„Systemnahe Programmierung“ (BA-INF 034) Wintersemester 2023/2024

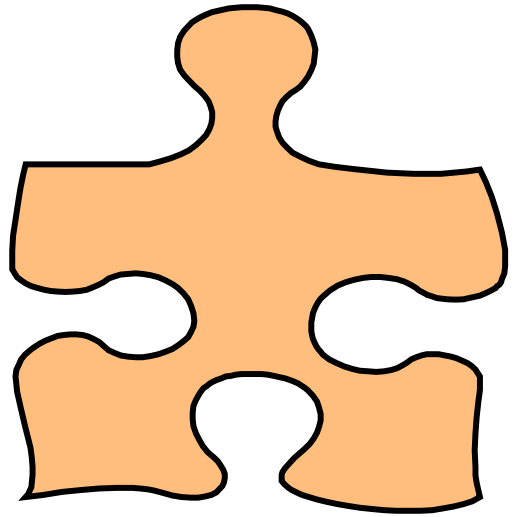
Dr. Matthias Frank, Dr. Matthias Wübbeling

Institut für Informatik 4
Universität Bonn

E-Mail: *{matthew, matthias.wuebbeling}@cs.uni-bonn.de*
Sprechstunde: nach Vereinbarung

2. Fortgeschrittene Konzepte der Systemprogrammierung

3. Betriebssysteme/Threads
„Kommunikation innerhalb von Prozessen bzw. zwischen Prozessen eines Rechners“



2.1. Processes

2.2. Threads

2.3. Interprocess Communication IPC

Teil 3

Literature

- Mark Mitchell, Jeffrey Oldham, and Alex Samuel. Advanced Linux Programming. New Riders Publishing. First edition, 2001.
http://www.advancedlinuxprogramming.com/

- W. Richard Stevens, Stephen A. Rago. Advanced Programming in the UNIX Environment. Addison-Wesley. Third Edition, 2013.
also see http://www.apuebook.com/

- W. Richard Stevens. UNIX Network Programming, Volume 2: Interprocess Communications. Prentice Hall PTR. Second Edition, 1999.

Threads vs. Processes (1)

- The traditional model of UNIX assumes **interacting processes**
- Threads are a relatively new model (multithreading in Linux Kernel since 1996)
- (UNIX) version of APACHE defaults to pre-forking (version 2.4)
- Most performant solutions e.g. nginx use libevent without threads (one process per CPU)
- Similarities: Both require stack and execution context
- Main question: Which information is shared?
- Threading defaults to everything
- Processes defaults to nothing, can be changed on-demand

Threads vs. Processes (2)

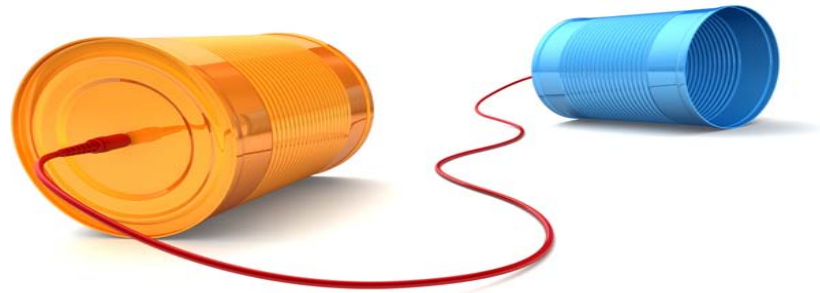
- Stability:
  - Processes are independent and cannot crash each other
- Flexibility:
  - Processes can define which information should be shared
  - Processes can reside on different systems (networking)
- Reliability:
  - Correct synchronization difficult in multi-threaded applications

→Use **processes for loosely coupled tasks**

→Various **interprocess communication primitives (IPC)** for sharing of information, synchronization and providing for reliable and efficient **handling of common interaction patterns**

Interprocess Communications (IPC)

- **Unix IPC** includes several mechanisms for
  - Communication
  - Synchronization
  - Event notification
- General methods as well as elegant solutions for common cases
- Producer – consumer paradigm
- Increased flexibility
- Multiple persistence schemes
- Partially with networking support



2.3. IPC – Outline

2.3.1. Basics

2.3.2. Files

file IO – as probably already known from basic programming

2.3.3. Message Passing

- .1 Pipes and FIFOs (Named Pipes)
- .2 Message Queues
- .3 Sockets

sockets for network communication as discussed in ch. 3.

