

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 transmit information to others?
 - How can multiple processes access shared resources?

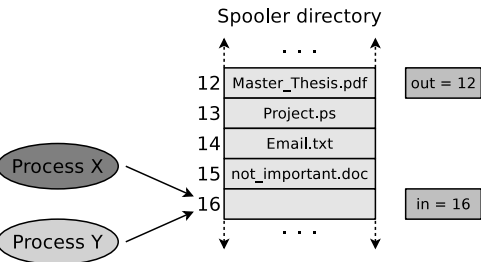
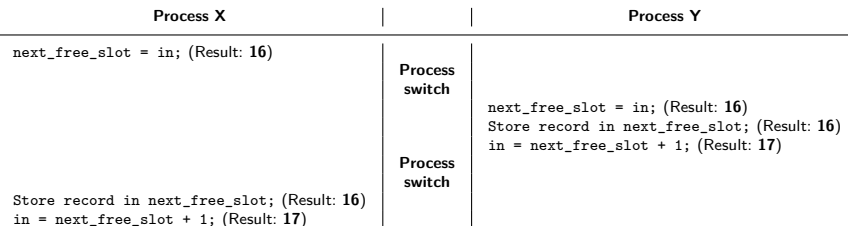
Question: What is the situation here with threads?

- 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 (\implies common data), the operating system must provide **mutual exclusion**

Critical Sections – Example: Print Spooler



- The spooling directory is consistent
 - But the entry of **process Y** was overwritten by **process X** and got lost
- 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
(\implies slide 55)

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.
Frequently shown picture in this context.
(author and license: unknown)

An Investigation of the Therac-25 Accidents. Nancy Leveson, Clark S. Turner. IEEE Computer, Vol. 26, No. 7, July 1993, S.18-41
http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html

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

- A race condition („Texas-Bug“) led to incorrect settings of the device and consequently to increased radiation doses.
 - The control process did not synchronize correctly with the user interface process
 - The error occurred only during a quick input correction (time window: 8 seconds) by the user
 - During testing the error did not occur because experience (routine) was required to operate the device this fast

The Worst Computer Bugs in History: Race conditions in Therac-25:

<https://www.bugsnag.com/blog/bug-day-race-condition-therac-25>

„Once the data entry phase was marked complete, the magnet setting phase began. However, if a specific sequence of edits was applied in the Data Entry phase during the 8 second magnet setting phase, the setting was not applied to the machine hardware, due to the value of the completion variable. The UI would then display the wrong mode to the user, who would confirm the potentially lethal treatment.“

Other interesting sources

https://www.dssz.informatik.tu-cottbus.de/information/slides_studis/ss2009/mehner_RisikoComputer_zs09.pdf

Killer Bug. Therac-25: Quick-and-Dirty: <https://www.viva64.com/en/b/0438/>

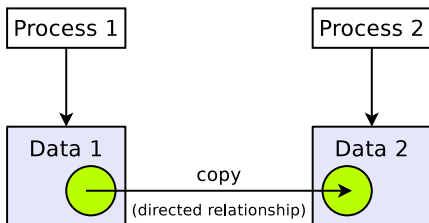
Killed by a machine: The Therac-25: <https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/>

Communication vs. Cooperation

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

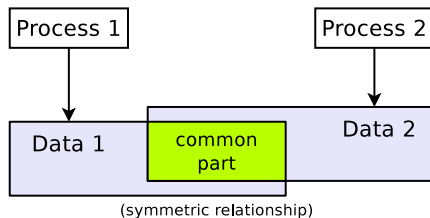
Communication

(= explicit data transport)



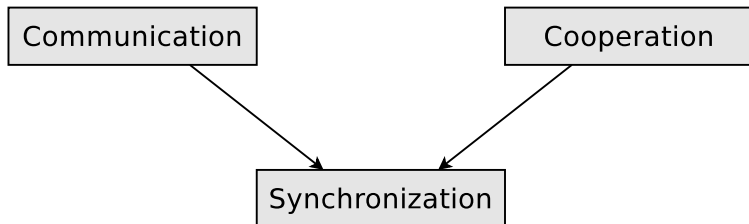
Cooperation

(= access to common data)

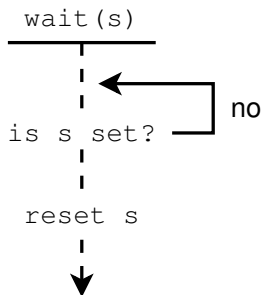
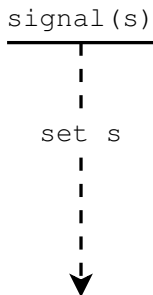


