9th Slide Set Operating Systems

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences (1971–2014: Fachhochschule Frankfurt am Main) Faculty of Computer Science and Engineering christianbaun@fb2.fra-uas.de

Learning Objectives of this Slide Set

- At the end of this slide set You know/understand...
 - what critical sections and race conditions are
 - what synchronization is
 - how signaling influences the execution order of the processes
 - how critical sections can be secured via blocking
 - what problems (starvation and deadlocks) may arise from blocking
 - how deadlock detection with matrices works
 - different options to implement **communication** between processes:
 - Shared memory
 - Message queues
 - Pipes
 - Sockets
 - different options to implement **cooperation** between processes
 - how critical sections can be protected via semaphores
 - the difference between semaphore and mutex

Exercise sheet 9 repeats the contents of this slide set which are relevant for these learning objectives

Interprocess Communication (IPC)

- Processes do not only carry out read and write operations on data, but also:
 - call each other
 - wait for each other
 - coordinate with each other
 - In short: They must interact with each other
- Important questions regarding interprocess communication (IPC):
 - How can a process to transmit information to others?
 - How can multiple processes access shared resources?

Question: What is the situation here with threads?

- For threads, the same challenges and solutions exist as for interprocess communication with processes
- Only the communication between the threads of a process is no problem because they
 operate in the same address space

Critical Sections

- If multiple processes run in parallel, the processes consist of. . .
 - Uncritical sections: The processes do not access shared data or carry out only read operations on shared data
 - Critical sections: The processes carry out read and write operations on shared data
 - Critical sections must not be processed by multiple processes at the same time
- In order for processes to be able to access a shared memory (⇒⇒ common data), the operating system must provide mutual exclusion

Critical Sections – Example: Print Spooler

```
Process X
                                                                                Process Y
next free slot = in: (16)
                                             Process switch
                                                              next_free_slot = in; (16)
                                                              Store record in next free slot: (16)
                                                              in = next free slot + 1: (17)
                                             Process switch
Store record in next free slot: (16)
in = next free slot + 1: (17)
                        Spooler directory

    The spooling directory is

                                                                consistent
                         Master Thesis pdf
                                               out = 12
                     13
                             Project.ps
                     14
                              Email.txt
Process X
                                                                       X and got lost
                         not important.doc
                      16
                                               in = 16
Process Y
```

- - But the entry of process Y was overwritten by process
- Such a situation is called race condition

Race Condition

- Unintended race condition of 2 processes, which want to modify the value of the same record
 - The result of a process depends on the order or timing of other events
 - Frequent reason for bugs, which are hard to locate and fix
- Problem: The occurrence of the symptoms depends on different events
 - The symptoms may be different or disappear with each test run
- Race conditions can be avoided with the semaphore concept
 (⇒ slide 64)

Therac-25: Race Condition with tragic Result (1/2)

- Therac-25 is a linear particle accelerator for the radiation therapy of cancer tumors
- Mid-1980s: In the United States some accidents happened because of poor programming and quality assurance
 - Some patients got an up to 100 times increased radiation dose

Image source: Google image search



Therac-25: Race Condition with tragic Result (2/2)

An Investigation of the Therac-25 Accidents. Nancy Leveson, Clark S. Turner IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41 http://courses.cs.vt.edu/-cs3604/1ib/Therac_25/Therac 1.html

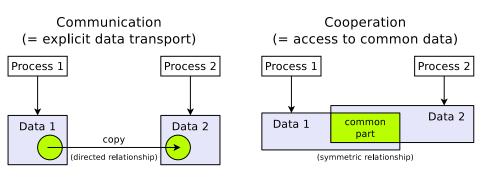
- 3 patients died because of bugs
- 2 patients died because of a race condition, which resulted in inconsistent settings of the device, causing an increased radiation dose
 - The control process did not synchronize correctly with the user interface process
 - The bug only occurred in case the operator was too fast
 - During testing, the bug did not occur, because experience (routine) was required to operate the device this fast





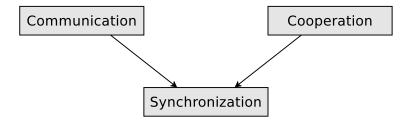
Communication vs. Cooperation

- Interprocess communication has 2 aspects:
 - Functional aspect: communication and cooperation
 - Temporal aspect: synchronization



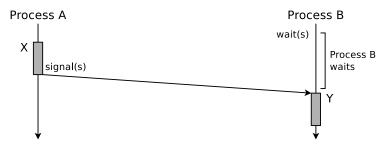
Forms of Interaction

- Communication and cooperation base on synchronization
 - Synchronization is the most elementary form of interaction
 - Reason: communication and cooperation need a synchronization between the interacting partners to obtain correct results
 - Therefore, we first discuss the synchronization

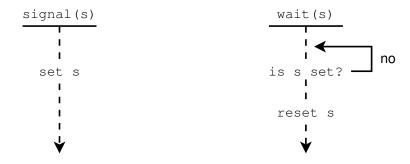


Signaling

- One way to synchronize processes
- Used to specify an execution order
- Example: Section **X** of process P_A must be executed **before** section **Y** of process P_R
 - The signal operation signals that process P_A has finished section **X**
 - Perhaps, process P_B must wait for the signal of process P_A



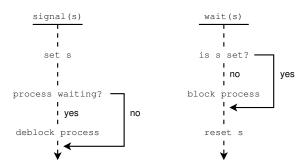
Most Simple Form of Signaling (Busy Waiting)



- The figure shows busy waiting at the signal variable s
 - The signal variable can be located in a local file, for example
 - Drawback: CPU resources are wasted, because the wait operation occupies the processor at regular intervals
- This technique is also called spinlock or polling

Signal and Wait

- Better concept: Blocking of process P_B until process P_A has finished section \mathbf{X}
 - Advantage: No CPU resources are wasted
 - Drawback: Only a single process can wait
 - In literature, this technique is also called passive waiting

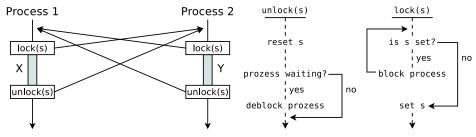


One way to specify in Linux an execution order with passive waiting, is by using the function sigsuspend. Thereby a process blocks itself until another process sends it an appropriate signal (usually SIGUSR1 or SIGUSR2) with the command kill (or the system call of the same name) and in this way signals that it should continue working.