2.3.4. Shared Memory

2.3.5. Semaphores

semaphores with processes as already mentioned in ch. 2.2.

2.3.6. Signals

2.3.1. Basics – IPC Flavors

- System V IPC
  - UNIX System V is a UNIX system from AT&T first released 1983
  - SysV and BSD were the dominant UNIX lines
  - Introduced several IPC concepts
- POSIX IPC
  - “Portable Operating System Interface”
  - A family of IEEE standards defining an Unix API
  - First part 1988
  - Concepts similar to SysV IPC with slightly different APIs
- D-Bus
  - (High level) communication between desktop applications
  - Libdbus, kdbus, systemd-bus, dbus-java, as implementations
- In this lecture: **POSIX IPC**

Communicate with Whom?

- How to find the communication partner?

1. “Relationship”, e.g., parent-child (comparable to threads)
2. “Name” for communication channel
   - Networking: host + port
   - File system:
     - For files: path
     - Files as pointer, e.g., PID files with well-known name (/var/run/*.pid)
   - POSIX IPC names (next slide)
   - D-Bus namespaces
     - org/gnome/..

Posix IPC Names (1)

- POSIX Standard defines
  - Pathname-like string (limited to PATH_MAX bytes including terminating null byte)
  - If starting with '/', access from different processes to the same name access the same IPC channel
- Several details undefined
  - Multiple '/' allowed?
  - Effect when not starting with '/'?
  - Visible in file system?

Posix IPC Names (2)

- For Linux and Solaris use: **“/somename”**
  - Begins with ‘/’
  - Just 1 ‘/’!
- Linux
  - Name resides in virtual file system
  - Can be explicitly mounted
- Solaris
  - Name results in actual file in “/tmp”, e.g., “/tmp/MQsomename”
- Portability problem example: Digital Unix 4.0b
  - Creates actual file in the file system
  - Permission problems! -> use of “/tmp/somename” recommended

