Operating Systems

Timo Hönig
Bochum Operating Systems and System Software (BOSS)
Ruhr University Bochum (RUB)

XI. Inter-Process Communication
June 28, 2023 (Summer Term 2023)

RUB Faculty of Computer Science

RUHR UNIVERSITÄT BOCHUM RUB

www.informatik.rub.de
Chair of Operating Systems and System Software

Agenda

- Recap
- Organizational Matters
- Principles of Inter-Process Communication
- Local Inter-Process Communication with UNIX-style Operating Systems
  - Signals
  - Pipes
  - Message Queues
- Inter-Process Communication Across Systems
  - Sockets
  - Remote Procedure Calls
- Summary and Outlook

Literature References

Silberschatz, Chapter 3.4

Tanenbaum, Chapters 2.3 and 2.5

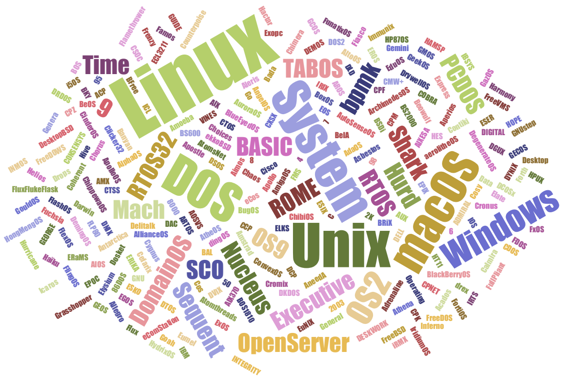
Recap

- scheduling processes to **maximize CPU utilisation**
- **long/medium/short-term** scheduling
- **basic** (FIFO, RR) and **advanced** (SPN, HRRN, MFQ) scheduling methods
  - discussion on practicability, performance, quality of the different scheduling methods
  - priority inversion: implications with process synchronisation
- **Linux O(1)-Scheduler**

Agenda

- Recap
- **Organizational Matters**
- Principles of Inter-Process Communication
- Local Inter-Process Communication with UNIX-style Operating Systems
  - Signals
  - Pipes
  - Message Queues
- Inter-Process Communication Across Systems
  - Sockets
  - Remote Procedure Calls
- Summary and Outlook

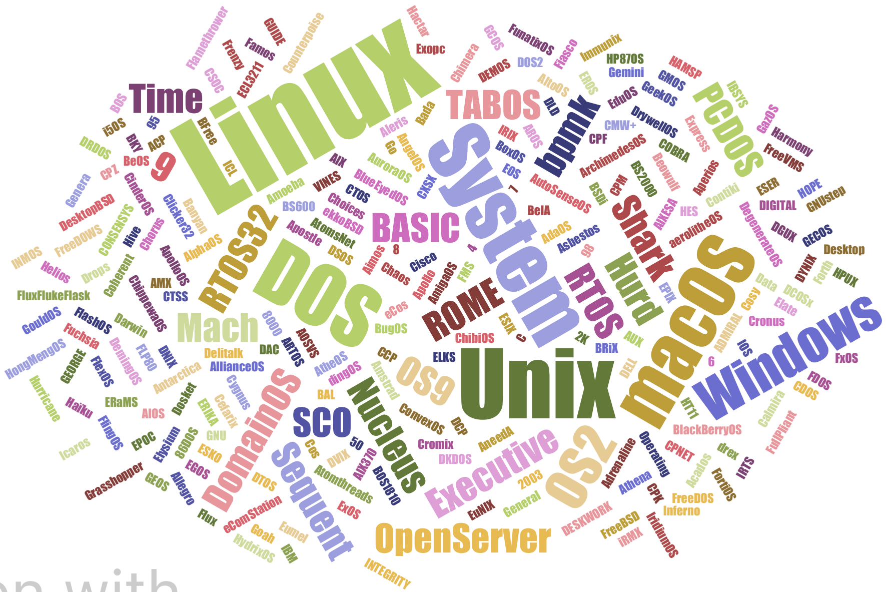
Organizational Matters



- lecture
  - Wednesday, 10:15 – 11:45
  - format: synchronous, **hybrid**
    - → in presence (Room HID, Building ID)
    - → online lecture (Zoom)
  - **exam:** August 7, 2023 (first appointment)
    September 25, 2023 (retest appointment)
  - **evaluation is live!**

    `https://tinyurl.com/25twb8qr`
- manage course material, asynchronous communication: Moodle
- `https://moodle.ruhr-uni-bochum.de/course/view.php?id=50698`

Agenda



- Recap
- Organizational Matters
- **Principles of Inter-Process Communication**
- Local Inter-Process Communication with UNIX-style Operating Systems
  - Signals
  - Pipes
  - Message Queues
- Inter-Process Communication Across Systems
  - Sockets
  - Remote Procedure Calls
- Summary and Outlook

Principles of Inter-Process Communication

- processes interact
  - wait for each other: **synchronisation**
  - share and exchange data: **communication**

- waiting mechanisms
  - are necessary for controlled communication
  - potentially lead to **deadlock** situations

- up to now: data **exchange** has only had minor consideration
  - lightweight and featherweight processes
    - ➔ in the same address space
    - ➔ using different address spaces
  - transfer of arbitrary types and amounts of data

