https://folk.ntnu.no/michaeng/tdt4186_22/
michael.engel@ntnu.no

Example Exam Questions

1 Processes (10 points)

The following pieces of C code are executed on a Unix system. The missing code pieces (includes, definition of main) are not shown for brevity. Assume that all fork calls succeed.

1.1. **Process creation** (5 p.)

Give the **total number of processes** resulting from executing the code, including the process executing this code itself. **Explain your solution**.

```
1  #include <unistd.h>
2
3  int main(void) {
4    pid_t pid1, pid2;
5    pid1 = fork();
6    pid2 = fork();
7    if (pid1>0 && pid2==0) {
8      if (fork()) fork();
9    }
10 }
```

1.2. **Process execution order** (5 p.)

Consider the following example program. List **all possible outputs** this program can produce when executed on a Unix system. The output consists of strings made up of multiple letters.

```
1  #include <unistd.h>
2  #include <sys/wait.h>
3
4  // W(A) means write(1, "A", sizeof "A")
5  #define W(x) write(1, #x, sizeof #x)
6
7  int main() {
8    W(A);
9    int child = fork();
10   W(B);
11   if (child)
12     wait(NULL);
13   W(C);
14 }
```

2 System calls and the shell (10 points)

2.1. **System calls** (5 p.)

The system call `ssize_t read(int fd, void *buf, size_t count);` reads count bytes from the file with file descriptor fd into the buffer pointed to by buf.

To enable safe and secure operation of the OS, the OS has to check properties of the parameters to the read system call. Describe which properties of the parameters should be checked before the OS performs the requested file access.

Assume that `buf != NULL` and `count >= 0`. Page faults are allowed to take place after the checks.

2.2. **Unix shell** (5 p.)

When you enter a command in a Unix shell, this command can either be an internal or an external command.

Can the `cd` (change directory) command be implemented as an external command and work as expected?

If yes, explain its implementation. If no, explain why not.

3 Synchronization (10 points)

3.1. **Synchronization** (5 p.)

The following three functions of a program run in separate threads each and print some prime numbers. All three threads are ready to run at the same time.

Use synchronization using the **semaphores** S1, S2 and S3 and **wait/signal operations** on the semaphores to ensure that the program outputs the prime numbers in increasing order (2, 3, 5, 7, 11, 13).

Insert appropriate wait or signal operations in the code lines indicated with // **SYNC** and give the correct initial values for the semaphores. Give the completed functions and the initial values for the semaphores.

Hint: Note that it might not be required to add a wait or signal operations in all of the places indicated.

```
1  Semaphore S1 = ___;
2  Semaphore S2 = ___;
3  Semaphore S3 = ___;
4
5  f1() {
6    // SYNC
7    printf("3");
8    // SYNC
9    printf("5");
10   // SYNC
11 }
12
13 f2() {
14   // SYNC
15   printf("2");
16   // SYNC
17   printf("13");
18   // SYNC
19 }
20
21 f3() {
22   // SYNC
23   printf("7");
24   // SYNC
25   printf("11");
26   // SYNC
27 }
```

3.2. **Semaphore implementation** (3 p.)

Is it possible to implement concurrency primitive such as mutexes on modern multicore CPUs without the use of special instructions such as `xchg`?

In other words, can such concurrency primitives be written purely in C on current multicore processors?

Explain your answer.

3.3. **Deterministic process execution** (2 p.)

Given an `int` variable `x` in a C program, assume that the instruction `x++;` is executed by two threads of the program in parallel.

If you run the program multiple times, explain why in some cases the variable is sometimes only incremented by one instead of by two.