Sharing of Information

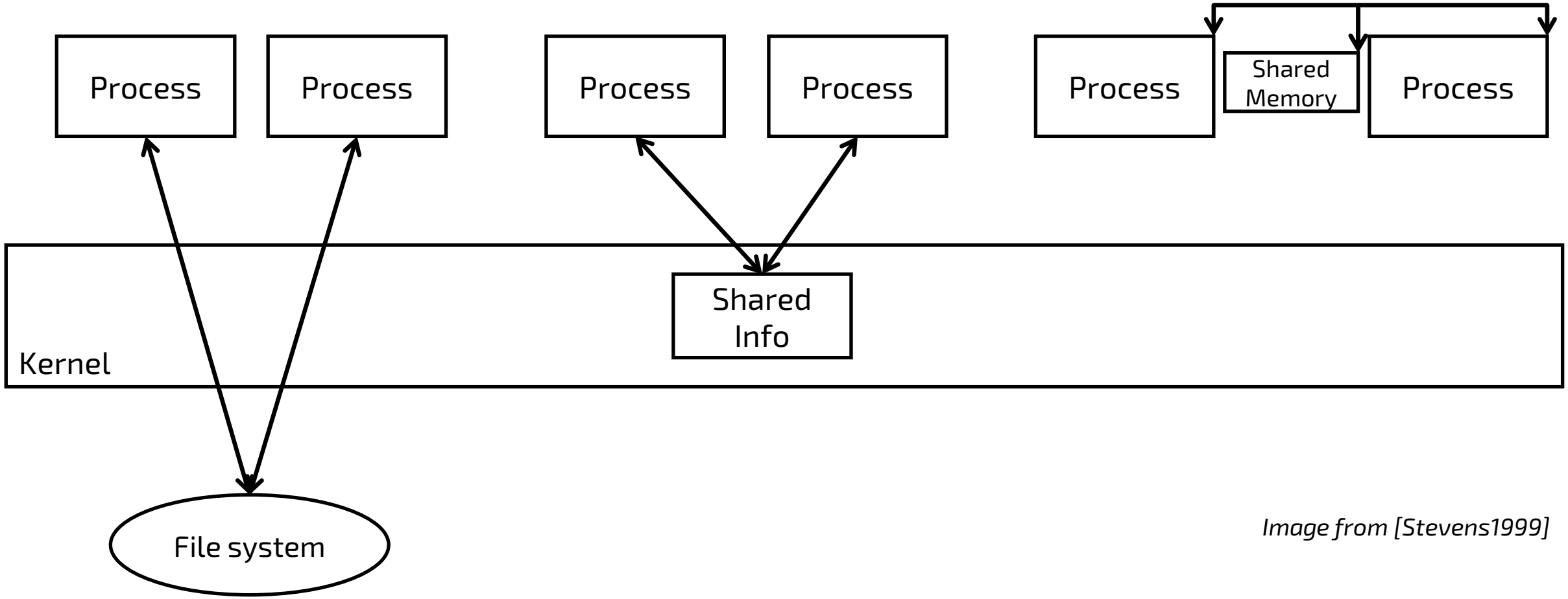


Image from [Stevens1999]

Sharing of Information – Threads, cf. chapter 3.2.

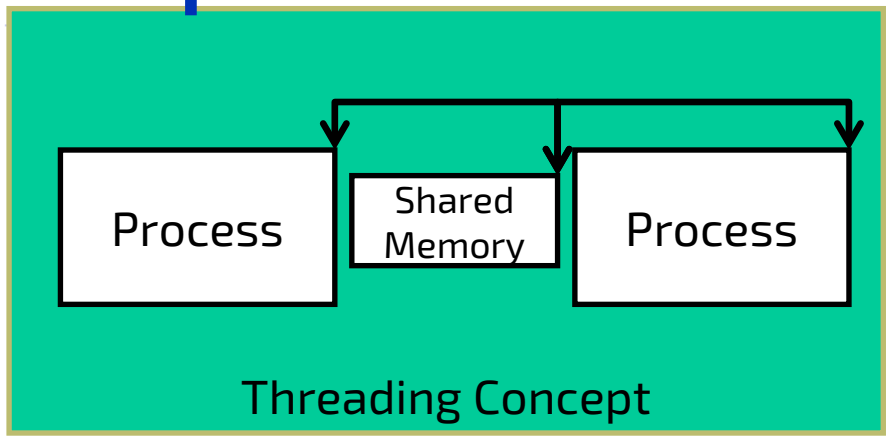


Image from [Stevens1999]

Note:
This figure shows where an IPC object is maintained.
Cf. „persistence“ on the following slides.

Persistence

Definition from [Stevens1999]:

“We can define the persistence of any type of IPC as how long an object of that type remains in existence.”

Note:

- Persistence can be different from where (and how long) the information is maintained

Persistence

1. Process persistence
   - IPC objects exists until all accessing processes closes connection
   - E.g., network connection by sockets
2. Kernel persistence
   - Until kernel reboot
   - Or until explicit deletion of IPC object
   - E.g., message queue
3. Filesystem persistence
   - Until explicit deletion
   - Survives reboots
   - E.g., files, databases

   Again: Persistence can be different from where (and how long) the information is maintained

Persistence of various types of IPC objects

from [Stevens1999]

Type of IPC	Persistence
Pipe	process
FIFO (named pipes)	process
Posix mutex	process
Posix condition variable	process
Posix read-write lock	process
fcntl record locking	process
Posix message queue	kernel
Posix named semaphore	kernel
Posix memory-based semaphore	process
Posix shared memory	kernel
System V message queue	kernel
System V semaphore	kernel
System V shared memory	kernel
TCP socket	process
UDP socket	process
Unix domain socket	process

Persistence – Notes

"…" from [Stevens1999]

- "Note that no type of IPC has filesystem persistence, but we have mentioned that the three types of POSIX IPC may, depending on the implementation." (see below)
- "Obviously, writing data to a file provides filesystem persistence, but this is normally not used as a form of IPC. Most forms of IPC are not intended to survive a system reboot, because the processes do not survive the reboot."

Filesystem Persistence:

- "Posix message queues, semaphores, and shared memory have this property, if they are implemented using mapped files (not a requirement)."

Persistence – Notes (ctd.)

"…" from[Stevens1999]

- Once again: Persistence can be different from where (and how long) the information is maintained.
- "We must be careful when defining the persistence of an IPC object, because it is not always as it seems."
- E.g. "the data within a pipe is maintained within the kernel, but pipes have process persistence and not kernel persistence – after the last process that has the pipe open for reading closes the pipe, the kernel discards all the data and removes the pipe."
- "Similarly, even though FIFOs have names within the filesystem, they also have process persistence because all the data in a FIFO is discarded after the last process that has the FIFO open closes the FIFO."