Inter-Process Communication (IPC)

- multiple processes work jointly on **one** task
  - **simultaneous use** of available information **by several processes**
  - increase performance/reduce processing time by parallelisation
  - hiding of processing times by execution "in the background"

- communication through **shared memory**
  - data exchange: concurrent writing to (or reading from) a shared memory
  - attention must be paid to synchronisation

- **focus:** communication through **messages**
  - messages are exchanged between processes
  - shared memory is **not** mandatory

Message-based Communication

- communication primitives (typical signature):

```
send (destination, message)
receive (source, message)
```

- there are differences in:
  - synchronisation method
  - addressing mode
  - additionally: several properties

Synchronisation for Message-based IPC

- **synchronisation on send / receive**
  - **synchronous message exchange** (also „**rendezvous**“)
    - ➔ recipient blocks until the message is received
    - ➔ sender blocks until the arrival of the message is confirmed
  - **asynchronous message exchange**
    - ➔ sender passes the message to the operating system and then resumes operation
    - ➔ blocking on both sides optional
    - ➔ buffering always required
- frequently found:
  - asynchronous message exchange with potentially blocking sending and receiving

Addressing Modes for Message-based IPC

- **direct addressing**
  - process ID (e.g., signals)
  - communication endpoint of a process (e.g., port, socket)
- **indirect addressing**
  - channels (e.g., pipes)
  - mail boxes, message buffer (e.g., message queue)
- additional dimension: **addressing of groups**
  - **unicast** – send message to <u>exactly</u> one process
  - **multicast** – send message to <u>a selection</u> of processes
  - **broadcast** – send message to <u>all</u> processes

Additionally Properties of Message-based IPC

- **message format**
  - stream oriented vs. message oriented
  - fixed length vs. variable length
  - typed vs. untyped

- **transmission**
  - unidirectional vs. bidirectional (i.e., half-duplex, full-duplex)
  - reliable vs. unreliable
  - order of messages is kept vs. is not kept

Agenda

- Recap
- Organizational Matters
- Principles of Inter-Process Communication
- Local Inter-Process Communication with UNIX-style Operating Systems
  - Signals
  - Pipes
  - Message Queues
- Inter-Process Communication Across Systems
  - Sockets
  - Remote Procedure Calls
- Summary and Outlook

Signals

- signals are interrupts recreated in software
  - similar to interrupts of a processor through I/O devices
  - **minimal form** of inter-process communication (transmission of the signal number, **no payload**)

- sender:
  - processes - with the help of the system call **`kill(2)`**
  - operating system - when certain events occur

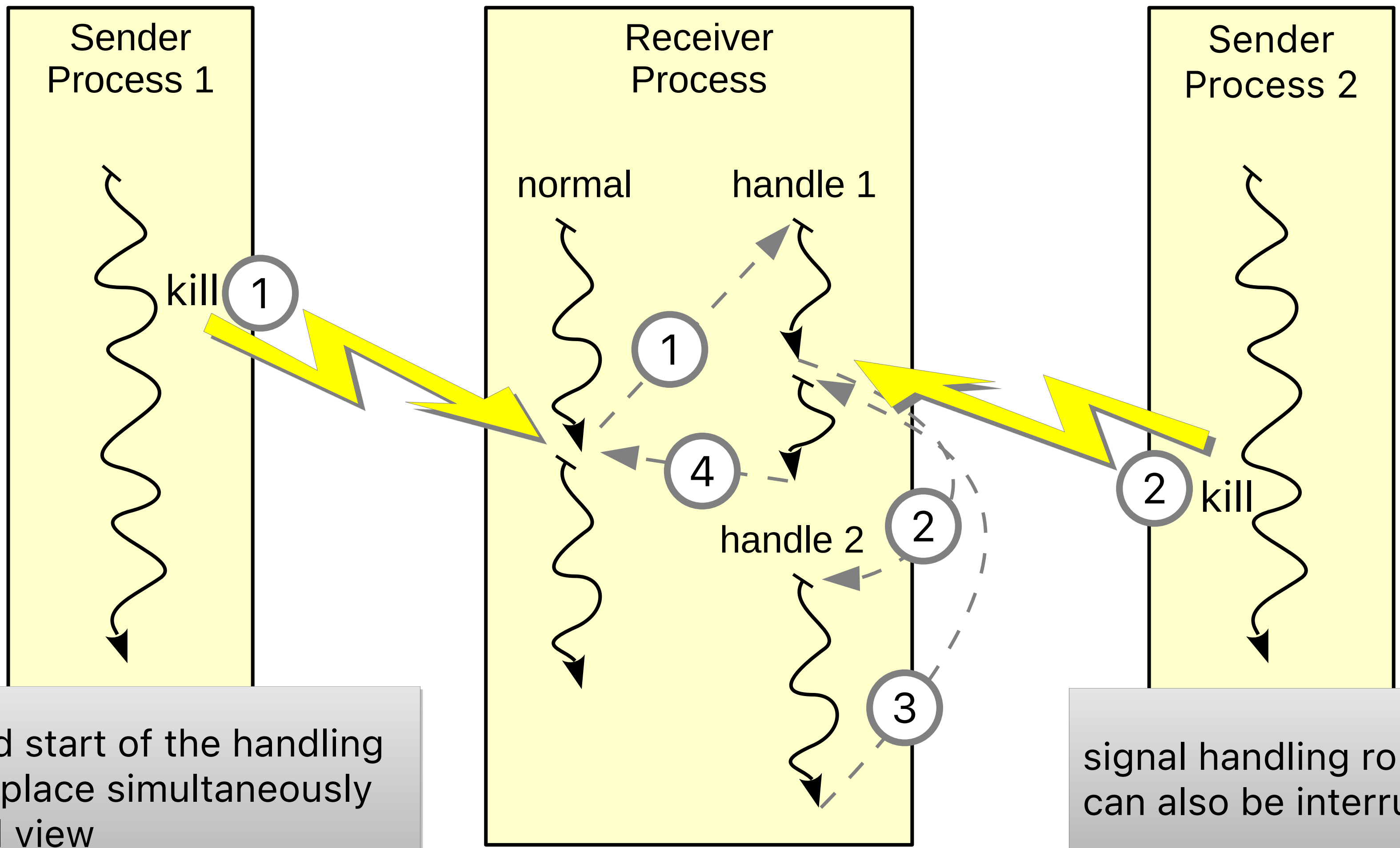
- receiver process performs signal handling:
  - ignore
  - terminate process
  - call a signal handler function
    - ➔ after handling the signal, the process continues at the interrupted position