4 Deadlocks (10 points)

Consider the following code:

```
1  Semaphore L1=1, L2=1, L3=1;
2
3  // Thread 1:
4  wait(L1);
5  wait(L2);
6  // critical section requiring L1 and L2 locked.
7  signal(L2);
8  signal(L1);
9
10 // Thread 2:
11 wait(L3);
12 wait(L1);
13 // critical section requiring L3 and L1 locked.
14 signal(L1);
15 signal(L3);
16
17 // Thread 3:
18 wait(L2);
19 wait(L3);
20 // critical section requiring L2 and L3 locked.
21 signal(L3);
22 signal(L2);
```

Initially, all three mutexes are initialized as “not locked”. Also assume that the threads can execute in any arbitrary interleavings.

4.1. Can there be a problem when executing this multithreaded code? If yes, show an interleaving resulting in the problem. If no, explain why not. (5 p.)

4.2. If there is a problem, propose a fix (Note that each critical section requires two different locks, you cannot change this assumption). (5 p.)

5 Memory allocation (10 points)

5.1. **Buddy algorithm** (5 p.)

A system has a memory of size 32 MB (1 MB = 2^{20} bytes).

Four processes A, B, C and D request memory one after the other (in the order A, B, C, D).

Use the **buddy algorithm** to per form dynamic memory allocation for the processes.

The following allocations take place in the given order. The first allocation for process A is already given in the table below (you can copy and paste the table into the answer test field):

- 1. Process A allocates 6 MB
- 2. Process B allocates 9 MB
- 3. Process C allocates 1 MB
- 4. Process D allocates 6 MB

Hint: If a memory allocation cannot be fulfilled, please indicate this appropriately next to the respective allocation.

Important: Note that each field in the table represents two MB!

Start after the first allocation by process A and give the new allocation of the memory after each of the allocations 2, 3 and 4.

Allocation	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
1. Process A allocates 6 MB	A	A	A	A												
2. Process B allocates 9 MB																
3. Process C allocates 1 MB																
4. Process D allocates 6 MB																

5.2. **LRU (5 p.)**

In a system with page addressing and the page replacement strategy LRU (least recently used), a process performs accesses to memory pages in the given reference sequence:

Reference sequence: 5, 3, 5, 1, 2, 5, 4, 6, 1

The operating system provides three page frames to the process.

Give the number of the virtual memory page that is paged into the respective page frame at each given request in the reference sequence. You can use the fields under “control state” to note the age of the respective page.

Reference sequence		5	3	5	1	2	5	4	6	1
Main memory	frame 1	5								
	frame 2									
	frame 3									
Control states	frame 1	0								
	frame 2									
	frame 3									

6 Virtual memory (10 points)

6.1. **Segmented memory** (4 p.)

Determine the corresponding physical addresses for memory accesses to logical addresses (hexadecimal):

- 0x0000BEEF
- 0x1CEB00DA

using memory segmentation.

In the logical address, the most significant 8 bits of the address indicate the position in the segment table. Indicate in your answer if one of the memory accesses would result in an access violation.

Segment table:

Index	Start address	Length
00	0x0000 00E0	0x21 20FF
01	0xB542 0000	0x01 0000
02	0x0515 0000	0x20 0000
03	0x0006 0000	0x00 FFFF
...		
0x1C	0x0001 0000	0xFF FFFF

6.2. **Paging** (4 p.)

In a system with page addressing (paging), the page frame table is in the state given below. The length of an address is 16 bits. The 12 least significant bits of an address are the offset inside the page (page size = 4096 bytes).

Determine the physical addresses for the logical addresses

- 0x6AB1
- 0xF1B7

Page table:

Page number	Start addresss
0	0xF000
1	0x3000
2	0x8000
3	0x1000
4	0xC000
5	0x2000
6	0x4000
7	0xB000
...	...
15	0x5000
6	0x4000
7	0xB000

6.3. **Logical address structure** (2 p.)

In a system where virtual address `0x722B2104` is mapped to physical address `0x16AB2104`, what is the **largest page size** that could be used for this mapping? Explain your calculation.

