Operating Systems

Solutions for the example exam questions

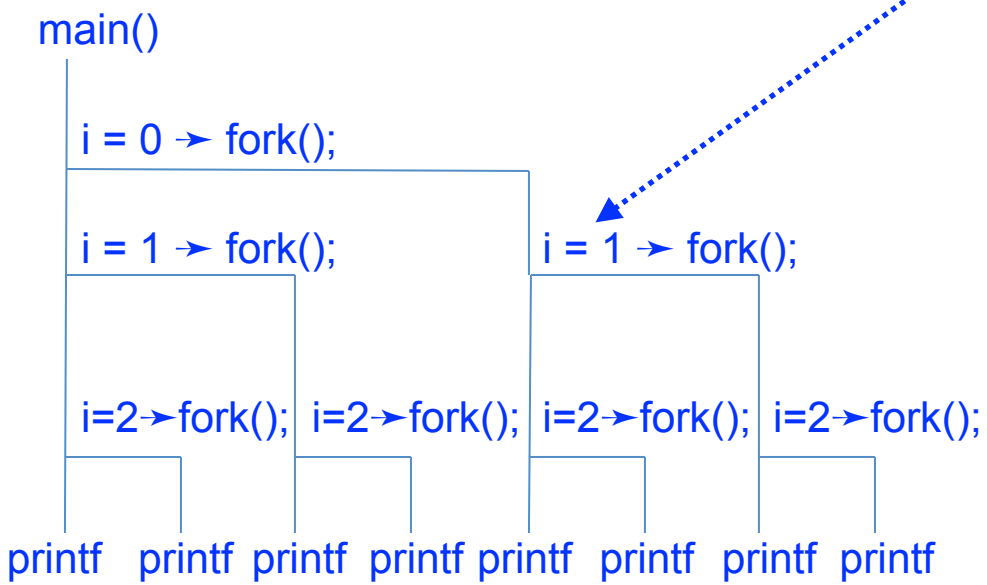
Michael Engel

1 Processes

1.1 Process execution order

How many times does the following program print “Hello World”? Draw a simple tree diagram to show the parent-child hierarchy of the spawned processes.

“Hello World” is printed 8 (eight) times:



You had to recognize that `fork()` copies the process state, especially the value of the loop variable `i`!

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main() {
5    int i;
6    for (i = 0; i < 3; i++)
7      fork();
8    printf("Hello World\n");
9    return 0;
10 }
```

1 Processes

1.2 Process execution order (this should read process synchronization…)

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

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.

```
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 Shell pipelines

In a very special Unix shell, the command A <> B means that the standard output of A should be connected to the standard input of B and the standard input of A should be connected to the standard output of B (so this creates a bidirectional shell pipeline).

The following C code is an excerpt of a shell that should implement this behavior. It contains a number of errors. Indicate the location of each error (give the line number), shortly describe the problem and propose fixes that realize the desired behavior.

(continued on the next slide…)

2 Shell pipelines

…Indicate the location of each error (give the line number), shortly describe the problem and propose fixes that realize the desired behavior.

1. `#define WRITE 1`
   With 2, the program would cause a buffer overflow of arrays `pipe1` and `pipe2`
2. `pipe2[2]`
   Wrong dimension of 1 used here
3. `dup2(pipe2[READ], STDIN);`
   Wrong order of arguments to dup2, would lead to overwriting the file descriptor stored in `pipe2[READ]` instead of `STDIN`
4. `pipe2[WRITE];`
   The code closes both sides of one pipe
5. `if (fork() == 0) {`
   We need two child processes for our special pipeline, so both fork calls need to execute code in the new process

```
1  #define WRITE  2   ①
2  #define READ   0
3
4  int pipe1[2], pipe2[1]; ②
5
6  pipe(pipe1);
7  pipe(pipe2);
8
9  if(fork() == 0) {
10      dup2(pipe1[WRITE], STDOUT);
11      dup2(STDIN, pipe2[READ]); ③
12      close(pipe1[READ]);
13      close(pipe1[WRITE]); ④
14      exec(A);
15 }
16
17 if(fork() != 0) { ⑤
18      dup2(pipe1[READ], STDIN);
19      dup2(pipe2[WRITE], STDOUT);
20      close(pipe1[WRITE]);
21      close(pipe2[READ]);
22      exec(B);
23 }
24
25 close(pipe1[READ]);
26 close(pipe1[WRITE]);
27 close(pipe2[READ]);
28 close(pipe2[WRITE]);
```

3 Threading

The following code that ***uses two separate threads*** describes a famous bug in MySQL, a widely used open source SQL database server.

```
1  // Thread 1:
2
3  if (thd->proc_info) {
4    ...
5    fputs(thd->proc_info, ...);
6    ...
7  }
8
9  // Thread 2:
10 thd->proc_info = NULL;
```

3.1. Explain the problem in the code

The code contains an atomicity problem:

The test of `thd->proc_info` and its use in the `fputs` statement are not in a critical section.