Alternative system calls and function calls by which a process can block itself until it is woken up again by a system call are pause and sleep

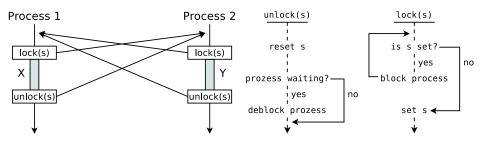
Securing critical Sections by Locking / Blocking

- Signaling always specifies the execution order
 - But if it is just necessary to ensure that there is no overlap in the execution of the critical sections, it is possible to use the two operations lock and unlock



- Blocking (locking) prevents the overlapping execution of 2 critical sections
 - Example: Critical Sections **X** of process P_A and **Y** of process P_B

Locking and Unlocking Processes in Linux (1/2)



Useful system calls and standard library function to call the operations lock and unlock in Linux sigsuspend, kill, pause and sleep

- Alternative 1: Implementation of locking with the signals SIGSTOP (No. 19) and SIGCONT (No. 18)
 - With SIGSTOP another process can be stopped
 - With SIGCONT another process can be reactivated

Locking and Unlocking Processes in Linux (2/2)

- Alternative 2: A local file serves as a locking mechanism for mutual exclusion
 - Each process verifies before entering its critical section whether it can open the file exclusively
 - e.g. with the system call open or the standard library function fopen
 - If this is not the case, it must pause for a certain time (e.g. with the system call sleep) and then try again (busy waiting).
 - Alternatively, it can pause itself with sleep or pause and hope that the
 process that has already opened the file unblocks it with a signal at the
 end of its critical section (passive waiting)

Summary: Difference between Signaling and Blocking

- Signaling specifies the execution order
 Example: Execute section X of process P_A before section Y of P_B
- Blocking / Locking secures critical sections
 The execution order of the critical sections of the processes is not specified! It is just ensured that the execution of critical sections does not overlap

Problems caused by Blocking

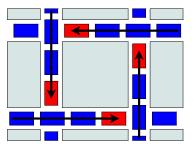
Image Source: Google Image Search

Starvation

 If a process does never remove a lock, the other processes need to wait infinitely long for the release

Deadlock

- If several processes wait for resources, locked by each other, they lock each other mutually
- Because all processes, which are involved in the deadlock, must wait forever, no one can initiate an event that resolves the situation





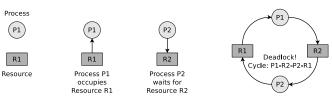
Conditions for Deadlock Occurrence

System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. Computing Surveys, Vol. 3, No. 2, June 1971, pp. 67-78 http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

- A deadlock situation can arise if these conditions are all fulfilled
 - Mutual exclusion
 - At least 1 resource is occupied by exactly 1 process or is available
 non-sharable
 - Hold and wait
 - A process, which currently occupies at least 1 resource, requests additional resources which are being held by another process
 - No preemption
 - Resources, which are occupied by a process can not be deallocated by the operating system, but on released by the holding process voluntarily
 - Circular wait
 - A cyclic chain of processes exists
 - Each process requests a resource that the next process in the chain occupies.
- If one of these conditions is not fulfilled, no deadlock can occur

Resource Graphs

- The relations of processes and resources can be visualized using directed graphs
- In this way, deadlocks can also be modeled
 - The nodes of a resource graph are:
 - Processes: Are shown as circles
 - Resources: Are shown as rectangles
 - An edge from a process to a resource means:
 - The process is blocked because it waits for the resource
 - An edge from a resource to a process means:
 - The process occupies the resource



A good description of resource graphs provides the book **Betriebssysteme** – **Eine Einführung**, *Uwe Baumgarten, Hans-Jürgen Siegert*, 6th Edition, Oldenbourg Verlag (2007), Chapter 6

Deadlock Detection with Matrices

- One drawback of deadlock detection with resource graphs is that only individual resources can be represented with it
 - If multiple copies (instances) of a resource exist, then graphs are not suited for the visualisation and detection of deadlocks
 - If multiple copies of a resource exist, a matrices-based algorithm can be used, which requires 2 vectors and 2 matrices
- We specify 2 vectors
 - Existing resource vector
 - Indicates the number of existing resources of each class
 - Available resource vector
 - Indicates the number of free resources of each class
- Additionally 2 matrices are required
 - Current allocation matrix
 - Indicates, which resources are currently occupied by the processes
 - Request matrix
 - Indicates, which resource the processes would like to occupy

Deadlock Detection with Matrices – Example (1/2)

Source of the example: Tanenbaum. Moderne Betriebssysteme. Pearson. 2009

Existing resource vector = $\begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$

- 4 resources of class 1 exist
- 2 resources of class 2 exist
- 3 resources of class 3 exist
- 1 resource of class 4 exist

- Process 1 occupies 1 resource of class 3
- Process 2 occupies 2 resources of class 1 and 1 resource of class 4
- Process 3 occupies 1 resource of class 2 and 2 resources of class 3

Available resource vector = $\begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$

- 2 resources of class 1 are available
- 1 resource of class 2 is available
- No resources of class 3 are available
- No resources of class 4 are available

Request matrix =
$$\begin{vmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{vmatrix}$$

- Process 1 is blocked, because no free resources of class 4 exist
- Process 2 is blocked, because no free resources of class 3 exist
- Process 3 is not blocked

Deadlock Detection with Matrices – Example (2/2)

If process 3 finished executing, it deallocates its resources

Available resource vector
$$= \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$$

- 2 resources of class 1 are available
- 2 resources of class 2 are available
- 2 resources of class 3 are available
- No resources of class 4 are available
- If process 2 finished executing, it deallocates its resources

Request matrix =
$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ - & - & - & - \end{bmatrix}$$

- Process 1 is blocked, because no free resources of class 4 exist
- Process 2 is not blocked

$$\text{Available resource vector} = \left(\begin{array}{cccc} 4 & 2 & 2 & 1 \end{array} \right) \qquad \text{Request matrix} = \left[\begin{array}{cccc} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{array} \right]$$

● Process 1 is not blocked ⇒ no deadlock in this example

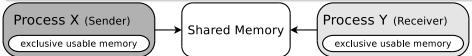
Conclusion about Deadlocks

- Sometimes it is tolerated that deadlocks can occur
 - What matters is how important a system is
 - A deadlock, which statistically occurs every 5 years, is not a problem in a system, which crashes because of hardware failures or other software problems one time per week
- Deadlock detection is complicated and causes overhead
- In all operating systems, deadlocks can occur:
 - Full process table
 - No more new processes can be created
 - Maximum number of inodes allocated
 - No new files or directories can be created
- ullet The probability that this happens is low, but eq 0
 - Such potential deadlocks are accepted because an occasional deadlock is not as troublesome as the otherwise necessary restrictions (e.g. only 1 running process, only 1 open file, more overhead)