Signals

- with the help of signals, processes can be informed about exceptional situations (c.f., hardware interrupts)
- examples:
  - **`SIGINT`** abort process (e.g., Ctrl-C)
  - **`SIGSTOP`** stop process (e.g., Ctrl-Z)
  - **`SIGCHLD`** child process terminated
  - **`SIGSEGV`** memory protection violation of the process
  - **`SIGKILL`** process is killed
- the default signal handling (terminate, stop, ...) can be redefined for most signals
  - see **`signal`**`(7)`

Signals – Logical View

Hollywood Principle: *„Don't call us, we'll call you."*



signaling and start of the handling routine take place simultaneously in the logical view

signal handling routines can also be interrupted

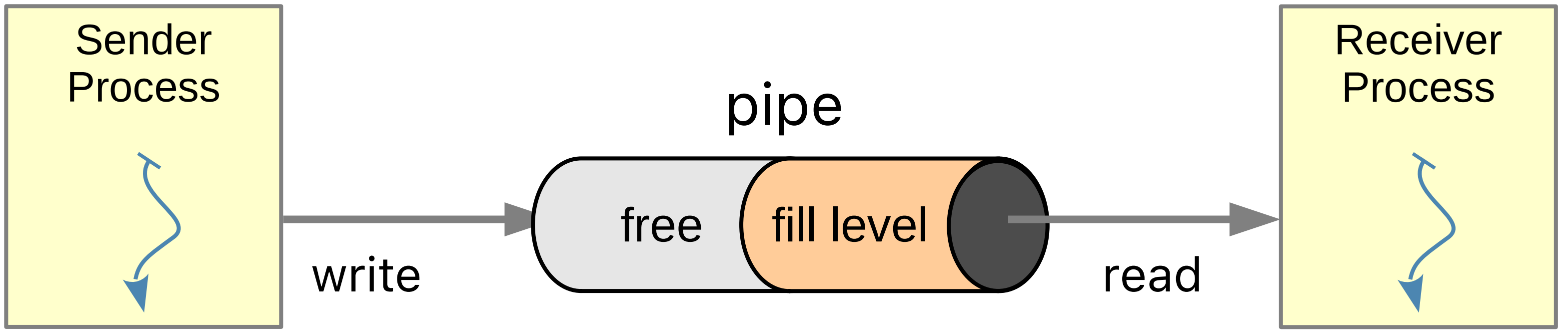
Signals – Technical View

- signal handling always takes place at the transition from kernel to user mode
- What happens when the receiving process...

1. runs in state **RUNNING** (e.g., segmentation error, bus error)?
   - → immediate start of the signal handling routine
2. does not run but is in state **READY** (e.g., `kill` system call)?
   - → the signal is registered in the process control block (PCB)
   - → as soon as the process runs again, the signal handling takes place
3. waits on an I/O event, in state **BLOCKED**?
   - → the I/O system call (e.g., `read`) is aborted with `EINTR`
   - → the process state is set to **READY**
   - → after that: as for 2.
   - → if applicable, the interrupted system call is executed again (`SA_RESTART`)

IPC with Pipes

- channel between two communication partners
  - unidirectional
  - buffered (fixed buffer size)
  - reliable
  - stream oriented
- pipe operations: read and write
  - order of the bytes is preserved (byte stream)
  - processes block when the pipe is full (write) or empty (read)



Pipes – Programming

- **unnamed pipes**
  - create a pipe:
    `int pipe (int pipefd[2])`
  - After a successful call (return value is zero):
    - → **read from** the pipe via `pipefd[0]` (system call read)
    - → **write to** the pipe via `pipefd[1]` (system call write)
  - now one just has to pass one end of the pipe to another process (next slide)