Forms of Interaction

- 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**



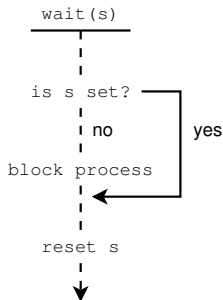
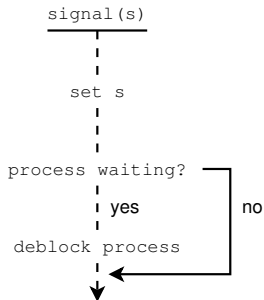
Most Simple Form of Signaling (Busy Waiting)



- 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 **X**
 - Advantage: No CPU resources are wasted
 - Drawback: Only a single process can wait
 - In literature, this technique is also called **passive waiting**

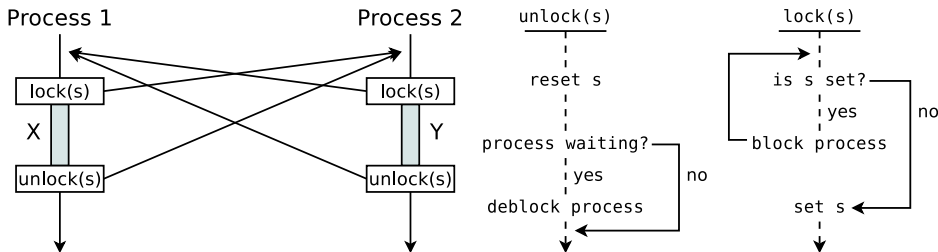


One way to specify in Linux an execution order with passive waiting, is by using the function `sigsuspend`. Thereby a process blocks itself until another process sends it an appropriate signal (usually `SIGUSR1` or `SIGUSR2`) with the command `kill` (or the system call of the same name) and in this way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

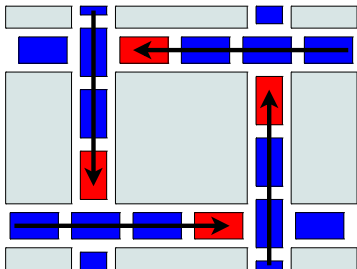
Problems caused by Blocking

• 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

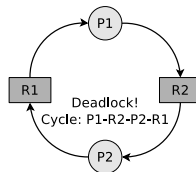
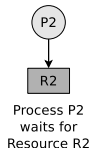
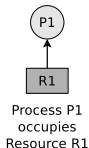
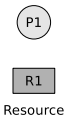


Source: <https://i.redd.it/vvu6v8pxvue11.jpg>
(author and license: unknown)

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

Process



A good description of resource graphs provides the book **Betriebssysteme – Eine Einführung**, Uwe Baumgarten, Hans-Jürgen Sievert, 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 = $(4 \quad 2 \quad 3 \quad 1)$

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

Current allocation matrix = $\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$

- 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 = $(2 \quad 1 \quad 0 \quad 0)$

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

- 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 execution, it deallocates its resources

Available resource vector = $(2 \quad 2 \quad 2 \quad 0)$

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

- 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 execution, it deallocates its resources
- Process 1 is blocked, because no free resources of class 4 exist
- **Process 2 is not blocked**

Available resource vector = $(4 \quad 2 \quad 2 \quad 1)$

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

- **Process 1 is not blocked** \implies no deadlock in this example

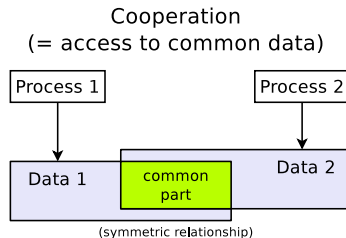
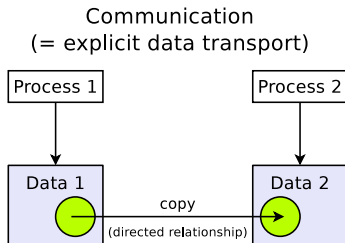
Conclusion about Deadlocks

- 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
- The probability that this happens is low, but $\neq 0$
 - 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)

Communication of Processes

- Communication

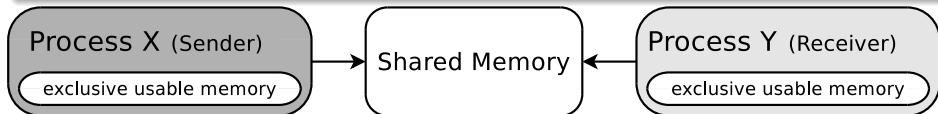
- Shared Memory
- Message Queues
- Pipes
- Sockets