Shared Memory

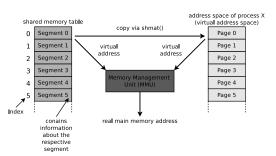
- Interprocess communication via a shared memory is also called memory-based communication
- **Shared memory segments** are memory areas, which can be accessed by multiple processes
 - These memory areas are located in the address space of multiple processes
- The processes need to coordinate the access operations by themselves and ensure that their memory requests are mutually exclusive
 - A receiver process, cannot read data from the shared memory, before the sender process has finished its current write operation
 - ullet If access operations are not coordinated carefully \Longrightarrow inconsistencies

In all other forms of interprocess communication, the operating system takes care about the synchronization of the access operations



Shared Memory in Linux/UNIX

- Linux/UNIX operating systems contain a **shared memory table**, which contains information about the existing shared memory segments
 - This information includes: Start address in memory, size, owner (username and group) and privileges



 A shared memory segment is always addressed via its index number in the shared memory table

- Advantage:
 - A shared memory segment which is not attached to a process, is not erased by the operating system automatically

Working with Shared Memory

Linux/UNIX operating systems provide 4 system calls for working with shared memory

- shmget(): Create a shared memory segment or access an existing one
- shmat(): Attach a shared memory segment to a process
- shmdt(): Detach a shared memory segment from a process
- shmctl(): Request status information (e.g. privileges) of a shared memory segment, modify or erase it

One example of working with shared memory segments in Linux can be found on the website of this course

ipcs

The command ipcs provides information about existing shared memory segments

1 #include <sys/ipc.h>

```
2 #include <sys/shm.h>
 3 #include <stdio.h>
   #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
 7
       int shared memory id = 12345:
8
       int returncode_shmget;
9
10
       // Create shared memory segment or access an existing one
11
       // IPC CREAT = create a shared memory segment, if it does not still exist
12
       // 0600 = Access privileges for the new message queue
13
       returncode shmget = shmget(shared memory id, MAXMEMSIZE, IPC CREAT | 0600);
14
15
       if (returncode_shmget < 0) {
16
           printf("Unable to create the shared memory segment.\n");
17
           perror("shmget"):
       } else {
18
19
           printf("The shared memory segment has been created.\n");
20
21 }
```

Attach a Shared Memory Segment (in C)

```
1 #include <sys/types.h>
  #include <sys/ipc.h>
 3 #include <svs/shm.h>
   #include <stdio.h>
   #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
 9
       int returncode_shmget;
10
       char *sharedmempointer:
11
12
       // Create shared memory segment or access an existing one
13
       returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
14
15
16
           // Attach shared memory segment
17
           sharedmempointer = shmat(returncode shmget, 0, 0):
18
           if (sharedmempointer == (char *)-1) {
19
               printf("Unable to attach the shared memory segment.\n");
20
               perror("shmat"):
21
           } else {
22
               printf("The shared memory segment has been attached %p\n", sharedmempointer);
23
           }
24
25
```

```
$ ipcs -m
       Shared Memory Segments
kev
            shmid
                        owner
                                    perms
                                                bytes
                                                            nattch
                                                                         status
                                    600
0x00003039 56393780
                        bnc
                                                20
```

Write into a Shared Mem. Segment and read from it (in C)

```
1 #include <sys/types.h>
 2 #include <sys/ipc.h>
 3 #include <svs/shm.h>
 4 #include <stdio.h>
 5 #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
9
       int returncode_shmget, returncode_shmdt, returncode_sprintf;
10
       char *sharedmempointer:
11
12
       // Create shared memory segment or access an existing one
13
       returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
14
15
           // Attach shared memory segment
16
           sharedmempointer = shmat(returncode_shmget, 0, 0);
17
18
19
           // Write a string into the shared memory segment
20
           returncode_sprintf = sprintf(sharedmempointer, "Hallo Welt.");
21
           if (returncode sprintf < 0) {
22
               printf("The write operation did fail.\n");
23
           } else {
24
               printf("%i chareacters written into the segment.\n". returncode sprintf):
25
           }
26
27
           // Read the string from the shared memory segment
28
           if (printf ("%s\n", sharedmempointer) < 0) {
29
               printf("The read operation did fail.\n");
30
           }
31
```

Detach a Shared Memory Segment (in C)

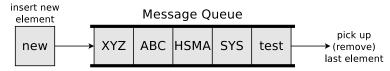
```
1 #include <sys/types.h>
 2 #include <sys/ipc.h>
 3 #include <svs/shm.h>
 4 #include <stdio.h>
  #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
 9
       int returncode_shmget;
10
       int returncode shmdt:
11
       char *sharedmempointer:
12
13
       // Create shared memory segment or access an existing one
14
       returncode shmget = shmget(shared memory id. MAXMEMSIZE. IPC CREAT | 0600):
15
16
17
           // Attach the shared memory segment
18
           sharedmempointer = shmat(returncode_shmget, 0, 0);
19
20
21
           // Detach the shared memory segment
22
           returncode_shmdt = shmdt(sharedmempointer);
23
           if (returncode shmdt < 0) {
24
               printf("Unable to detach the shared memory segment.\n");
25
               perror("shmdt");
26
           } else {
27
               printf("The shared memory segment has been detached.\n"):
28
           }
29
       }
30
```

Erase a Shared Memory Segment (in C)

```
1 #include <sys/types.h>
 2 #include <svs/ipc.h>
 3 #include <sys/shm.h>
  #include <stdio.h>
   #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
 8
       int shared_memory_id = 12345;
       int returncode_shmget;
10
       int returncode_shmctl;
11
       char *sharedmempointer;
12
13
       // Create shared memory segment or access an existing one
14
       returncode shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
15
16
17
           // Ease shared memory segment
18
           returncode shmctl = shmctl(returncode shmget, IPC RMID, 0):
19
           if (returncode shmctl == -1) {
20
               printf("Unable to erase the shared memory segment.\n");
21
               perror("semctl");
22
           } else {
23
               printf("The shared memory segment has been erased.\n");
24
           }
25
26
```

Message Queues

- Are linked lists with messages
- Operate according to the FIFO principle
- Processes can store data inside and picked them up from there
- Benefit: Even after the termination of the process, which created the message queue, the data inside the message queue stays available



Linux/UNIX operating systems provide 4 system calls for working with message queues