After the test the value of `thd->proc_info` could be changed to `NULL` by Thread 2, invalidating the purpose of the `if` in Thread 1. This would result in a crash, since the `FILE*` passed to `fputs` is now `NULL`.

3 Threading

The following code that ***uses two separate threads*** describes a famous bug in MySQL, a widely used open source SQL database server.

```
1  Semaphore proc_info_lock = 1;
2
3  // Thread 1:
4       This is the actual critical region!
5  if (thd->proc_info) {
6       ...
7       wait(&proc_info_lock);
8       fputs(thd->proc_info, ...);
9       signal(&proc_info_lock);
10      ...
11 }
12
13 // Thread 2:
14
15 wait(&proc_info_lock);
16 thd->proc_info = NULL;
17 signal(&proc_info_lock);
```

3.2. Does the following code fix the problem? Explain your answer.

The code still contains an atomicity problem because the test of `thd->proc_info` and its use in the `fputs` statement are still not in a critical region *together*.

Thus, after the test the value of `thd->proc_info` could still be changed to `NULL` by Thread 2 despite the use of the locks.

4 Deadlocks

Consider the following code.

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

```
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);
```

4 Deadlocks

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.

```
1  Semaphore L1=1, L2=1, L3=1;
2
3  // Thread 1:
4  wait(L1);
5  wait(L2);
6  // critical section requiri
7  signal(L2);
8  signal(L1);
9
10 // Thread 2:
11 wait(L3);
12 wait(L1);
13 // critical section requiri
14 signal(L1);
15 signal(L3);
16
17 // Thread 3:
18 wait(L2);
19 wait(L3);
20 // critical section requiri
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:
  thread 1 waiting for L2 (held by thread 3),
  thread 2 waiting for L1 (held by thread 1),
  thread 3 waiting for L3 (held by thread 2).

4 Deadlocks

4.2 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.

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

5 Memory management

5.1 Buddy allocation

Use dynamic memory allocation according to the buddy method. The second row of each table below shows the current allocation of memory with a total memory size of 32 MB and a block size of 2 MB. Fill in markings for the allocated blocks according to the request given along with each table.

Buddy: always round up to the nearest power of two!

Scenario 1: Process C requests 3 MB.

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

Scenario 2: Process D requests 12 MB.

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

5 Memory management

5.1 Buddy allocation

Use dynamic memory allocation according to the buddy method. The second row of each table below shows the current allocation of memory with a total memory size of 32 MB and a block size of 2 MB. Fill in markings for the allocated blocks according to the request given along with each table.

Scenario 3: Process E requests 14 MB.

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

This request cannot be satisfied – no free block of 16 MB size is available!

Scenario 4: Process F requests 7 MB.

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

5 Memory management

5.2 Fragmentation

The memory allocation algorithms discussed in the course can result in external and internal fragmentation. Explain the key points for both terms and name a placement strategy (but not the same strategy twice!) for which the respective fragmentation effect shows up.

External fragmentation:

- Memory fragments that cannot be used for additional allocations are created *outside of* the allocated memory area
- For list-based strategies such as First Fit, Best Fit, ...

Internal fragmentation:

- There is unused memory *inside of* the allocated memory areas
- e.g. for the Buddy allocator, since all requests are rounded up to the nearest larger power of two

5 Memory management

5.3 Virtual memory

Describe shortly what a TLB is and why it is required to build efficient computers using virtual memory.

A TLB (Translation Lookaside Buffer) is a ***cache*** for ***page table entries***.

Page tables are stored in main memory. Without a TLB (or in case of a *TLB miss*), to translate a virtual into a physical address, a ***page table walk*** is required to traverse the page table, which is a tree data structure.

For a page table of depth 3, three main memory accesses for the page table walk would be required to obtain the translation for a single virtual address translation ➛ unacceptable performance degradation!

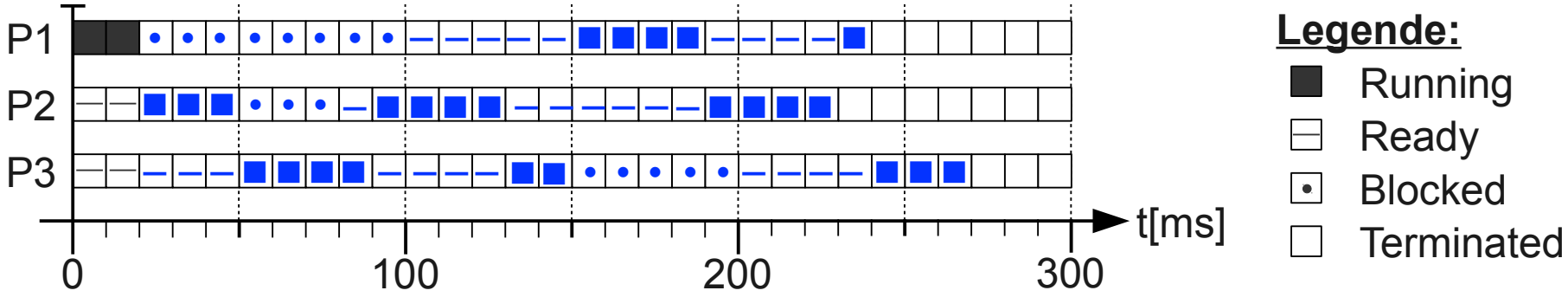
TLBs (and caches in general) improve performance due to the ***locality principle*** (you could also explain temporal and spatial locality shortly here if you like)

6 Scheduling

Process	P1	P2	P3
Compute time	70	110	90
I/O start	20	30	60
I/O duration	80	30	50

An operating system has three processes P1, P2 and P3.
The processes are activated in the order P1, P2, P3 and are all ready to run at time t = 0. The compute times in milliseconds (ms) of each process and the starting time of I/O operations *relative to the compute time* are given in the following table.

Use the following Gantt diagram to show how the three processes P1, P2, and P3 are processed if the scheduling strategy is round robin with a chosen time slice of 40 ms. Each process performs exactly one I/O operation with the starting time and duration given in the table. The process switching time can be neglected. Mark the related process state for each process and time step according to the patterns shown next to the diagram.



7 File I/O

7.1 File reads

Assume the initial contents of file myfile2 and the execution of the following block of code. The initial contents of the file myfile2 are: `foo#bar#baz`

What is the output printed by the program?

The output is `"ar#"`:

- `lseek` (l. 8) positions the file read pointer to the fifth character (count from 0): `foo#bar#baz`
- `read` (l. 15) reads 3 chars from the file starting with "a": `foo#bar#baz`
- Finally, line 20 sets the string end to the fourth character in buffer