Shared Memory

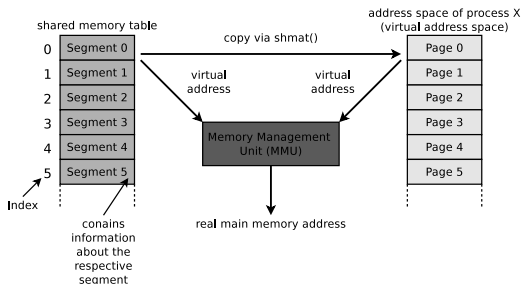
- Interprocess communication via a shared memory is also called **memory-based communication**
- **Shared memory segments** are memory areas, which can be accessed by multiple processes
 - These memory areas are located in the address space of multiple processes
- The processes need to coordinate the 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
 - If access operations are not coordinated carefully \implies 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

When the operating system is rebooted, the shared memory segments and their contents are lost

Working with Shared Memory (System V vs. POSIX)

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
- The command `ipcs` provides information about existing shared memory segments (System V)

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

- Some developers prefer the System V API and Others the POSIX API. . . ʘ_ʘ/

C function calls for for working with POSIX shared memory segments (some defined in the header file `mman.h`)

- `shm_open()`: Create a shared memory segment or access an existing one
- `ftruncate()`: Specify the size of a shared memory segment
- `mmap()`: Attach a shared memory segment to a process
- `munmap()`: Detach a shared memory segment from a process
- `close()`: Close the descriptor of a shared memory segment Speichersegments schließen
- `shm_unlink()`: Erase a Segment
- In Linux, POSIX shared memory segments can be found in the `/dev/shm` directory

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

Create a (System V) Shared Memory Segment (in C)

```
1 #include <sys/ipc.h>
2 #include <sys/shm.h>
3 #include <stdio.h>
4 #define MAXMEMSIZE 20
5
6 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");
18    } else {
19        printf("The shared memory segment has been created.\n");
20    }
21 }
```

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

$ printf "%d\n" 0x00003039           # Convert from hexadecimal to decimal
12345
```

Attach a (System V) Shared Memory Segment (in C)

```
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     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 -----
```

key	shmid	owner	perms	bytes	nattch	status
0x00003039	56393780	bnc	600	20	1	

Write into a (System V) Segment and read from it (in C)

```
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget, returncode_shmldt, 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 (System V) Shared Memory Segment (in C)

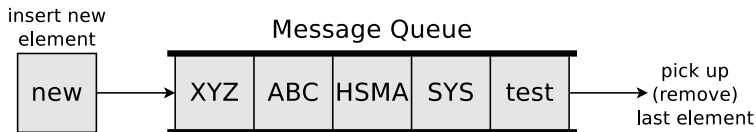
```
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget;
10    int returncode_shmdt;
11    char *sharedmempointer;
12
13    // Create shared 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 (System V) Shared Memory Segment (in C)

```
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     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    // Erase 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 pick 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 (System V)

- `msgget()`: Create a message queue or access an existing one
- `msgsnd()`: Write message into message queues (\Rightarrow send operation)
- `msgrcv()`: Read message from message queues (\Rightarrow receive operation)
- `msgctl()`: Request status information (e.g. privileges) of a message queue, modify or erase it
- The command `ipcs` provides information about existing System V message queues

Create (System V) Message Queues (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
7 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) {
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",
19              returncode_msgget);
20    }
```

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

$ printf "%d\n" 0x00003039      # Convert from hexadecimal to decimal
12345
```