- msgget(): Create a message queue or access an existing one
- msgsnd(): Write messages into message queues (⇒ send operation)
- msgrcv(): Read messages from message queues (⇒ receive operation)
- msgct1(): Request status information (e.g. privileges) of a message queue, modify or erase it

The command ipcs provides information about existing message queues

1 #include <stdlib.h> 2 #include <sys/types.h> 3 #include <sys/ipc.h>

12345

Create Message Queues (in C)

```
#include <stdio.h>
 5 #include <svs/msg.h>
 6
   int main(int argc, char **argv) {
 8
       int returncode_msgget;
 9
10
       // Create message queue or access an existing one
11
       // IPC CREAT => create a message queue, if it does not still exist
12
       // 0600 = Access privileges for the new message queue
13
       returncode_msgget = msgget(12345, IPC_CREAT | 0600);
14
       if(returncode_msgget < 0) {</pre>
15
            printf("Unable to create the message queue.\n");
16
            exit(1):
17
       } else {
18
            printf("The message queue 12345 with the ID %i has been created.\n",
                 returncode_msgget);
19
20 }
$ ipcs -a
       Message Queues
kev
           msaid
                                              used-bytes
                       owner
                                  perms
                                                           messages
0x00003039 98304
                      hn c
                                  600
```

\$ printf "%d\n" 0x00003039 # Convert from hexadecimal to decimal

Write Messages into Message Queues (in C)

```
1 #include <stdlib.h>
 2 #include <sys/types.h>
 3 #include <sys/ipc.h>
  #include <stdio.h>
 5 #include <svs/msg.h>
  #include <string.h>
                                             // This header file is required for strcpy()
  struct msgbuf {
                                             // Template of a buffer for msgsnd and msgrcv
 9
       long mtype;
                                             // Message type
10
                                             // Send buffer
      char mtext[80];
11
   } msg;
12
13
   int main(int argc, char **argv) {
14
       int returncode_msgget;
15
16
       // Create message queue or access an existing one
17
       returncode_msgget = msgget(12345, IPC_CREAT | 0600);
18
19
20
       msg.mtvpe = 1;
                                             // Specifiv the message type
21
       strcpy(msg.mtext, "Testnachricht"); // Write the message into the send buffer
22
23
       // Write a message into the message queue
24
       if (msgsnd(returncode_msgget, &msg, strlen(msg.mtext), 0) == -1) {
25
           printf("Unable to write the message into the message queue.\n"):
26
           exit(1);
27
       }
28
```

• The message type (a positive integer value) specifies the user

Result of writing a Message into a Message Queue

• Before...

```
$ ipcs -q
----- Message Queues ------
key msqid owner perms used-bytes messages
0x00003039 98304 bnc 600 0 0
```

• Afterwards...

```
$ ipcs -q
----- Message Queues -------
key msqid owner perms used-bytes messages
0x00003039 98304 bnc 600 80 1
```

Pick a Message from a Message Queue (in C)

```
1 #include <stdlib.h>
 2 #include <sys/types.h>
 3 #include <sys/ipc.h>
 4 #include <stdio.h>
 5 #include <sys/msg.h>
 6 #include <string.h>
                                        // This header file is required for strcpy()
 7 struct msgbuf {
                                        // Template of a buffer for msgsnd and msgrcv
                                        // Message type
       long mtype;
       char mtext[80];
                                        // Send buffer
10
   } msg:
11
12
   int main(int argc, char **argv) {
13
       int returncode_msgget, returncode_msgrcv;
14
       msg receivebuffer:
                                       // Create a receive buffer
15
16
       // Create message queue or access an existing one
17
       returncode msgget = msgget(12345, IPC CREAT | 0600)
18
19
       msg.mtvpe = 1;
                                       // Pick the first message of type 1
       // MSG NOERROR => The message will be truncated when it is too long
20
21
       // IPC NOWAIT => Do not bock the process if no message exists
22
       returncode_msgrcv = msgrcv(returncode_msgget, &msg, sizeof(msg.mtext), msg.mtype,
            MSG NOERROR | IPC NOWAIT):
23
       if (returncode msgrcv < 0) {
24
           printf("Unable to pick a message from the message queue.\n");
25
           perror("msgrcv");
26
       } else {
27
           printf("This message was picked from the message queue: %s\n", msg.mtext);
28
           printf("The received message is %i characters long.\n", returncode_msgrcv);
29
       }
30
```

Erase a Message Queue (in C)

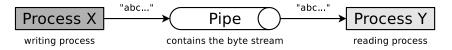
```
1 #include <stdlib.h>
 2 #include <sys/types.h>
 3 #include <sys/ipc.h>
  #include <stdio.h>
  #include <sys/msg.h>
7
   int main(int argc, char **argv) {
 8
       int returncode_msgget;
       int returncode msgctl;
10
11
       // Create message queue or access an existing one
12
       returncode_msgget = msgget(12345, IPC_CREAT | 0600);
13
14
15
       // Erase message queue
16
       returncode msgctl = msgctl(returncode msgget, IPC RMID, 0);
17
       if (returncode_msgctl < 0) {
18
           printf("Unable to erase the message queue with the ID %i.\n", returncode_msgget);
19
           perror("msgctl");
20
           exit(1):
21
       } else {
22
           printf("The message queue with the ID %i has been erased.\n", returncode_msgget);
23
24
       exit(0):
25 }
```

One example of working with message queues in Linux can be found on the website of this course

Pipes (1/2)

An anonymous Pipe...

- is a buffered unidirectional communication channel between 2 processes
 - If communication in both directions shall be possible at the same time, 2 pipes are necessary – one for each communication direction
- operates according to the FIFO principle
- has a limited capacity
 - ullet Pipe = filled \Longrightarrow the writing process gets blocked
 - Pipe = empty ⇒ the reading process gets blocked
- is created with the system call pipe()
 - During this process, the kernel of the operating system creates an Inode
 slide set 6) and 2 file descriptors (handles)
 - Processes access the access identifiers with read() and write() system calls (or standard library functions) for reading data from or writing data into the pipe



Pipes (2/2)

- When child processes are created with fork(), the child processes also inherit access to the file descriptors
- Anonymous pipes allow process communication only between closely related processes
 - Only processes, which are closely related via fork() can communicate with each other via anonymous pipes
 - If the last process, which has access to an anonymous pipe, terminates, the pipe gets erased by the operating system
- Processes, which are not closely related with each other, can communicate via named pipes
 - These pipes can be accessed by using their names
 - They are created in C by: mkfifo("<pathname>",<permissions>)
 - Any process, which knows the name of a pipe, can use the name to access the pipe and communicate with other processes
- The operating system ensures **mutual exclusion**
 - At any time, only a single process can access a pipe