```
1  int fd = open("myfile2", O_RDWR);
2
3  if (fd <= 0) {
4    printf("Error");
5    exit(-1);
6  }
7
8  if(lseek(fd, 5, SEEK_SET) < 0) {
9    printf("Error");
10   exit(-1);
11 }
12
13 char buffer[100];
14
15 if(read(fd, buffer, 3) != 3) {
16   printf("Error");
17   exit(-1);
18 }
19
20 buffer[3] = 0;
21
22 printf("%s\n", buffer);
23 close(fd);
```

7 File I/O

```
1  int fd = open("myfile", O_WRONLY | O_APPEND | O_TRUNC) ;
2
3  if (fd < 0) {
4    printf("Error");
5    exit(-1);
6  }
7
8  write(fd, "dee", 3);
9  close(fd);
```

Important: without O_TRUNC!
See the update
on the following slide!

7.2 File writes

Assume the initial contents of file myfile and the execution of the following block of code. What are the final contents of myfile? The initial contents of the file myfile are: `Tweedle`

Assume that there are no errors with permissions and that there are no newlines.

Here, the `O_APPEND` flag to `open` is important:

`O_APPEND` `append on each write`

Thus, since the file exists and already has the content `Tweedle`, the `write` call in line 8 appends the buffer contents at the end of the file, resulting in:

`Tweedledee`

7 File I/O

```
1  int fd = open("myfile", O_WRONLY | O_APPEND | O_TRUNC) ;
2
3  if (fd < 0) {
4    printf("Error");
5    exit(-1);
6  }
7
8  write(fd, "dee", 3);
9  close(fd);
```

This is the corrected version including O_TRUNC

7.2 File writes

Assume the initial contents of file myfile and the execution of the following block of code. What are the final contents of myfile? The initial contents of the file myfile are: `Tweedle`

Assume that there are no errors with permissions and that there are no newlines.

Here, the `O_APPEND` and `O_TRUNC` flags to `open` are important:

`O_APPEND`	`append on each write`
`O_TRUNC`	`truncate size to 0`

Thus, when the file is opened for writing, it is truncated (emptied: size = 0).

Afterwards, the `write` call in line 8 appends the buffer contents at the end of the (empty) file, resulting in:

```
dee
```

8 File systems

A Unix filesystem has 2 kB (1 kB = 1024 bytes) blocks and 4 byte disk addresses. Each inode contains 10 direct entries, one singly-indirect entry and one doubly-indirect entry.

8.1. Calculate the maximum possible file size in this file system

10*2 kB + (2048/4)*2 kB + (2048/4) * (2048/4) *2 kB
= 20 kB + 1024 kB + 524288 kB = 525332 kB (or 537939968 bytes or 513.02 MB)

8.2. Suppose half of all files are exactly 1.5 kB in size and the other half of all files are exactly 2 kB. What fraction of the disk space would be wasted? (Consider only the blocks used to store data)

Both 1.5 kB and 2 kB files will use 2 kB of space. For each 2 kB file, 0 kB is wasted; for each 1.5 kB file, 0.5 kB is wasted.
Therefore, the fraction wasted is (0/2)*50%+(0.5/2)*50% = 12.5%.

8.3. Based on the same condition as in the previous item above, does it help to reduce the fraction of wasted disk space if we change the block size to 1 kB? Justify your answer.

No. Nothing is changed. Both 1.5 kB and 2 kB files will still use 2 kB of space. For each 2 kB file, 0 kB is wasted; for each 1.5 kB file, 0.5 kB is wasted.
Therefore, the fraction wasted is (0/2)*50%+(0.5/2)*50% = 12.5%, which is unchanged.

9 Security

9.1 Unix login

The Unix login program checks whether or not a user has entered the correct password before taking on the user's ID and executing the user's shell.

Explain how it can check passwords this way without storing the user's actual password on the system.

The code examples here are only pseudocode, of course!

login can check either against the *encrypted* version of a password:

```
if (compare(encrypt(password), stored_encrypted_password) == CORRECT) …
```

or it can check against a stored password *hash*:

```
if (hash(password), stored_hashed_password) == CORRECT) …
```

To keep the hash or encrypted password safe from regular users of the system, login reads a hash or encrypted password from `/etc/shadow`, which is not readable by a normal user (login runs with root privileges).

9 Security

9.2 System subversion

Suppose an attacker breaks into a Unix machine, obtains root (superuser) privileges, and manages to keep them for a long period of time (e.g., many months).

What might the attacker do to learn users' real passwords, even if they are not stored on the system?

The attacker could replace login with a malicious version that records the passwords users enter in a file where the attacker can read them.

10 Storage systems

Many RAID devices now ship with the following options:

- RAID 0 - data striped across all disks
- RAID 1 - each disk mirrored
- RAID 5 - striped parity

Given a system with 8 disks of 1 TB each, answer the following questions:

10.1 For each of the three RAID levels, how much usable storage does the system provide?

- RAID 0: striping, no redundancy ➛ 8 * 1 TB = 8 TB
- RAID 1: mirroring, two copies of a disk ➛ two copies of (4 * 1 TB) = 4 TB
- RAID 5: one parity disk used, 7 disks for data ➛ 7 * 1 TB = 7 TB

10 Storage systems

10.2 Assume a workload consisting only of small reads, evenly distributed. Assuming that no verification is performed on reads, what is the throughput of each level assuming that a single disk can perform 100 reads/sec?

- RAID 0: 800 reads/sec
- RAID 1: 800 reads/sec – reads can be satisfied from both disks in a pair
- RAID 5: 800 reqs/sec – no need to read the parity (since no verification!), so no loss of read performance, only space

10.3. Assume a workload consisting only of small writes, evenly distributed. Again, calculate the throughput assuming that a single disk can execute 100 writes/sec.

- RAID 0: 800 writes/sec
- RAID 1: 400 writes/sec – need to write to both disks in a pair
- RAID 5: 200 writes/sec if you do two reads + two writes to update the parity, or 100 writes/sec if you read all of the disks to recalculate the parity

10 Storage systems

10.4. For each level, what is the minimum number of disks that may fail before data may be lost?

- RAID 0: no redundancy ➛ 0 disk failures allowed
  (but data loss is guaranteed at the first lost disk)
- RAID 1: 2 if you happen to lose both disks in a pair
- RAID 5: 2, but data loss is guaranteed on failure of the second disk

10.5. For each level, what is the minimum number of disks that must fail to **guarantee** data loss?

- RAID 0: 1
- RAID 1: 5, if you happen to get *really lucky* and lose one from each pair before losing the fifth
- RAID 5: 2