- **named pipes**
  - pipes can also be placed in the file system as special files:
    `int mkfifo (const char *filename, mode_t mode)`
  - standard functions (i.e., open, `read`, `write` and `close`) can be used
    - → file access rights control who can use the pipe

Pipes – Programming

```
enum { READ=0, WRITE=1 };

int main (int argc, char *argv[]) {
  int res, fd[2];
  if (pipe (fd) == 0) {                   /* create pipe */
    res = fork ();
    if (res > 0) {                        /* parent process */
      close (fd[READ]);                   /* close reading side */
      dup2 (fd[WRITE], 1);                /* write stdout to pipe */
      close (fd[WRITE]);                  /* free descriptor */
      execlp (argv[1], argv[1], NULL);    /* execute writer */
    }
    else if (res == 0) {                  /* child process */
      close (fd[WRITE]);                  /* close writing side */
      dup2 (fd[READ], 0);                 /* read from pipe as stdin */
      close (fd[READ]);                   /* free descriptor */
      execlp (argv[2], argv[2], NULL);    /* execute reader */
    }
  }
  /* error handling */
}
```

Pipes – Programming

"./connect ls wc" is equivalent to the shell command "ls | wc"

```c
enum { READ=0, WRITE=1 };

int main (int argc, char *argv[]) {
  int res, fd[2];
  if (pipe (fd) == 0) {                    /* create pipe */
    res = fork ();
    if (res > 0) {                         /* parent process */
```

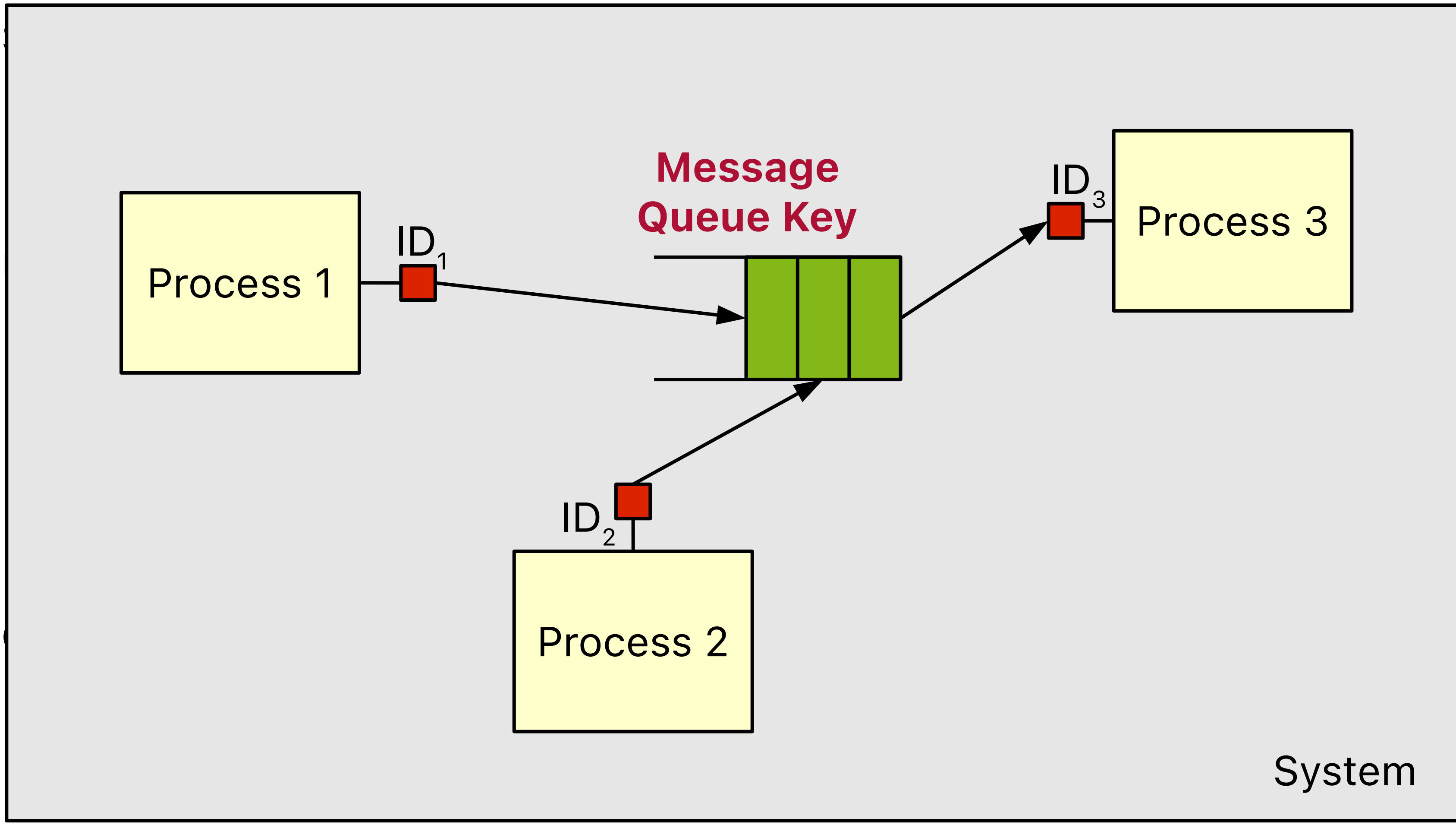
```
user@host$ ls
connect   connect.c   execl.c   fork.c   orphan.c   wait.c
user@host$ ./connect ls wc
      6        6       49
```

```c
      close (fd[WRITE]);                   /* close writing side */
      dup2 (fd[READ], 0);                  /* read from pipe as stdin */
      close (fd[READ]);                    /* free descriptor */
      execlp (argv[2], argv[2], NULL);     /* execute reader */
    }
  }
  /* error handling */
}
```