An Anonymous Pipe Example (in C) – Part 1/2

One example of working with named pipes in Linux can be found on the website of this course

```
#include <stdio.h>
   #include <unistd.h>
   #include <stdlib.h>
  void main() {
     int pid_des_Kindes;
 7
     // Zugriffskennungen zum Lesen (testpipe[0]) und Schreiben (testpipe[1]) anlegen
     int testpipe[2]:
10
     // Die Pipe testpipe anlegen
11
     if (pipe(testpipe) < 0) {
12
       printf("Das Anlegen der Pipe ist fehlgeschlagen.\n");
13
       // Programmabbruch
14
       exit(1):
15
     } else {
16
       printf("Die Pipe testpipe wurde angelegt.\n");
17
18
19
     // Einen Kindprozess erzeugen
20
     pid_des_Kindes = fork();
21
22
     // Es kam beim fork zu einem Fehler
23
     if (pid_des_Kindes < 0) {
24
       perror("Es kam bei fork zu einem Fehler!\n");
25
       // Programmabbruch
26
       exit(1);
27
```

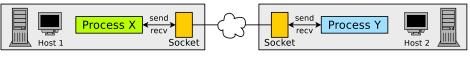
An Anonymous Pipe Example (in C) – Part 2/2

```
28
     // Elternprozess
29
     if (pid_des_Kindes > 0) {
30
       printf("Elternprozess: PID: %i\n", getpid()):
31
       // Lesekanal der Pipe testpipe blockieren
32
       close(testpipe[0]);
33
       char nachricht[] = "Testnachricht";
34
       // Daten in den Schreibkanal der Pipe schreiben
35
       write(testpipe[1], &nachricht, sizeof(nachricht));
36
37
38
     // Kindprozess
39
     if (pid_des_Kindes == 0) {
40
       printf("Kindprozess: PID: %i\n", getpid());
41
       // Schreibkanal der Pipe testpipe blockieren
42
       close(testpipe[1]);
43
       // Einen Empfangspuffer mit 80 Zeichen Kapazität anlegen
44
       char puffer[80]:
45
       // Daten aus dem Lesekanal der Pipe auslesen
46
       read(testpipe[0], puffer, sizeof(puffer));
47
       // Empfangene Daten ausgeben
48
       printf("Empfangene Daten: %s\n", puffer);
49
50
```

```
$ gcc pipe_beispiel.c -o pipe_beispiel
$ ./pipe_beispiel
Die Pipe testpipe wurde angelegt.
Elternprozess: PID: 6363
Kindprozess: PID: 6364
Empfangene Daten: Testnachricht
```

Sockets

- Full duplex-ready alternative to pipes and shared memory
 - Allow interprocess communication in distributed systems
- An user process can request a socket from the operating system and afterwards send and receive data via the socket
 - The operating system maintains all used sockets and the related connection information



- Ports are used for the communication via sockets
 - Port numbers are randomly assigned during connection establishment
 - Port numbers are assigned randomly by the operating system
 - Exceptions are port numbers of well-known applications, such as HTTP (80) SMTP (25), Telnet (23), SSH (22), FTP (21),...
- Sockets can be used in a blocking (synchronous) and non-blocking (asynchronous) way

Different Types of Sockets

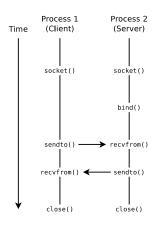
- Connectionless sockets (= datagram sockets)
 - Use the Transport Layer protocol UDP
 - Advantage: Better data rate as with TCP
 - Reason: Lesser overhead for the protocol
 - Drawback: Segments may arrive in wrong sequence or may get lost
- Connection-oriented sockets (= stream sockets)
 - Use the Transport Layer protocol TCP
 - Advantage: Better reliability
 - Segments cannot get lost
 - Segments always arrive in the correct sequence
 - Drawback: Lower data rate as with UDP
 - Reason: More overhead for the protocol

Using Sockets

- Almost all major operating systems support sockets
 - Advantage: Better portability of applications
- Functions for communication via sockets:
 - Creating a Socket: socket()
 - Binding a socket to a port number and making it ready to receive data: bind(), listen(), accept() and connect()
 - Sending/receiving messages via the socket: send(), sendto(), recv() and recvfrom()
 - Closing eines Socket: shutdown() or close()

Overview of the sockets in Linux/UNIX: netstat -n or lsof | grep socket

Connection-less Communication via Sockets - UDP



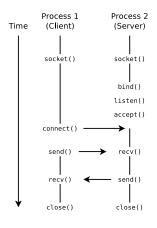
Client

- Create socket (socket)
- Send (sendto) and receive data (recvfrom)
- Close socket (close)

Server

- Create socket (socket)
- Bind socket to a port (bind)
- Send (sendto) and receive data (recvfrom)
- Close socket (close)

Connection-oriented Communication via Sockets – TCP



Client

- Create socket (socket)
- Connect client with server socket (connect)
- Send (send) and receive data (recv)
- Close socket (close)

Server

- Create socket (socket)
- Bind socket to a port (bind)
- Make socket ready to receive (listen)
 - Set up a queue for connections with clients
- Server accepts connections (accept)
- Send (send) and receive data (recv)
- Close socket (close)

Create a Socket: socket

int socket(int domain, int type, int protocol);

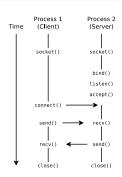
- A call of socket() returns an integer value
 - The value is called socket descriptor (socket file descriptor)
- domain: Specifies the protocol family
 - PF_UNIX: Local interprocess communication in Linux/UNIX
 - PF_INET: IPv4
 - PF_INET6: IPv6
- type: Specifies the type of the socket (and thus the protocol):
 - SOCK_STREAM: Stream socket (TCP)
 - SOCK_DGRAM: Datagram socket (UDP)
 - SOCK_RAW: RAW socket (IP)
- In most cases the protocol parameter is set to value zero
- Create a socket with socket():

```
1 sd = socket(PF_INET, SOCK_STREAM, 0);
2    if (sd < 0) {
3        perror("The socket could not be created");
4        return 1;
5    }</pre>
```

Bind Address and Port Number: bind

int bind(int sd, struct sockaddr *address, int addrlen);