7 Scheduling (10 points)

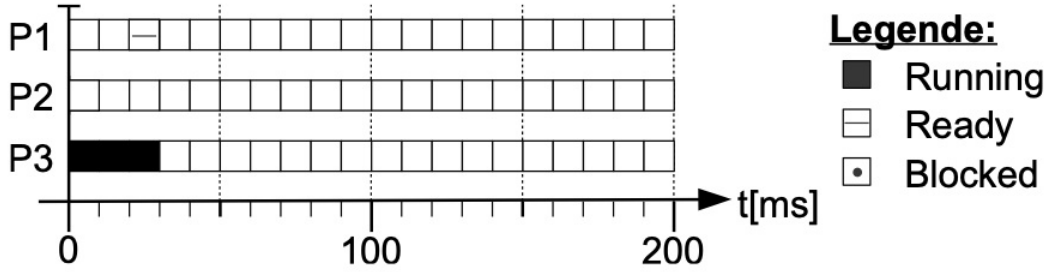
7.1. **FCFS scheduling** (5 p.)

An operating system has three cyclically executing processes P1, P2 and P3 (i.e. the process starts from its beginning after it ran through a CPU- and I/O-burst each).

The processes arrive (are ready to execute) at the time points given in the table below. All times are given in milliseconds (ms).

Process	Arrival time	CPU time	I/O time
P1	20	40	20
P2	30	20	10
P3	0	30	40

Enter the execution order and process state of the processes P1, P2 and P3 in the given Gantt diagram for a first-come, first-served (FCFS) scheduler.



Every process is starting from the beginning after the successful execution of one CPU- and I/O-burst each. After each CPU-burst of a process, a I/O-burst always follows.

Process switch times can be ignored. Each I/O operation can run in parallel with other I/O operations and CPU-bursts.

Mark the states of the three processes (running, ready, blocked) in the diagram according to the legend. It is sufficient to give the states for the first 200 milliseconds.

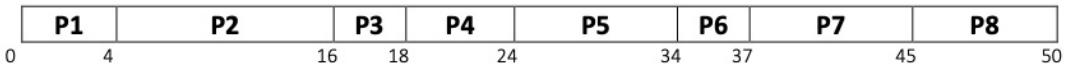
Hint: The first three time units are already filled in.

7.2. **Scheduling algorithms and waiting time** (5 p.)

The following set of processes is given along with their arrival time and the length of their CPU-bursts:

Process	Arrival time (ms)	Burst time (ms)
P1	0	4
P2	2	12
P3	5	2
P4	6	8
P5	8	10
P6	12	3
P7	15	8
P8	22	5

For the FCFS scheduling algorithm, the Gantt diagram giving the sequence and timing of process execution looks as follows:



The **waiting time** of a process is defined as the different between the actual termination of the process and its earliest possible termination time if no other processes would interfere.

In our example, the waiting time for P3 would be (18 ms - 7 ms) = 11 ms – it could be finished at (P3 arrival time + P3 burst time) = (5 ms + 2 ms) = 7 ms, but actually finishes at 18 ms.

The average waiting time is the arithmetic average over all waiting times. For FCFS in our example, this is = (0+2+11+15+32+6+17+5) ms / 8 = 88/8 ms = 11 ms.

Draw the respective Gantt diagrams and calculate the average waiting time for the following scheduling algorithms:

- 1. Non-preemptive SJF (shortest job first)
- 2. Preemptive RR (round robin) with a time slice (quantum) of 6 ms
- 3. Preemptive SRTF (shortest remaining time first)

8 I/O, disk scheduling and file systems (10 points)

8.1. **Unix file I/O** (3 p.)

Consider the following function `read_input()`. Assume that the file which the descriptor fd refers to contains the following sequence of characters:

```
abcd
```

Which string does the buffer buf contain after the `read()` call returns? Assume that no errors occur.