IPC with Message Queues

- system-wide unique address (key) serves for identification
  - access rights as to files
  - process local number (`msqid`) is required for **all** operations
- undirected M:N communication
- buffered operation
  - adjustable size per queue
- messages are typed (long)
- operations to send and receive a message
  - blocking vs. non-blocking
  - receive all messages vs. receive a specific message type only

IPC with Message Queues



Message Queues – Programming

- create a message queue and get message queue id:

```
int msgget (key_t key, int msgflg);
```

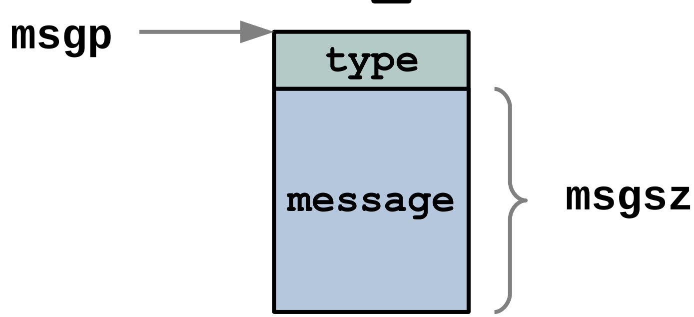
  - all communicating processes must know the key
  - keys are unique within an operating system instance
  - if a key is already assigned, **no** message queue with the same key can be created
- message queues can be created **without** a key (private queues, **key=IPC_PRIVATE**)
  - non-private message queues are persistent
  - they must be explicitly deleted (**cmd=IPC_RMID**):

```
int msgctl (int msqid, int cmd
            struct msqid_ds *buf);
```

Message Queues – Programming

- send a message

```c
int msgsnd (int msqid, const void *msgp,
            size_t msgsz, int msgflg);
```



- receive a message

```c
int msgrcv (int msqid, void *msgp, size_t msgsz,
            long msgtype, int msgflg);
```

  - `msgtype = 0`: first message
  - `msgtype > 0`: first message with given type
  - `msgtype < 0`: message with smallest type <= |msgtype|

Message Queues – Commands

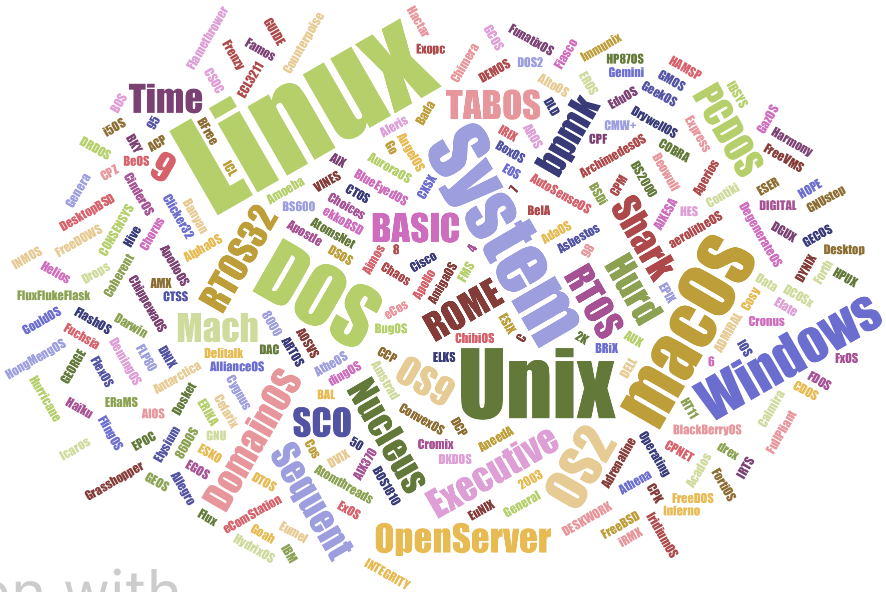
- show all message queues

```
ipcs -q
```

