linux/syscalls.h` in the Linux source code. Discuss the readability of the code.

Exercise 16. Write definitions for the following macros:

```
#define FUNCTION_WITH_COUNTER(name, x) ...
#define COUNT(name) ...
```

The macros should automatically add counters to your function definition that keep track of how often it was invoked. For simplicity we only consider functions that take one `int` parameter and return an `int`.
More concretely, the following usage is supposed to work as expected:

```
FUNCTION_WITH_COUNTER(square, y) {
        return y * y;
}

FUNCTION_WITH_COUNTER(increment, z) {
        return z + 1;
}

int main() {
        printf("%d\n", square(3));
        printf("%d\n", square(4));
        printf("%d\n", square(5));

        printf("%d\n", increment(3));
        printf("%d\n", increment(4));

        printf("number function calls %d, %d\n",
                COUNT(square), COUNT(increment));
        return 0;
}
```

To test your macro you can use `gcc -E sourcefile.c`, which outputs the result of the preprocessor. You can write a multi-line macro, you add a backslash at the end of each line.