Explain the purpose of the memset() call in the code here.

```
1 void read_input(int fd, char **buf) {
2   *buf = malloc(1024);
3   memset(*buf, 0, 1024);
4   lseek(fd, 3, SEEK_SET);
5   read(fd, *buf, 1023);
6 }
```

8.2. **Disk scheduling** (3 p.)

A disk has 16 tracks. The related I/O scheduler receives a number of read requests for a certain set of tracks.

Initially, the read requests in set L1 are already known to the I/O scheduler. The requests in set L2 arrive after the I/O scheduler has processed one requests; the requests in L3 arrive after the I/O scheduler has processed three additional requests (so overall four).

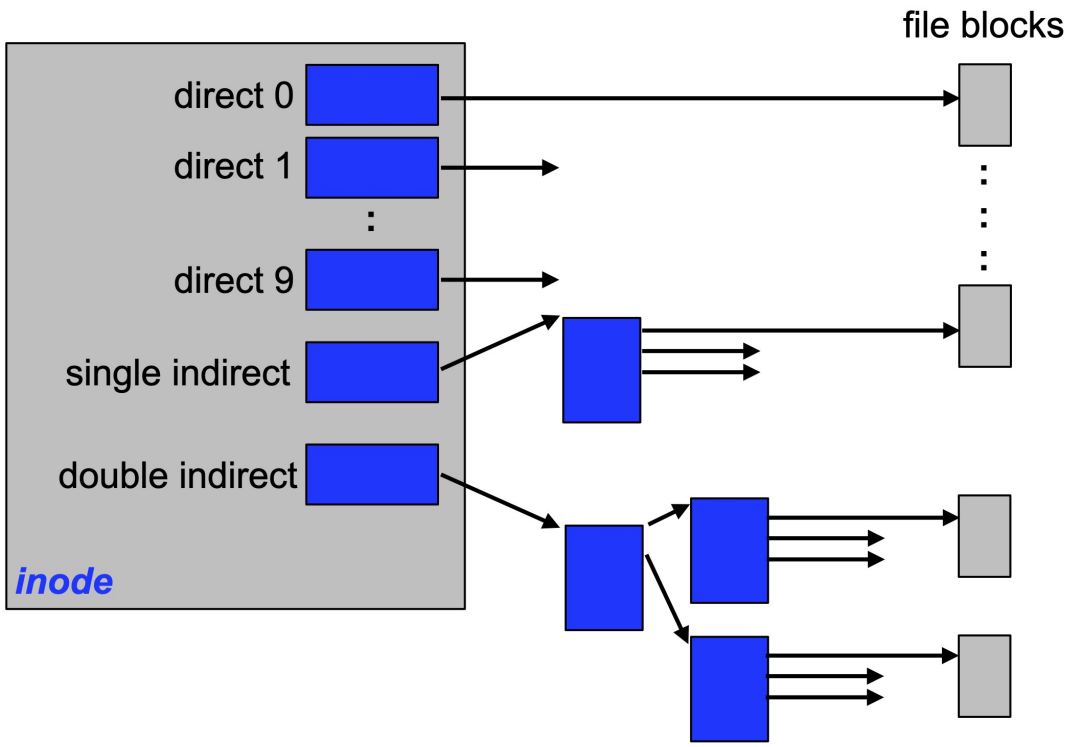
At the start, the disk read/write head is located at track 0.

- L1 = { 4, 7, 11, 3 }
- L2 = { 2, 13, 1 }
- L3 = { 15, 5, 6 }

Give the order of tracks that are read for an I/O scheduler that uses the **First In First Out (FIFO)** strategy.

8.3. **Inode-based file systems** (4 p.)

A file system uses inodes as shown in the picture – direct, single and double indirect inodes (but no triple indirect ones).



Calculate the maximum possible file size in this file system under the given assumptions:

- The block size is 1024 bytes
- A block number requires four (4) bytes of storage

Describe all steps of your calculation and give the value for the maximum possible file size.