- remove message queue

```
ipcrm -Q <key>
```

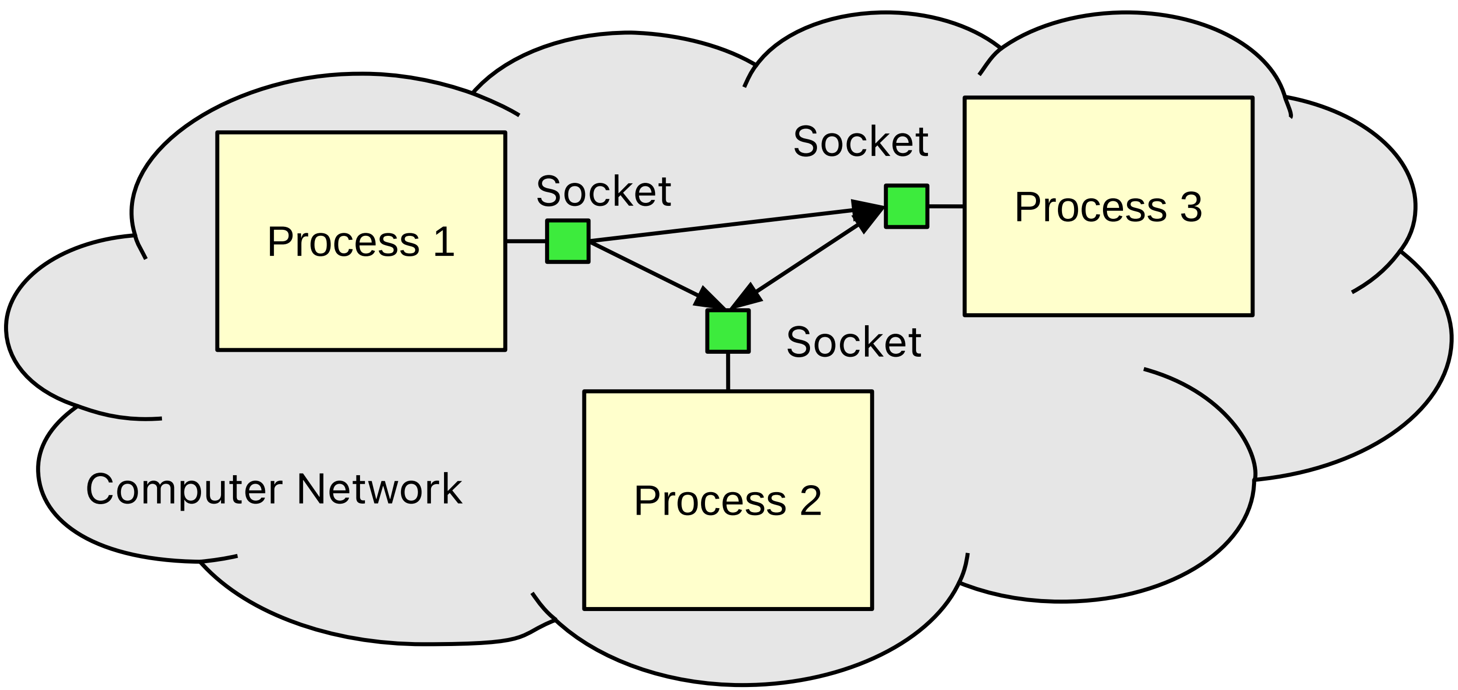
Agenda



- Recap
- Organizational Matters
- Principles of Inter-Process Communication
- Local Inter-Process Communication with UNIX-style Operating Systems
  - Signals
  - Pipes
  - Message Queues
- **Inter-Process Communication Across Systems**
  - **Sockets**
  - **Remote Procedure Calls**
- Summary and Outlook

Sockets

- sockets are **general communication endpoints** in a computer network
  - bidirectional
  - buffered
- abstracts from details of the communication system
  - characterised by **domain** (protocol family), **type** and **protocol**



Socket Domains

- UNIX domain socket
  - UNIX **domain sockets** behave like **bidirectional pipes**
  - creation as a special file in the file system is possible
- Internet domain socket
  - designed for inter-computer communication using Internet protocols
  - Appletalk Domain, DECnet Domain, ...
- domains determine possible protocols
  - for example, Internet domain: TCP/IP or UDP/IP
- domains determine the address family
  - example: Internet domain - IP address and port number

Socket Types and Protocols

- the major types of sockets:
  - **`SOCK_STREAM`:** stream-oriented, connection-oriented, reliable (TCP)
  - **`SOCK_DGRAM`:** message-oriented, unreliable (UDP)

- Internet domain protocols:
  - TCP/IP protocol
    - → stream- and connection-oriented, reliable
  - UDP/IP protocol
    - → message-oriented, connectionless, unreliable
    - → messages can be lost or duplicated
    - → sequence can get out of order
    - → packet boundaries are preserved (datagram protocol)

→ protocol specification is often redundant

Socket Programming

- creating sockets
  - generate a socket with system call **`socket`** (return value is a file descriptor):

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

  - address binding
    - sockets are generated without address
    - binding to an address is done by:

```
int bind (int sockfd,
          const struct sockaddr_in *addr,
          socklen_t addrlen);
```

    - **`struct sockaddr_in`** (for Internet address family) contains:

`sin_family`: `AF_INET`
`sin_port`: 16 bit port number
`sin_addr`: struct with IP address (e.g., 192.168.2.1)

note: for IPv6 there are `sockaddr_in6` and `AF_INET6`

Socket Programming

- **datagram sockets**
  - packet-oriented
  - no connection setup necessary
  - **sending** a datagram:

```
ssize_t sendto (int sockfd, const void *buf,
             size_t len, int flags,
                const struct sockaddr *dest_addr,
                socklen_t addrlen);
```

  - **receiving** a datagram:

```
ssize_t recvfrom (int sockfd, void *buf,
                  size_t len, int flags,
                  struct sockaddr *src_addr,
                  socklen_t *addrlen);
```