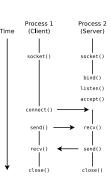
- bind() binds the newly created socket (sd) to the address (address) of the server
 - sd is the socket descriptor from the previous call of socket.()
 - address is a data structure, which contains the IP address of the server and a port number
 - addrlen is the length of the data structure, which contains the IP address and port number



Make a Server ready to receive Data: listen

int listen(int sd, int backlog);

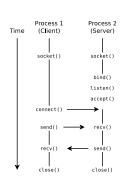
- listen() specifies how many connection requests can be buffered by the socket
 - If the listen() queue has no more free capacity, further connection requests from clients are rejected
 - sd is the socket descriptor from the previous call of socket()
 - backlog contains the number of possible connection requests, which can be stored in the gueue
 - Default value: 5
 - A server for datagrams (UDP) does not need to call listen(), because it does not establish connections to clients



Accept a Connection Request: accept

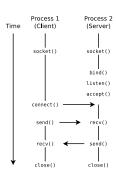
int accept(int sd, struct sockaddr *address, int *addrlen);

- accept() is used by the server to fetch the first connection request from the queue
- The return value is the socket descriptor of the new socket
- If the queue contains no connection requests, the process is blocked until a connection request arrives
- address contains the address of the client
- After a connection request was accepted with accept(), the connection with the client is established



Establish a Connection by the Client

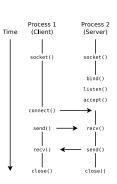
- Via connect(), the client tries to establish a connection to a server socket
- The client must know the address (hostname and port number) of the server
- sd is the socket descriptor
- address contains the address of the server.
- addrlen is the length of the data structure, which contains the address of the server



Connection-oriented Exchange of Data: send and recv

```
int send(int sd, char *buffer, int nbytes, int flags);
int recv(int sd, char *buffer, int nbytes, int flags);
```

- Data are exchanged via send() and recv() over an existing connection
- send() sends a message (buffer) via the socket (sd)
- recv() receives a message from the socket sd and stores it in the buffer (buffer)
- sd is the socket descriptor
- buffer contains the data to be sent or received
- nbytes specifies the number of bytes in the buffer
- The value of flags is usually zero



Connection-oriented Exchange of Data: read and write

```
int read(int sd, char *buffer, int nbytes);
int write(int sd, char *buffer, int nbytes);
```

- In UNIX it is in normal case also possible to use read() and write()
 for receiving and sending data via a socket
 - "Normal case" means, that read() and write() can be used, when the parameter flags of send() and recv() contains value zero
- The following calls have the same result

```
send(socket, "Hello World", 11,0);
write(socket, "Hello World", 11);
```

Connection-less Exchange of Data: sendto and recvfrom

- If a process knows the address of the socket (host and port), to which it should send data, it uses sendto()
- sendto() always transmits together with the data the local address
- sd is the socket descriptor
- buffer contains the data to be sent or received
- nbytes specifies the number of bytes in the buffer
- to contains the address of the receiver
- from contains the address of the sender
- addrlen is the length of the data structure, which contains the address

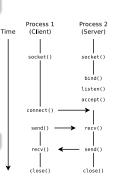
Close a Socket: close

int shutdown(int sd, int how);

- shutdown() closes a bidirectional socket connection
- The parameter how specifies whether no more data will be received (how=0), no more data will be send (how=1), or both (how=2)

int close(int sd):

 If close() is used instead of shutdown(), this corresponds to a shutdown(sd,2)



Sockets via UDP - Example (Server)

```
Process 1
                                                                                          Process 2
                                                                            (Client)
                                                                                          (Server)
                                                                    Time
1 #!/usr/bin/env python
  # Server: Receives a message via UDP
                                        # Import module socket
   import socket
                                                                            socket()
                                                                                          socket()
  # For all interfaces of the host
   HOST = ''
                                        # '' = all interfaces
   PORT = 50000
                                        # Port number of server
                                                                                           bind()
  # Create socket and return socket deskriptor
  sd = socket.socket(socket.AF INET. socket.SOCK DGRAM)
12
13
  try:
                                                                            sendto() ------ recvfrom()
    sd.bind(HOST, PORT)
                                      # Bind socket to port
15
   while True:
     data = sd.recvfrom(1024)
16
                                      # Receive data
                                                                           recvfrom() ← sendto()
     print 'Received:', repr(data) # Print out received data
   finally:
19
     sd.close()
                                       # Close socket
                                                                            close()
                                                                                           close()
```

Sockets via UDP – Example (Client)

```
(Client)
                                                                                             (Server)
                                                                      Time
  #!/usr/bin/env python
   # Client: Sends a message via UDP
                                                                                             socket()
                                                                              socket()
   import socket
                                         # Import module socket
   HOST = 'localhost'
                                         # Hostname of Server
   PORT = 50000
                                         # Port number of Server
                                                                                              bind()
   MESSAGE = 'Hello World'
                                         # Message
  # Create socket and return socket deskriptor
   sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                                                                              sendto() ------ recvfrom()
12
  sd.sendto(MESSAGE, (HOST, PORT)) # Send message to socket
14
                                                                             recvfrom() 	←
                                                                                          — sendto()
  sd.close()
                                         # Close socket
                                                                               close()
                                                                                             close()
```

Process 2

Process 1

Sockets via TCP - Example (Server)

```
1 #!/usr/bin/env python
  # Echo Server via TCP
                              # Import module socket
  import socket
  HOST = ''
                               # '' = all interfaces
   PORT = 50007
                               # Port number of server
   # Create socket and return socket deskriptor
  sd = socket.socket(socket.AF INET, socket.SOCK STREAM)
11
   sd.bind(HOST, PORT)
                               # Bind socket to port
13
  sd.listen(1)
                               # Make socket ready to receive
15
                               # Max. number of connections = 1
16
  conn, addr = sd.accept() # Socket accepts connections
18
  print 'Connected by', addr
   while 1:
                               # Infinite loop
21
     data = conn.recv(1024) # Receive data
    if not data: break
                               # Break infinite loop
23
       conn.send(data)
                               # Send back received data
24
25 conn.close()
                               # Close socket
```

```
Process 1
                           Process 2
Time
          (Client)
                            (Server)
         socket()
                            socket()
                             bind()
                            listen()
                            accept()
         connect()
          send()
                             recv()
          recv()
                             send()
          close()
                             close()
```