Write Messages into (System V) Message Queues (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
8 struct msgbuf {               // Template of a buffer for msgsnd and msgrcv
9     long mtype;               // Message type
10    char mtext[80];           // Send buffer
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.mtype = 1;             // Specify 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 (System V) 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
8     long mtype;               // Message type
9     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.mtype = 1;             // Pick the first message of type 1
20     // MSG_NOERROR => The message will be truncated when it is too long
21     // IPC_NOWAIT  => Do not block the process if no message exists
22     returncode_msgrcv = msgrcv(returncode_msgget, &msg, sizeof(msg.mtext), msg.mtype,
23                               MSG_NOERROR | IPC_NOWAIT);
24     if (returncode_msgrcv < 0) {
25         printf("Unable to pick a message from the message queue.\n");
26         perror("msgrcv");
27     } else {
28         printf("This message was picked from the message queue: %s\n", msg.mtext);
29         printf("The received message is %i characters long.\n", returncode_msgrcv);
30     }
31 }
```

Erase a (System V) 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
7 int main(int argc, char **argv) {
8     int returncode_msgget;
9     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 System V message queues in Linux can be found on the website of this course

Message Queues in Linux (System V vs. POSIX)

- The functions described so far for working with message queues are part of the **System V** interface
- Some developers prefer the System V API and Others the POSIX API... 🙄

C function calls for POSIX message queue specified in the header file `mqqueue.h`

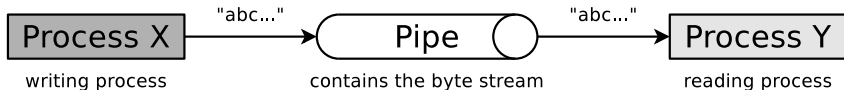
- `mq_open()`: Create a message queue or access an existing one
- `mq_send()`: Write (send) a message into a message queue. Blocking function
- `mq_timedsend()`: Write (send) a message into a message queue. Blocking instruction with a timeout
- `mq_receive()`: Read (receive) a message from a message queue. Blocking instruction
- `mq_timedreceive()`: Read (receive) a message from a message queue. Blocking instruction with a timeout
- `mq_getattr()`: Request the attributes of a message queue. These are: Number of messages in the queue, maximum message size, maximum number of messages...
- `mq_setattr()`: Modify the attributes of a message queue
- `mq_notify()`: Notify the process as soon as a message is available
- `mq_close()`: Close a message queue
- `mq_remove()`: Erase a message queue
- POSIX message queues are created In Linux in the folder `/dev/mqueue`

One example of working with POSIX 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
 - Pipe = filled \implies the writing process gets blocked
 - Pipe = empty \implies the reading process gets blocked
- is created with the system call `pipe()`
 - During this process, the kernel of the operating system creates an Inode (\implies 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

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int pid_des_Kindes;
7     // Zugriffskennungen zum Lesen (testpipe[0]) und Schreiben (testpipe[1]) anlegen
8     int testpipe[2];
9
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    }
```

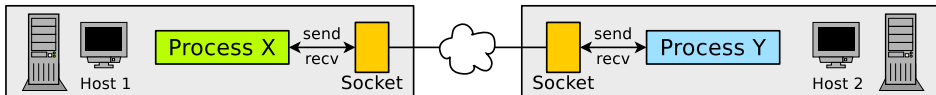
An Anonymous Pipe Example (in C) – Part 2/2

```
28 // Elternprozess
29 if (pid_des_Kindes > 0) {
30     printf("Elternprozess: PID: %i\n", getpid());
31     // Lese kanal 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 Lese kanal 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

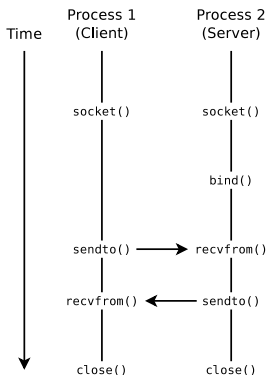
Using Sockets

- 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`

Examples of Interprocess communication via sockets (TCP and UDP) in Linux can be found on the website of this course

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`)

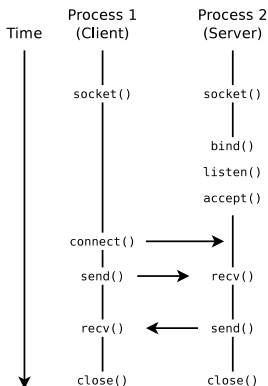
Connection-oriented Communication via Sockets – TCP

• Client

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

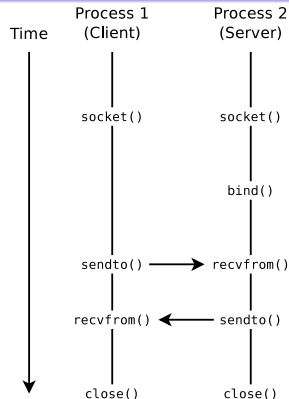
• Server

- Create socket (`socket`)
- Bind socket to a port (`bind`)
- Make socket ready to receive (`listen`)
 - Set up a queue for connection requests.
Specifies the number of connection requests, which can be stored in the queue
- Server accepts connections (`accept`)
 - Fetch the first connection request from the queue
- Send (`send`) and receive data (`recv`)
- Close socket (`close`)



Sockets via UDP – Example (Server)

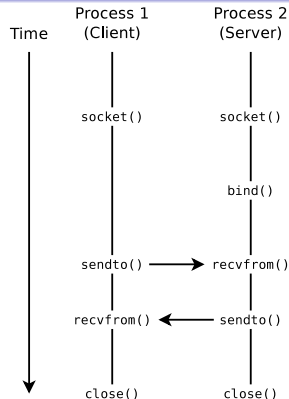
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[]) {
10     int sd, adresse_laenge;
11     char puffer[1024] = { 0 };
12     struct sockaddr_in adresse, client_adresse;
13     memset(&adresse, 0, sizeof(adresse));
14     memset(&client_adresse, 0, sizeof(client_adresse));
15     adresse.sin_family = AF_INET;
16     adresse.sin_addr.s_addr = INADDR_ANY;
17     adresse.sin_port = htons(atoi(argv[1]));
18
19     sd = socket(AF_INET, SOCK_DGRAM, 0);
20     bind(sd, (struct sockaddr *) &adresse, sizeof(adresse));
21     adresse_laenge = sizeof(client_adresse);
22     recvfrom(sd, (char *)puffer, sizeof(puffer), 0,
23             (struct sockaddr *) &client_adresse, &adresse_laenge);
24     printf("Empfangene Nachricht: %s\n", puffer);
25     char antwort[]="Server: Nachricht empfangen.\n";
26     sendto(sd, (const char *)antwort, sizeof(antwort), 0,
27           (struct sockaddr *) &client_adresse, adresse_laenge);
28     close(sd);
29     exit(0);
30 }
```



```
$ gcc udp_server.c -o udp_server
$ ./udp_server 50002
```

