TDT4186 Example Exam Spring 2022

Operating Systems / operativsystemer

Solutions

Michael Engel

1. Processes

- These 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 // Code 1
2 pid_t pid1, pid2;
3 pid1 = fork();
4 pid2 = fork();
5 if (pid1>0 && pid2==0) {
6   if (fork())
7     fork();
8 }
```

Process creation (5 points):

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

There are 6 processes in total. The initial process (let's say it has pid 100) generates one child process (pid 101) using fork in line 3, so we have two processes (pids 100 and 101) executing the fork system call in line 4.

After fork returns both in pid 100 and 101, we have two additional child processes (let's assume pids 102 and 103).

Line 5 now checks for the value returned from the first and second fork call. Only if the first fork returned a value > 0 (i.e. we are in the original process with pid 100) **and** the second form returned a value = 0,
the code in lines 6–7 is executed. So we have **six processes altogether**.

1. Processes

- 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.

Process execution order (5 points):

Two possible outputs:
ABBCC and ABCBC

Here you had to know that the order in which parent and child process continue to execute after returning from fork() is not fixed, it can be parent first or child first.

```
#include <unistd.h>
#include <sys/wait.h>

#define W(x) write(1, #x, sizeof #x)

int main() {
  W(A);
  int child = fork();
  W(B);
  if (child)
    wait(NULL);
  W(C);
}
```

2. System calls

System call parameters (5 points)

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.

Three parameters have to be checked to ensure safe and secure operation:

- fd is a valid file descriptor (>=0) and open (optional: for reading on a file)
- buf has to be a valid pointer in the user address space (without checking, the kernel might write to memory outsize of the address space of the process executing read)

Common errors:

- stating that fd > 0 (0 is a valid fd) or count >=0 (size_t is usually an unsigned int type!), leaving out parameters to check, checking size of the buffer (not possible!)

2. System calls

Unix shell (5 points)

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.

Changing directories is implemented using the chdir(2) system call in Unix. So an external "cd" program might look like this (we omit the includes and argument checks here):

```
int main(int argc, char **argv) { chdir(argv[1]); }
```

Each process in Unix has its own current (or working) directory.

After the shell forks, our external "cd" command is started by using a system call from the exec() syscall family. Exec replaces the memory contents of the shell's child process with that of "cd" and starts the program, which successfully ***changes its own directory***.

However, we wanted to change the current directory of the shell. This implies changing the memory contents of the shell itself (i.e., the variable holding the current path). Since the address spaces of the shell and our external "cd" command are separate, the shell path cannot be changed by the external command.

3. Synchronization

Synchronization (5 points)

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

One solution is shown on the right (permutations of the semaphores are also valid). With initial values of all semaphores = 0, only f2 can run, prints 2, signals S1 and then waits for S2. signal(S1) starts f1, which was waiting for S1 and can now print 3 and 5 and then signal S3. signal(S3) now starts f3, which prints 7 and 11 and signals S2. This returns execution to f2, which can then finally print 13.

An alternative valid solution would be to initialize the semaphore not used to start f1 or f3 (here: S2) to 1 and add a wait(S2) to the start of f2 (or, again, permutations of the semaphores).

Partial credit was given for partially correct orders of the output.

Common errors:

- Incorrect orders, incorrect initialization of semaphores, wait(S2) forgotten in f2

```
Semaphore S1 = 0;
Semaphore S2 = 0;
Semaphore S3 = 0;

f1() {
  wait(S1); // SYNC
  printf("3");
  // SYNC
  printf("5");
  signal(S3); // SYNC
}

f2() {
  // SYNC
  printf("2");
  signal(S1);
  wait(S2); // SYNC
  printf("13");
  // SYNC
}

f3() {
  wait(S3); // SYNC
  printf("7");
  // SYNC
  printf("11");
  signal(S2); // SYNC
}
```

3. Synchronization

Semaphore implementation (3 points)

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.

This question was intended to test your understanding of the problem implementing concurrency primitives.

Some of you proposed algorithms such as the bakery algorithm. This is in general correct and received full credit when mentioning that these algorithms are problematic on multicores.

Some of you mentioned cache hierarchies and cache synchronization – fine.

Some of you only mentioned atomicity of instructions. This itself does ***not*** help on a multiprocessor – what has to be atomic is the ***bus transaction*** related to an atomic instruction, since code can run really in parallel on a multicore system.

Some of you mentioned that it is possible in C since the OS provides mutexes – but these also have to be implemented in some language, so that is ignoring the problem (and, in turn, received no credit).

3. Synchronization

Deterministic process execution (2 points)

Given an integer 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.

The C instruction x++ on an integer variable x is commonly translated into three separate parts, e.g.:

```
1 tmp = read_memory(&x);
2 tmp = tmp + 1;
3 write_memory(&x, tmp);
```

In machine language, tmp would refer to a CPU register. Thus, x++ is not an ***atomic instruction***. If this code is executed in parallel in multiple threads, a ***context switch*** can occur somewhere between line 1 and 3, which would result in the copies of tmp in both threads having the same value. Thus, the same value of (x+1) is ***written back twice.*** This effect is called a ***race condition.***

Partial credit for partially correct answers (e.g. incorrect or missing indication when the context is switched).

4. Deadlocks

4.1 (5 points)

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

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

```
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);
```

Yes, there is a **deadlock**. Consider the following interleaving:

```
thread 1:
wait(L1);
        thread 2:
        wait(L3);
                thread 3:
                wait(L2);
```

Now there will be a **circular wait condition**:
thread 1 waiting for L2 (held by thr. 3) → thr. 2 waiting for L1 (held by thr. 1) → thr. 3 waiting for L3 (held by thr. 2).

4. Deadlocks

4.2 (5 points)

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

Obviously, there is a problem :-).

Solution:

Acquire the locks in order the order of L1, L2, L3.

5. Memory allocation

Buddy algorithm (5 points)

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 order A, B, C, D).

Use the buddy algorithm to perform 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.

5. Memory allocation

Common mistakes:
- alignment ignored
- incorrect size
- ignored *subsequent* allocatios
- ignored that size of block = 2 MB

Buddy algorithm (3 points)

1. Process A allocates 6 MB
(already given)

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
A	A	A	A												

2. Process B allocates 9 MB

Aligned block needed with size
of a power of 2 > 9 MB = 16 MB

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
A	A	A	A					X	X	X	X	X	X	X	X

3. Process C allocates 1 MB

Aligned block needed with size
of a power of 2 > 1MB = 2 MB
(blocks 10, 12, 14 also correct)

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
A	A	A	A	X				X	X	X	X	X	X	X	X

4. Process D allocates 6 MB

Allocation ***not possible***, 6 MB are
free (10-14), but not correctly aligned

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
A	A	A	A	X				X	X	X	X	X	X	X	X

5. Memory allocation

LRU (5 points)

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 (you can copy and paste the table into the answer test field).

Control states were not required.
Partial credit for partially correct solutions.

Common errors:

- Mistakes when determining the age or using LRU

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

6. Virtual memory

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

Segmented memory (4 points)

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.

Most significant 8 bits of the address indicate the position in the segment table:

```
0x00_00BEEF: segment index 0x00 = start 0x0000_00E0, length 0x21_20FF
   segment offset = 0x00_BEEF < length 0x21_20FF => valid
   physical address = start address + offset = 0x0000_00E0 + 0x00_BEEF = 0x0000_BFCF