Sockets via TCP – Example (Client)

```
Process 1
                                                                                              Process 2
  #!/usr/bin/env pvthon
                                                                        Time
                                                                                (Client)
                                                                                              (Server)
   # Echo Client via UDP
 3
   import socket
                                     # Import module socket
                                                                                socket()
                                                                                               socket()
   HOST = 'localhost'
                                     # Hostname of Server
   PORT = 50007
                                     # Port number of server
                                                                                                bind()
   # Create socket and return socket deskriptor
                                                                                               listen()
   sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                                                                                               accept()
11
   sd.connect(HOST, PORT)
                                     # Connect with server socket
                                                                               connect()
13
   sd.send('Hello, world')
                                     # Send data
                                                                                send()
                                                                                                recv()
15
   data = sd.recv(1024)
                                     # Receive data
17
                                                                                recv()
                                                                                                send()
   sd.close()
                                     # Close socket
19
   print 'Empfangen:', repr(data) # Print out received data
                                                                                close()
                                                                                               close()
```

Blocking and non-blocking Sockets

- If a socket is created, it per default in **blocking mode**
 - All method calls wait until the operation, they initiated, was carried out
 - e.g. a call of recv() blocks the process until data is received and can be read from the internal buffer of the socket
- The method setblocking() modifies the mode of a socket
 - ullet sd.setblocking(0) \Longrightarrow switches into non-blocking mode
 - ullet sd.setblocking(1) \Longrightarrow switches into blocking mode
- It is possible to switch between the modes at any time during process execution
 - e.g. the method connect() could be used in blocking mode and afterwards the method read() in non-blocking mode

Source: Peter Kaiser, Johannes Ernesti. Python – Das umfassende Handbuch. Galileo (2008)

- recv() and recvfrom()
 - The method return data only, when they are already stored in the buffer
 - If the buffer does not contain any data, the method throws an exception and the program execution continues
- send() and sendto()
 - The methods send the specified data only, when they can be written directly in the send buffer
 - If the buffer has no more free capacity, the method throws an exception and the program execution continues
- connect()
 - The method sends a connection request to the destination socket and does not wait until this connection is established
 - If connect() is called, while the connection request is still in progress, an exception is thrown
 - By calling connect() several times, it can be checked, whether the operation is still carried out

	Shared Memory	Message Queues	(anon./named) Pipes	Sockets
Sort of communication	Memory-based	Message-based	Message-based	Message-based
Bidirectional	yes	no	no	yes
Platform independent	no	no	no	yes
Processes must be related with each other	no	no	for anon. pipes	no
Communication over computer boundaries	no	no	no	yes
Remain intact without a bound process	yes	yes	no	no
Automatic synchronization	no	yes	yes	yes

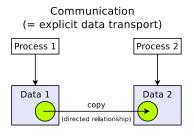
- Advantages of message-based communication versus memory-based communication:
 - The operating system takes care about the synchronization of accesses ⇒ comfortable because the user processes do not need to take care about the synchronization
 - Can be used in distributed systems without a shared memory
 - Better portability of applications

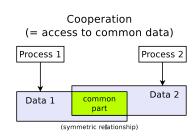
Storage can be integrated via network connections

- This allows memory-based communication between processes on different independent systems
- The problem of synchronizing the accesses also exists here

Cooperation

- Cooperation
 - Semaphor
 - Mutex





Semaphore

- In order to protect (lock) critical sections, not only the already discussed locks can be used, but also semaphores
- 1965: Published by Edsger W. Dijkstra
- A semaphore is a counter lock S with operations P(S) and V(S)
 - **V** comes from the dutch *verhogen* = raise
 - P comes from the dutch proberen = try (to reduce)
- The access operations are atomic ⇒ can not be interrupted (indivisible)
- May allow multiple processes accessing the critical section
 - In contrast to semaphores, can locks (\Longrightarrow slide 14) only be used to allow a single process entering the critical section at the same time

Cooperating sequential processes. Edsger W. Dijkstra (1965)

https://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF

Semaphore Access Operations (1/3)

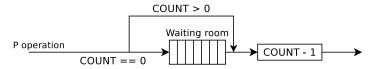
A Semaphore consists of 2 Data Structures

- COUNT: An integer, non-negative counter variable.
 Specifies how many processes can pass the semaphore now without getting blocked
- A waiting room for the processes, which wait until they are allowed to pass the semaphore.
 The processes are in blocked state until they are transferred into ready state by the operating system when the semaphore allows to access the critical section.
- Initialization: First, a new semaphore is created or an existing one is opened
 - For a new semaphore, the count variable is initialized at the beginning with a non-negative initial value

Semaphore Access Operations (2/3)

Image Source: Carsten Vogt

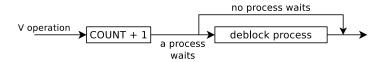
- **P operation** (*reduce*): It checks the value of the counter variable
 - If the value is 0, the process becomes blocked
 - If the value > 0, it is reduced by 1



Semaphore Access Operations (3/3)

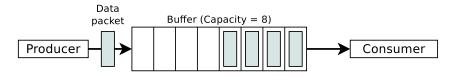
Image Source: Carsten Vogt

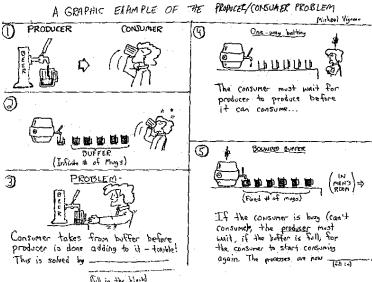
- **V** operation (raise): It first increases the counter variable by value 1
 - If processes are in the waiting room, one process gets deblocked
 - The process, which just got deblocked, continues its P operation and first reduces the counter variable



Producer/Consumer Example (1/3)

- A producer sends data to a consumer
- A buffer with limited capacity is used to minimize the waiting times of the consumer
- Data is placed into the buffer by the producer and the consumer removes data from the buffer
- Mutual exclusion is mandatory in order to avoid inconsistencies
- Buffer = full ⇒ producer must be blocked
- Buffer = empty ⇒ consumer must be blocked





Source: http://www.ccs.neu.edu/home/kenb/synchronize.html

Producer/Consumer Example (2/3)

- 3 semaphores are used to synchronize access to the buffer
 - empty
 - filled
 - mutex
- The semaphores filled and empty are used in opposite to each other
 - empty counts the number of empty locations in the buffer and its value is reduced by the producer (P operation) and raised by the consumer (V operation)
 - $\bullet \ \ \mathsf{empty} = 0 \Longrightarrow \mathsf{puffer} \ \mathsf{is} \ \mathsf{completely} \ \mathsf{filled} \Longrightarrow \mathsf{producer} \ \mathsf{is} \ \mathsf{blocked}$
 - filled counts the number of data packets (occupied locations) in the buffer and its value is raised by the producer (V operation) and reduced by the consumer (P operation)
 - filled = $0 \Longrightarrow \text{puffer is empty} \Longrightarrow \text{consumer is blocked}$
- The semaphore mutex is used to ensure for the mutual exclusion