Socket Programming

- **stream sockets**
  - stream-oriented
  - connection setup necessary
  - client (user, user program) wants to establish a **communication connection** to a server (service provider)
- client: connection setup for stream-oriented sockets
  - connecting the socket with:

```
int connect (int sockfd,
             const struct sockaddr *addr,
             socklen_t addrlen);
```

  - send and receive with **`write`** and **`read`** (or **`send`** and **`recv`**)
  - terminate the connection with **`close`** (closes the socket)

Socket Programming

- server: accepts requests/orders
  - binds socket to an address (otherwise not reachable)
  - prepares socket for connection requests with:

```
int listen (int sockfd, int backlog);
```

  - accepts individual connection requests with:

```
int accept (int sockfd, struct sockaddr *addr,
            socklen_t *addrlen);
```

    - ➔ returns a **new socket** connected to the client
    - ➔ blocks if there is **no** connection request
  - reads data with **`read`** and executes the implemented service
  - sends the result with **`write`** back to the sender
  - closes the new socket with **`close`**

Socket Programming

```
#define PORT 4711
#define MAXREQ (4096*1024)

char buffer[MAXREQ], body[MAXREQ], msg[MAXREQ];

void error(const char *msg) { perror(msg); exit(1); }

int main() {
  int sockfd, newsockfd;
  socklen_t clilen;
  struct sockaddr_in serv_addr, cli_addr;
  int n;
  sockfd = socket(PF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) error("ERROR opening socket");
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(PORT);
  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    error("ERROR on binding");
  listen(sockfd,5);
  ...
```

socket is created and bound to an address

Socket Programming

```
  ...
  while (1) {
    clilen = sizeof(cli_addr);
    newsockfd = accept (sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0) error("ERROR on accept");
    bzero(buffer,sizeof(buffer));
    n = read (newsockfd, buffer, sizeof(buffer)-1);
    if (n < 0) error("ERROR reading from socket");
    snprintf (body, sizeof (body),
              "<html>\n<body>\n"
              "<h1>Hi Browser</h1>\nYour request was: ...\n"
              "<pre>%s</pre>\n"
              "</body>\n</html>\n", buffer);
    snprintf (msg, sizeof (msg),
              "HTTP/1.0 200 OK\n"
              "Content-Type: text/html\n"
              "Content-Length: %d\n\n%s", strlen (body), body);
    n = write (newsockfd, msg, strlen(msg));
    if (n < 0) error("ERROR writing to socket");
    close (newsockfd);
  }
}
```

accepting a new connection

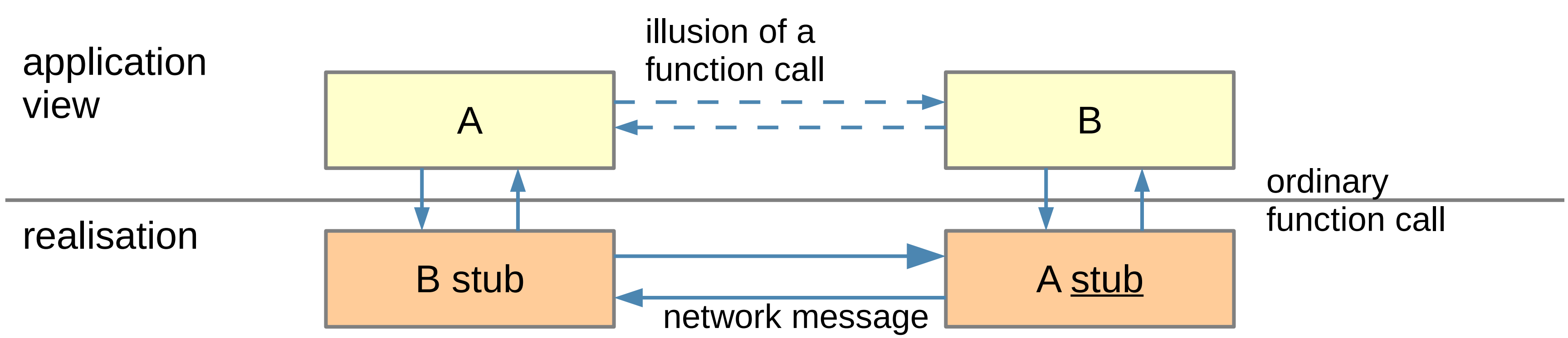
read HTTP request

generate reply and send back to client

close connection

Remote Procedure Call (RPC)