Sockets via UDP – Example (Client)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[]) {
10     int sd, adresse_laenge;
11     char puffer[1024] = { 0 };
12     struct sockaddr_in adresse;
13     memset(&adresse, 0, sizeof(adresse));
14     adresse.sin_family = AF_INET;
15     adresse.sin_port = htons(atoi(argv[2]));
16     adresse.sin_addr.s_addr = inet_addr(argv[1]);
17
18     sd = socket(AF_INET, SOCK_DGRAM, 0);
19     printf("Bitte Nachricht eingeben: ");
20     fgets(puffer, sizeof(puffer), stdin);
21     adresse_laenge = sizeof(adresse);
22     sendto(sd, (const char *)puffer, strlen(puffer), 0,
23           (struct sockaddr *) &adresse, adresse_laenge);
24     memset(puffer, 0, sizeof(puffer));
25     recvfrom(sd, (char *)puffer, sizeof(puffer), 0,
26            (struct sockaddr *) &adresse, &adresse_laenge);
27     printf("%s\n", puffer);
28     close(sd);
29     exit(0);
30 }
```



```
$ gcc udp_client.c -o udp_client
$ ./udp_client 127.0.0.1 50002
Bitte Nachricht eingeben: Test
Server: Nachricht empfangen.
```

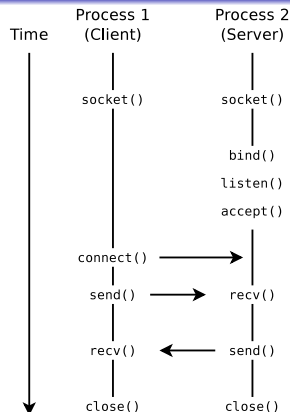
```
$ ./udp_server 50002
Empfangene Nachricht: Test
```

Sockets via TCP – Example (Server)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[]) {
10     int sd, fd, adresse_laenge;
11     char puffer[1024] = { 0 };
12     struct sockaddr_in adresse;
13     memset(&adresse, 0, sizeof(adresse));
14     adresse.sin_family = AF_INET;
15     adresse.sin_addr.s_addr = INADDR_ANY;
16     adresse.sin_port = htons(atoi(argv[1]));
17
18     sd = socket(AF_INET, SOCK_STREAM, 0);
19     bind(sd, (struct sockaddr *) &adresse, sizeof(adresse));
20     listen(sd, 5);
21     adresse_laenge = sizeof(adresse);
22     fd = accept(sd, (struct sockaddr *) &adresse, &adresse_laenge);
23     read(fd, puffer, sizeof(puffer));
24     printf("Empfangene Nachricht: %s\n", puffer);
25     char antwort[] = "Server: Nachricht empfangen.\n";
26     write(fd, antwort, sizeof(antwort));
27     close(fd);
28     close(sd);
29     exit(0);
30 }