0x1C_EB00DA: segment index 0x1C = start 0x0001_0000, length 0xFF_FFFF
   segment offset = 0xEB_00DA < length 0xFF_FFFF => valid
   physical address = start address + offset = 0x0001_0000 + 0xEB_00DA = 0x00EC_00DA
```

Partial credit for partially correct translations, no credit for result without explanation.

Common errors:

- Hexadecimal addition is hard (and some converted the numbers to decimal…)

6. Virtual memory

Paging (4 points)

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 and 0xF1B7.

0x6AB1: page number = four most significant bits = 0x06, offset (12 LSBs) = 0xAB1
page 0x6 has physical start address 0x4000
0x4000 + offset 0xAB1 = 0x4AB1

0xF1B7: page number = four most significant bits = 0x0F, offset (12 LSBs) = 0x1B7
page 0xF has physical start address 0x5000
0x5000 + offset 0x1B7 = 0x51B7

Partial credit for partially correct translations with accompanying explanation.

Common errors:

- incorrect lookup in the page table, adding the start address to the whole 16 bit virtual addr.

Page table:

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

6. Virtual memory

Logical address structure (2 points)

In a system where virtual address 0x72**2**B2104 is mapped to physical address 0x16**A**B2104, what is the largest page size that could be used for this mapping? Explain your calculation.

The largest page size that could be used is the set of low order bits that are identical between the two addresses. 0xB2104 is the common suffix of both addresses, so the largest page size is ***at least*** 2^{20} (because each hex digit can be represented by exactly 4 bits).

We then look to the first non-identical digit.

Digit **2** (from the virtual address) has a binary representation of 0010 while digit **A** (from the physical address) is 1010. These have **3 bits in common**, so the largest page size that could be used is 2^{23} for this pair of virtual and physical addresses.

Partial credit for correct explanation, but incorrect result.
No credit for result 2^{20} (or any other result) or result without explanation.

Common errors:

- This was a(n intentionally) hard question. The common additional 3 bits were often not considered.

7. Scheduling

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

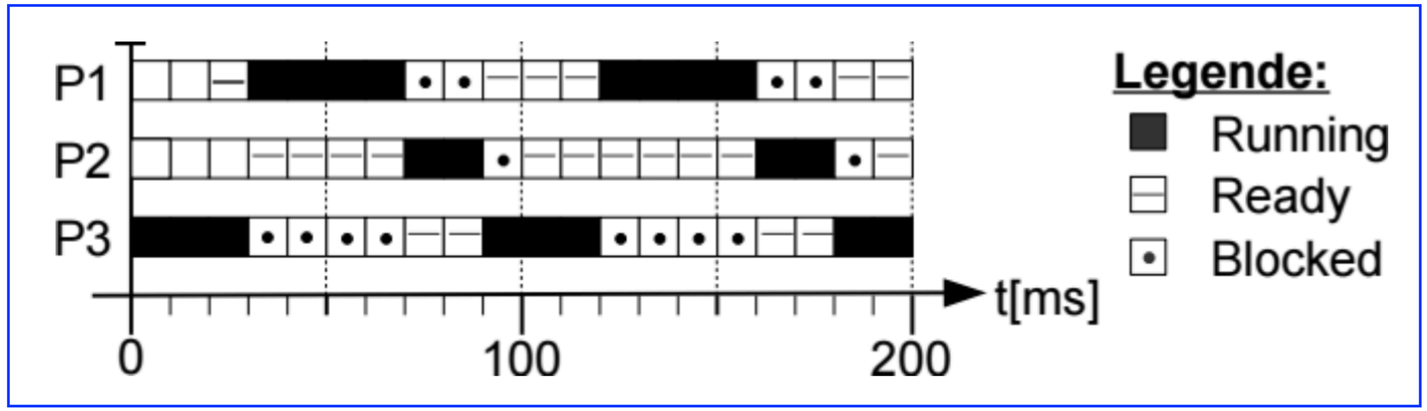
FCFS Scheduling (5 points)

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).

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.

P3 is the only process ready at t=0, so it runs until the start of its I/O burst at t=30. P1 arrived at t=20 and is the only process ready at t=30, so it executes for 40 time units and then starts I/O. At t=70, P2 and P3 are ready, P2 arrived earlier and is scheduled (etc.)



Common errors:

- Processes not considered periodic
- Wrong counts
- Assuming preemption

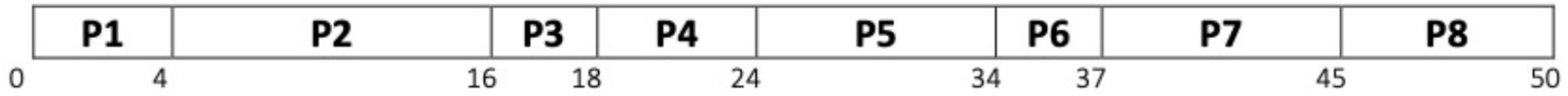
7. Scheduling

Scheduling algorithms and waiting time (5 points)

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	6
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:



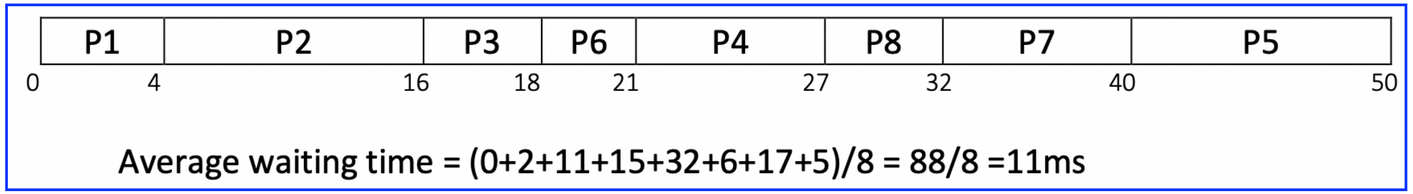
Draw the respective Gantt diagrams and calculate the average waiting time for the following scheduling algorithms:

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)

7. Scheduling

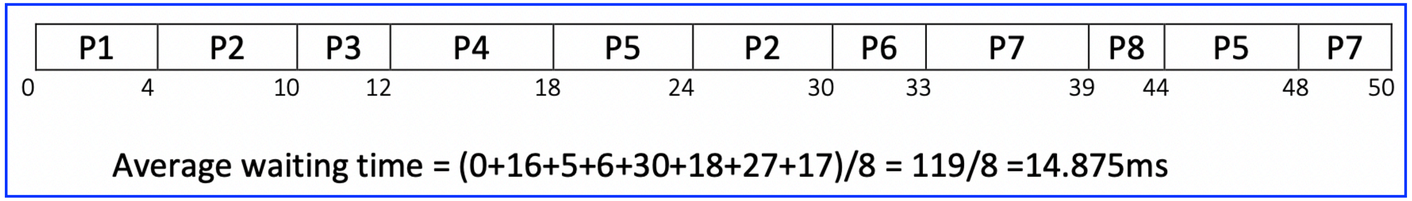
Scheduling algorithms and waiting time (5 points)

1. Non-preemptive SJF (shortest job first)



Average waiting time = (0+2+11+15+32+6+17+5)/8 = 88/8 =11ms

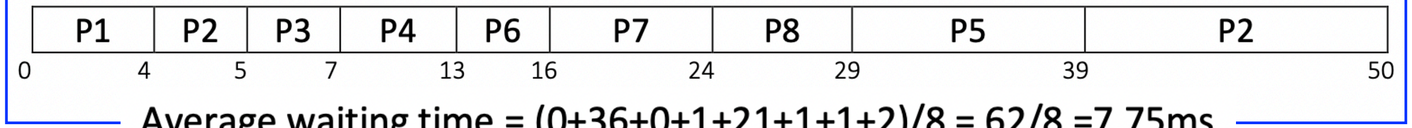
2. Preemptive RR (round robin) with a time slice (quantum) of 6 ms



Average waiting time = (0+16+5+6+30+18+27+17)/8 = 119/8 =14.875ms

3. Preemptive SRTF (shortest remaining time first)

Answer:



Average waiting time = (0+36+0+1+21+1+1+2)/8 = 62/8 =7.75ms

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

Common mistakes:

- errors in calculating the sum or the average… (partial credit when only making small mistakes)
- using an incorrect definition of waiting time

8. I/O, Disk Scheduling and File systems

Unix file I/O (3 points)

Consider the following function read_input().

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

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.