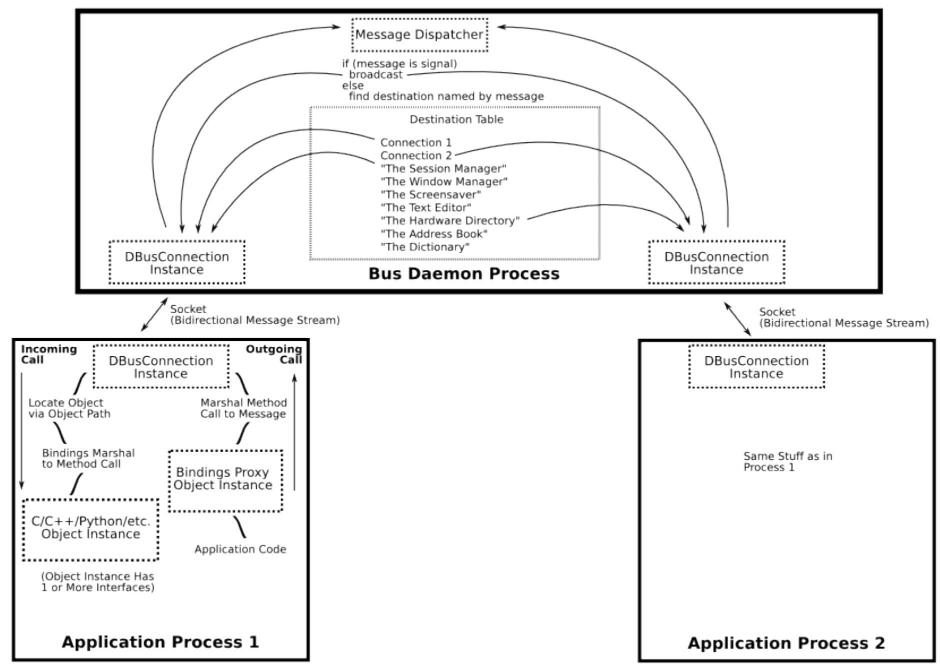
- function call across process boundaries (dt. Fernaufruf)
  - high level of abstraction
  - remote procedure calls are rarely offered directly by the system; needs mapping to other forms of communication (e.g., to messages)
  - mapping to multiple messages
    - ➔ job message transports call intention and parameters
    - ➔ result message transports results of the call



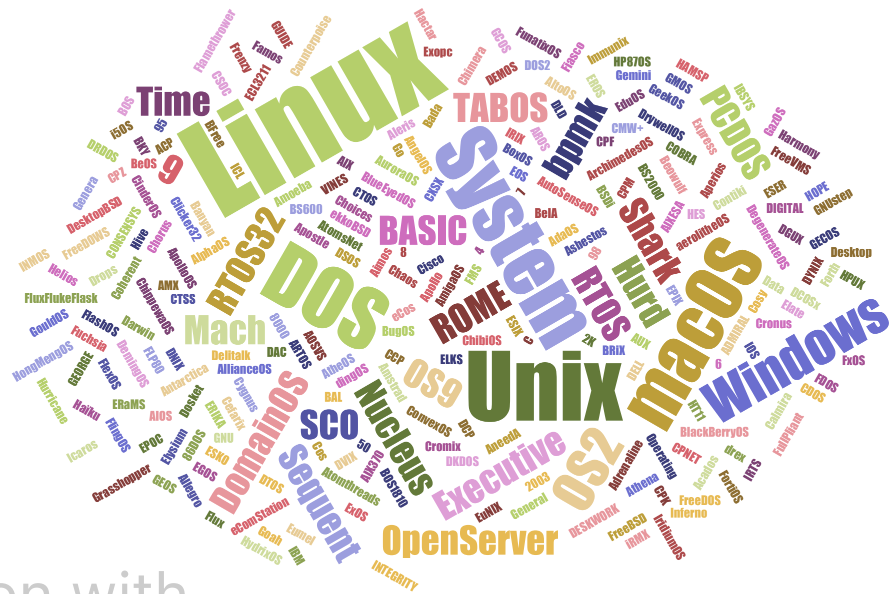
- Examples: NFS, D-Bus

Remote Procedure Call (RPC) – Discussion

- flexible means to dynamically provide functionalities of one process to others
- RPCs latencies
  - using RPCs can lead to high overheads
  - RPC overheads apply to both, on-system IPC and across-system IPC
- Example: D-Bus
  - high number of context switches
  - comfort comes at the expense of performance



Agenda



- Recap
- Organizational Matters
- Principles of Inter-Process Communication
- Local Inter-Process Communication with UNIX-style Operating Systems
  - Signals
  - Pipes
  - Message Queues
- Inter-Process Communication Across Systems
  - Sockets
  - Remote Procedure Calls
- **Summary and Outlook**

⏭ Summary and Outlook

- **summary**
  - types of interprocess communication: **message-based** and **shared memory**
  - message-based IPC
    - → data is copied
    - → works across computer boundaries
  - UNIX systems offer various abstractions
    - → signals, pipes, sockets, message queues
    - → sockets: standardised interface,
      implemented by all general-purpose operating systems
- **outlook: multi and many-core systems, systems research**
  - challenges in operating system design for large number of processing cores
  - advanced topics in operating system design, research topics at the chair

References and Acknowledgments

Lecture

- Systemnahe Programmierung in C (SPiC), Betriebssysteme (Jürgen Kleinöder, Wolfgang Schröder-Preikschat)
- Betriebssysteme und Rechnernetze (Olaf Spinczyk, Embedded Software Systems Group, Universität Osnabrück)

Teaching Books and Reference Book

[1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, John Wiley & Sons*, 2018*.

[2] Andrew Tanenbaum, Herbert Bos: *Modern Operating Systems*, Pearson, 2015.

[3] Wolfgang Schröder-Preikschat: *Grundlage von Betriebssystemen – Sachwortverzeichnis*, 2023.
`https://www4.cs.fau.de/~wosch/glossar.pdf`