```



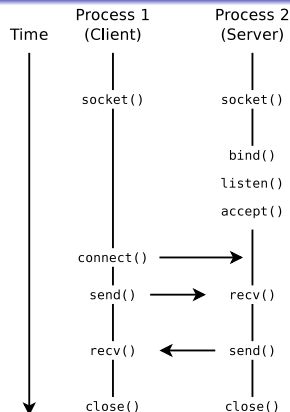
```
$ gcc tcp_server.c -o tcp_server
$ ./tcp_server 50003
```

Sockets via TCP – Example (Client)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[]) {
10     int sd;
11     char puffer[1024] = { 0 };
12     struct sockaddr_in adresse;
13     memset(&adresse, 0, sizeof(adresse));
14     adresse.sin_family = AF_INET;
15     adresse.sin_port = htons(atoi(argv[2]));
16     adresse.sin_addr.s_addr = inet_addr(argv[1]);
17
18     sd = socket(AF_INET, SOCK_STREAM, 0);
19     connect(sd, (struct sockaddr *) &adresse, sizeof(adresse));
20
21     printf("Bitte Nachricht eingeben: ");
22     fgets(puffer, sizeof(puffer), stdin);
23     write(sd, puffer, strlen(puffer));
24     memset(puffer, 0, sizeof(puffer));
25     read(sd, puffer, sizeof(puffer));
26     printf("%s\n", puffer);
27
28     close(sd);
29     exit(0);
30 }

```



```
$ gcc tcp_client.c -o tcp_client
$ ./tcp_client 127.0.0.1 50003
Bitte Nachricht eingeben: Test
Server: Nachricht empfangen.
```

```
$ ./tcp_server 50003
Empfangene Nachricht: Test
```

Comparison of Communication Systems

	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
 - Can be used in distributed systems without a shared memory
 - Better portability of applications

Storage can be integrated via network connections

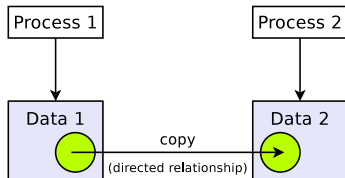
- This allows memory-based communication between processes on different independent systems
- The problem of synchronizing the accesses also exists here

Cooperation

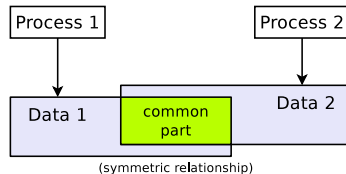
- Cooperation

- Semaphore
- Mutex

Communication
(= explicit data transport)



Cooperation
(= access to common data)



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 counter variable is initialized at the beginning with a non-negative initial value

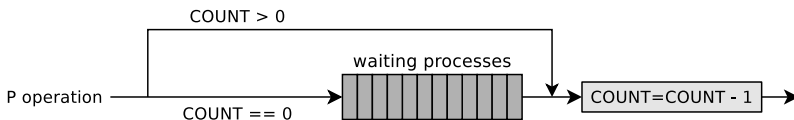
```
1 // apply the INIT operation on semaphore SEM
2 SEM.INIT(unsigned int init_value) {
3
4     // initialize the variable COUNT of Semaphore SEM
5     // with a non-negative initial value
6     SEM.COUNT = init_value;
7 }
```


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

```
1 SEM.P() {  
2   // if the counter variable = 0, the process becomes blocked  
3   if (SEM.COUNT == 0)  
4     < block >  
5  
6   // if the counter variable is > 0, the counter variable  
7   // is decremented immediately by 1  
8   SEM.COUNT = SEM.COUNT - 1;  
9 }
```

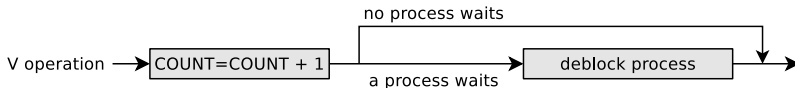


Semaphore Access Operations (3/3)

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

```
1 SEM.V() {  
2   // counter variable = counter variable + 1  
3   SEM.COUNT = SEM.COUNT + 1;  
4  
5   // if processes are in the waiting room, one gets deblocked  
6   if ( < SEM waiting room is not empty > )  
7       < deblock a waiting process >  
8 }
```



Simple Semaphore Example (in C) – Part 1/5

This program creates a child process. The parent process and the child process both try to print characters in the command line interface (critical section). Each process may print only one character at a time. Two semaphores are used to ensure mutual exclusion