Binary Semaphores

- Binary semaphores are initialized with value 1 and ensure that 2 or more processes cannot simultaneously enter their
 critical sections
- Example: The semaphore mutex from the producer/consumer example

Producer/Consumer Example (3/3)

```
1 typedef int semaphore;
                                  // semaphores are of type integer
  semaphore filled = 0;
                                  // counts the number of occupied locations in the buffer
  semaphore empty = 8:
                                  // counts the number of empty locations in the buffer
   semaphore mutex = 1:
                                  // controls access to the critial sections
   void producer (void) {
     int data:
9
     while (TRUE) {
                                  // infinite loop
10
       createDatapacket(data);
                                  // create data packet
11
       P(empty);
                                  // decrement the empty locations counter
12
       P(mutex);
                                  // enter the critical section
13
     insertDatapacket(data);
                                  // write data packet into the buffer
14
     V(mutex):
                                  // leave the critical section
15
       V(filled);
                                  // increment the occupied locations counter
16
17 }
18
   void consumer (void) {
20
     int data:
21
22
     while (TRUE) {
                                  // infinite loop
23
       P(filled):
                                  // decrement the occupied locations counter
24
    P(mutex);
                                  // enter the critical section
25
    removeDatapacket(data);
                                  // pick data packet from the buffer
26
     V(mutex);
                                  // leave the critical section
27
       V(empty);
                                  // increment the empty locations counter
28
       consumeDatapacket(data);
                                  // consume data packet
29
30 }
```

Semaphore Example: PingPong

```
1 // Initialization of semaphores
  s init (Sema Ping, 1);
  s init (Sema Pong, 0);
  task Ping is
  begin
7
      loop
           P(Sema Ping);
           print("Ping");
           V(Sema_Pong);
10
11
      end loop;
12
  end Ping;
13
  task Pong is
  begin
      loop
16
           P(Sema Pong):
17
           print("Pong, ");
18
           V(Sema Ping);
19
       end loop;
21 end Pong;
```

 The two endless-running processes and Ping print out continuously PingPong, PingPong, PingPong,...

Semaphore Example: 3 Runners (1/3)

- 3 runners should run a certain distance one behind the other
 - The 2nd runner is not allowed to start before the 1nd runner finished his run
 - The 3th runner is not allowed to start before the 2nd runner finished his run
- Is this solution correct?

```
1 // Initialization of semaphores
2 s_init (Sema, 0);
  task First is
             \langle r_{11}n \rangle
             V(Sema);
   task Second is
             P(Sema);
             \langle riin \rangle
10
             V(Sema);
11
12
  task Third is
             P(Sema);
14
             < riin >
15
```

Semaphore Example: 3 Runners (2/3)

- The solution is not correct!
- 2 sequence conditions exist:
 - Runner 1 prior runner 2
 - Runner 2 prior runner 3
- Both sequence conditions use the same semaphore
 - It can happen that runner 3 prior runner 2 decreases with its P operation the semaphore by value 1
- How could a correct solution look like?

```
1 // Initialization of semaphores
2 s_init (Sema, 0);
  task First is
           < riin >
           V(Sema);
  task Second is
           P(Sema);
10
           < riin >
           V(Sema);
11
12
  task Third is
14
           P(Sema);
           < riin >
```

Semaphore Example: 3 Runners (3/3)

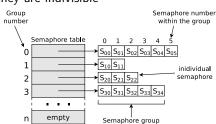
- Possible solution:
 - Introduce a second semaphore
 - The second semaphore is also initialized with value 0
 - Runner 2 increases the second semaphore with its V operation
 - Runner 3 decreases the second semaphore with its P operation

```
1 // Initialization of semaphores
2 s_init (Sema1, 0);
  s_init (Sema2, 0);
  task First is
           < run >
           V(Sema1):
  task Second is
           P(Sema1);
10
           < run >
11
           V(Sema2):
12
13
  task Third is
           P(Sema2):
15
           < run >
16
```

Semaphores in Linux

Image Source: Carsten Vogt

- The semaphore concept of Linux differs from the Dijkstra concept
 - The counter variable can be incremented or decremented with a P or V operation by more than value 1
 - Multiple access operations on different semaphores can be carried out in an atomic way, which means that they are indivisible
- Linux systems maintain a semaphore table, which contains references to arrays of semaphores
 - Individual semaphores are addressed using the table index and the position in the group



Linux/UNIX operating systems provide 3 system calls for working with semaphores

- semget(): Create new semaphore or a group of semaphores or open an existing semaphore
- semct1(): Request or modify the value of an existing semaphore or of a semaphore group or erase a semaphore
- semop(): Carry out P and V operations on semaphores
- Information about existing semaphores provides the command ipcs

Mutexes

- Semaphores offer the feature of counting
- However, if this feature is not required, a simplified semaphore version, the mutex can be used instead
 - Mutexes (derived from Mutual Exclusion) are used to protect critical sections, which are allowed to be accessed by only a single process at any given moment
 - Mutexes can only have 2 states: occupied and not occupied
 - Mutexes have the same functionality as binary semaphores

2 functions for accessing a Mutex exist

```
\begin{array}{ll} \mathtt{mutex\_lock} & \Longrightarrow & \mathsf{corresponds} \ \mathsf{to} \ \mathsf{the} \ \mathsf{P} \ \mathsf{operation} \\ \mathtt{mutex\_unlock} & \Longrightarrow & \mathsf{corresponds} \ \mathsf{to} \ \mathsf{the} \ \mathsf{V} \ \mathsf{operation} \end{array}
```

- If a process wants to access a critical section, it calls mutex_lock
 - If the critical section is locked, the process gets locked, until the process in the critical section is finished and calls mutex unlock
 - If the critical section is **not locked**, the process can enter it

Monitor and erase IPC Objects

- Information about existing shared memory segments provides the command ipcs
- The easiest way to erase semaphores, shared memory segments and message queues from the command line is the command ipcrm

```
ipcrm [-m shmid] [-q msqid] [-s semid]
        [-M shmkey] [-Q msgkey] [-S semkey]
```

- Or alternatively just...
 - ipcrm shm SharedMemoryID
 - ipcrm sem SemaphorID
 - ipcrm msg MessageQueueID