d – lseek seeks to absolute byte position 3 in the file. Positions are counted starting at offset **0.**
The file only contains the four given characters, so a read attempt of 1023 characters only returns the d, which is stored in buf[0].

Common errors:

- Off-by-one, e.g. assuming the file starts at index 1 instead of 0

8. I/O, Disk Scheduling and File systems

Unix file I/O (3 points)

Consider the following function read_input().

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

What is the purpose of the memset() call in the code above?

It clears the buffer pointed to by buf, ensure that the string read ends with a ***null/zero byte*** ('\0', 0x00).

Common errors:

- stating that the string needs to end with a ***zero*** (this could also mean the ASCII character '0' = 0x30 and is imprecise).
- Stating that the buffer at ****buf*** is cleared (this is the content of the buffer, not the address)

8. I/O, Disk Scheduling and File systems

Disk scheduling (3 points)

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).

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

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

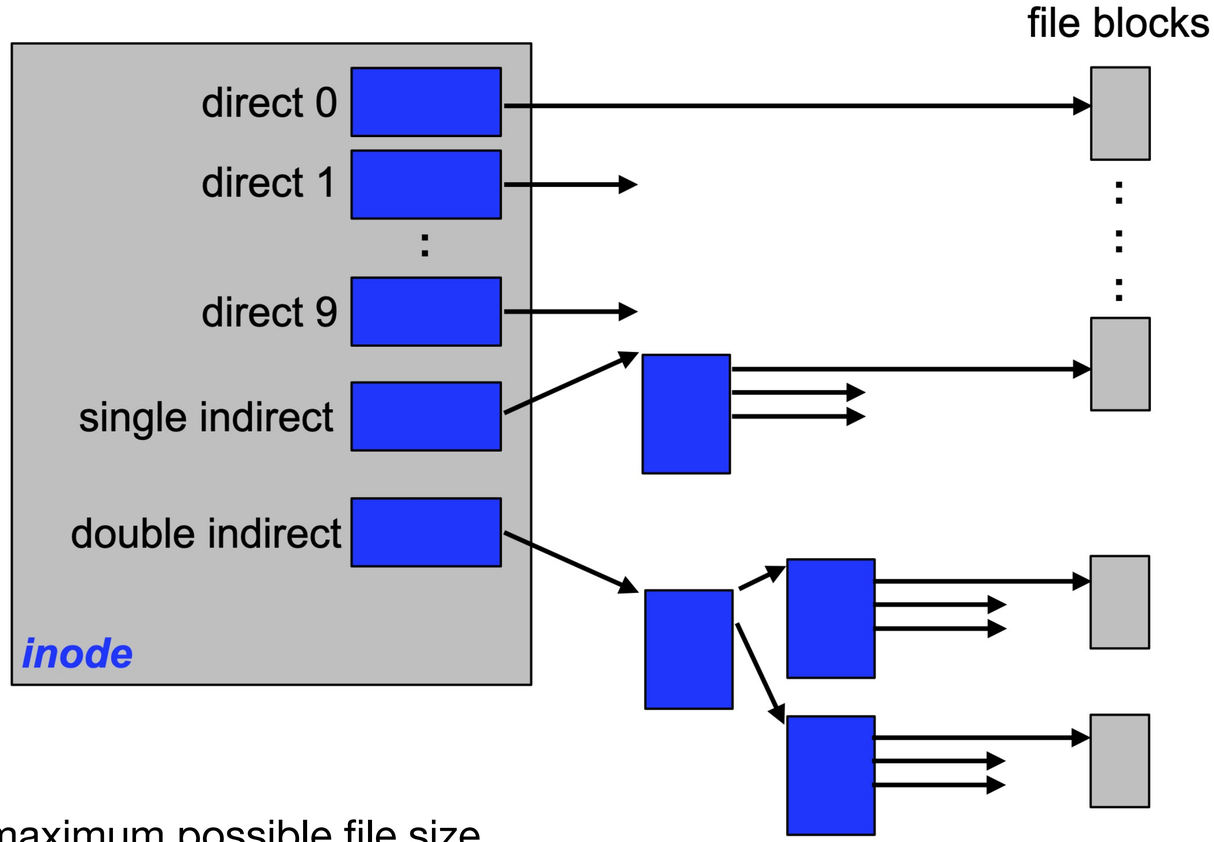
Give the order of tracks that are read for an I/O scheduler that uses the First In First Out (FIFO) strategy:

FIFO is the most simple order, so the result is: 4, 7, 11, 3, 2, 13, 1, 15, 5, 6

Common mistakes:

- thinking too complex, this is simply FIFO…

8. I/O, Disk Scheduling and File systems



Inode-based file systems (4 points)

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.

File size = (Block size) * (Number of blocks) = (Block size) * (10+ (number of indir. block entries) + (number of indir. block entries)^2)
= 1024 Bytes * (10 + 256 + 256*256) **// 256 entries per indirect block = 1024 bytes block size / 4 bytes block number**
= 67.381.248 Bytes
= 65.802 kB
= ca. 64.3 MB (all forms of the result were valid, ***calculation was required!***). Partial credit for simple calculation errors.

Common mistakes: only 9 instead of 10 direct blocks, assuming double and triple indirect blocks instead of single and double.