```
1 #include <stdio.h>    // für printf
2 #include <stdlib.h>    // für exit
3 #include <unistd.h>    // für read, write, close
4 #include <sys/wait.h>  // für wait
5 #include <sys/sem.h>   // für semget, semctl, semop
6
7 void main() {
8     int pid_des_kindes;
9     int sem_key1=12345;
10    int sem_key2=54321;
11    int returncode_semget1, returncode_semget2, returncode_semctl;
12    int output;
13
14    setbuf(stdout, NULL); // Das Puffern Standardausgabe (stdout) unterbinden
15
16    // Neue Semaphorgruppe 12345 mit einer Semaphore erstellen
17    // IPC_CREAT = Semaphore erzeugen, wenn Sie noch nicht existiert
18    // IPC_EXCL = Neuen Semaphorgruppe anlegen und nicht auf evtl. existierende Gruppe zugreifen
19    returncode_semget1 = semget(sem_key1, 1, IPC_CREAT | IPC_EXCL | 0600);
20    if (returncode_semget1 < 0) {
21        printf("Die Semaphorgruppe %i konnte nicht erstellt werden.\n", sem_key1);
22        perror("semget");
23        exit(1);
24    }
```

Helpful documentation of semget

<https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semget/semget.html>

Simple Semaphore Example (in C) – Part 3/5

```
52 // Einen Kindprozess erzeugen
53 pid_des_kindes = fork();
54
55 // Kindprozess
56 if (pid_des_kindes == 0) {
57     for (int i=0;i<5;i++) {
58         semop(returncode_semget2, &p_operation, 1); // P-Operation Semaphore 54321
59         // Kritischer Abschnitt (Anfang)
60         printf("B");
61         sleep(1);
62         // Kritischer Abschnitt (Ende)
63         semop(returncode_semget1, &v_operation, 1); // V-Operation Semaphore 12345
64     }
65     exit(0);
66 }
67
68 // Elternprozess
69 if (pid_des_kindes > 0) {
70     for (int i=0;i<5;i++) {
71         semop(returncode_semget1, &p_operation, 1); // P-Operation Semaphore 12345
72         // Kritischer Abschnitt (Anfang)
73         printf("A");
74         sleep(1);
75         // Kritischer Abschnitt (Ende)
76         semop(returncode_semget2, &v_operation, 1); // V-Operation Semaphore 54321
77     }
78 }
```

Helpful documentation of semop

<https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semop/semop.html>

Simple Semaphore Example (in C) – Part 4/5

```
79 // Warten auf die Beendigung des Kindprozesses
80 wait(NULL);
81
82 printf("\n");
83
84 // Semaphorgruppe 12345 entfernen
85 returncode_semctl = semctl(returncode_semget1, 0, IPC_RMID, 0);
86 if (returncode_semctl < 0) {
87     printf("Die Semaphorgruppe %i konnte nicht entfernt werden.\n", returncode_semget1);
88     exit(1);
89 } else {
90     printf("Die Semaphorgruppe mit ID %i und Key %i wurde entfernt.\n", returncode_semget1, sem_key1);
91 }
92
93 // Semaphorgruppe 54321 entfernen
94 returncode_semctl = semctl(returncode_semget2, 0, IPC_RMID, 0);
95 if (returncode_semctl < 0) {
96     printf("Die Semaphorgruppe %i konnte nicht entfernt werden.\n", returncode_semget2);
97     exit(1);
98 } else {
99     printf("Die Semaphorgruppe mit ID %i und Key %i wurde entfernt.\n", returncode_semget2, sem_key2);
100 }
101
102 exit(0);
103 }
```

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

Simple Semaphore Example (in C) – Part 5/5

```
$ gcc semaphore_beispiel_systemv.c -o semaphore_beispiel_systemv
Wert der Semaphore mit ID 98362 und Key 12345: 1
Wert der Semaphore mit ID 98363 und Key 54321: 0
ABABABABAB
Die Semaphorgruppe mit ID 98362 und Key 12345 wurde entfernt.
Die Semaphorgruppe mit ID 98363 und Key 54321 wurde entfernt.
```

```
$ ipcs -s

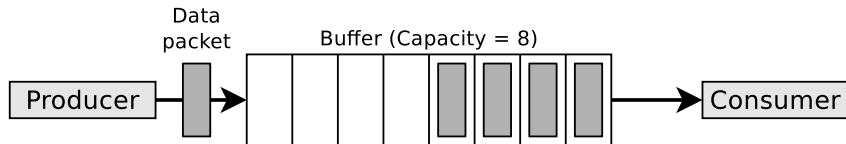
----- Semaphore Arrays -----
key          semid      owner      perms      nsems
0x00003039   98362        bnc        600         1
0x0000d431   98363        bnc        600         1

$ printf "%d\n" 0x00003039           # Convert from hexadecimal to decimal
12345
$ printf "%d\n" 0x0000d431
54321
```

- Without mutual exclusion by using the semaphores, the output sequence can be e.g. ABBABABABA or ABBAABABAB or ABABABABBA ...
- Without mutual exclusion by using the semaphores and without the sleep commands, the output sequence is usually AAAAABBBBBB and in rather seldom cases like AABAAABBBBB

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 \implies producer must be blocked
- Buffer = empty \implies consumer must be blocked



65/71

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)
 - $\text{empty} = 0 \implies$ puffer is completely filled \implies producer is 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)
 - $\text{filled} = 0 \implies$ puffer is empty \implies 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

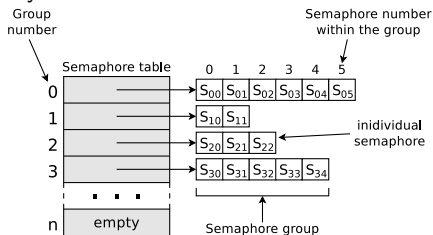
Producer/Consumer Example (3/3)

```
1 typedef int semaphore;           // semaphores are of type integer
2 semaphore filled = 0;            // counts the number of occupied locations in the buffer
3 semaphore empty = 8;             // counts the number of empty locations in the buffer
4 semaphore mutex = 1;             // controls access to the critical sections
5
6 void producer (void) {
7     int data;
8
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
19 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 }
```

Semaphores in Linux (System V)

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 **System V** semaphores

- `semget()`: Create new semaphore or a group of semaphores or open an existing semaphore
- `semctl()`: 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 (**System V**) provides the command `ipcs`

Semaphores in Linux (System V vs. POSIX)

- The concept of protecting critical sections described in so far is also called **system V semaphores** in the literature
- Some developers prefer the System V API and Others the POSIX API... ㄟ(ˊˋ)ㄟ

C function calls of the POSIX semaphores specified in the header file `semaphore.h`

- `sem_init()`: Create a new **unnamed** semaphore and thereby specify the initial value
- `sem_open()`: Create a new **named** semaphore and thereby specify the initial value
- `sem_post()`: Increment the value of a semaphore (V operation)
- `sem_wait()`: Decrement the value of a semaphore (P operation). Blocking operation
- `sem_trywait()`: Decrement the value of a semaphore (P operation). Non-blocking operation
- `sem_timedwait()`: Decrement the value of a semaphore (P operation). Blocking operation but with a timeout
- `sem_getvalue()`: Request the value of a semaphore
- `sem_destroy()`: Erase an **unnamed** semaphore
- `sem_close()`: Close a **named** semaphore
- `sem_unlink()`: Erase a **named** semaphore
- Named POSIX semaphores are created in Linux in the folder `/dev/shm` with names of the form `sem.<name>`

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

Mutexes

- If the Semaphore-feature of counting is not required, a simplified alternative, 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**
 - Mutexes are either for intra-process or inter-process synchronization

Functions for accessing a Mutex

<code>pthread_mutex_lock</code>	⇒	corresponds to the P operation
<code>pthread_mutex_unlock</code>	⇒	corresponds to the V operation

- A process that tries to access a critical section, calls `pthread_mutex_lock`
 - Critical section is **locked** ⇒ the process gets locked until the process in the critical section is finished and calls `pthread_mutex_unlock`
 - Critical section is **not locked** ⇒ the process can enter it

More information about mutexes

<https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>

Monitor and erase IPC Objects

- Information about existing (**System V**) shared memory segments, (**System V**) message queues and (**System V**) semaphores provides the command `ipcs`
- The easiest way to erase such shared memory segments, message queues and semaphores from the command line is the command `ipcrm`

```
ipcrm [-m shmids] [-q msgids] [-s semids]
      [-M shmkeys] [-Q msgkeys] [-S semkeys]
```

- **POSIX** memory segments and **POSIX** semaphores can be inspected and manually erased in the directory `/dev/shm`
- **POSIX** message queues can be inspected and manually erased in the directory `/dev/mqueue`