Possible Effects of fork, exec, _exit: *fork*

1. Child obtains copies of parent's handle, e.g., files
   - Child can interact with same objects as parent
   - Often used for interaction between child and parent
   - For some mechanisms, only possibility to establish communication channel
2. Shared memory also shared by child
   - Applies to shared memory only
3. Shared if in shared memory and process-shared attribute, e.g., mutex, semaphore
   - Accessible by any thread of any of the involved processes

Possible Effects of fork, exec, _exit: *exec*

1. Handles are still open unless specified to be closed, e.g., files
   - Allows anonymous IPC between processes with different executables
   - Possible security loophole
2. Handles are automatically closed, e.g., POSIX IPC
   - Applies to most named IPC objects
3. IPC objects vanish unless in shared memory and process-shared attribute, e.g., mutex
   - Only possibility for objects relying on shared memory, e.g., thread synchronization primitives

Possible Effects of fork, exec, _exit: *_exit*

- _exit: is called by exit() library function for cleanup
- All handles are closed
  - If the last handle to an IPC object is closed -> persistence schemes
- Important for synchronization and locking
  - Doesn't apply for thread synchronization (whole process exits)
  - Locks are released
  - Posix semaphores are **not** increased
  - Possible deadlock

The Way of Unix (1)

"This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to **handle text streams**, because that is a universal interface."

Doug McIlroy, inventor of Unix pipes

The Way of Unix (2)

- **Everything is a file**
  - Files/Disks/Partitions
  - Network connections/IPC
  - Devices (USB devices, mice, serial ports)
  - /proc/*
- Text files
  - Configuration files, log files, …
- Small tools doing one thing
- A few general multi-purpose tools
  - sed, awk, …
- Use of non-descriptive acronyms

Example: GNU Coreutils

- 86 commands for file, text and shell manipulation

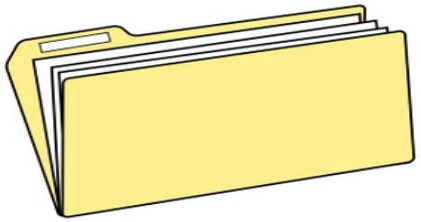
Image created with http://www.wordle.net/

2.3.2. Communication with Files

- Processes can communicate with files
  - Running at different times: persistence
  - Running simultaneously: files can be opened from multiple processes
- Simple, well-known operations: open, read, write, …
- With network file systems: support for processes on different systems
- But: no synchronization
  - Interference between processes possible
  - File-locks

Files API

- Rules for path names
  - Limited to PATH_MAX bytes including terminating null byte
  - API: no forbidden chars (besides `\0`), file system implementations can impose limitations
- Functions operate on **file descriptor**
  - “Small integer”
  - `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO` for process standard input/output/error
- Basic flow
  - open: obtain file descriptor for name
    - read/write: access file
    - lseek: set position for next read/write
  - close: free file descriptor



Files API – open and create files (1)

```
#include <fcntl.h>
int open(const char *pathname, int oflag, ... /* mode_t mode */ );
```

- Returns file descriptor or -1 in case of error
- oflag: options for opening/creating; “or”-mask

example
next slide

  - O_RDONLY, O_WRONLY, O_RDWR: read only, write only, read/write access: exactly one of these must be specified
  - O_APPEND: write appends to end of file
  - O_TRUNC: truncate file to length 0
  - O_CREAT: create file if not existing; if specified, third parameter
  - mode: access permissions, cf. next slide
  - O_EXCL: error if O_CREAT is specified and file already exists

Files API – open and create files (2)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *path, mode_t mode);
```

- Equivalent to

  open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)

example with „or“-masking of flags

- mode: “or”-mask of
- S_IRUSR, S_IWUSR, S_IXUSR: read, write, execute for user
- S_IRGRP, S_IWGRP, S_IXGRP: read, write, execute for group
- S_IROTH, S_IWOTH, S_IXOTH: read, write, execute for other

Files API – read

```
#include <unistd.h>
ssize_t read(int filedes, void *buf, size_t nbytes);
```

- Returns the number of bytes read
  - 0 indicates end of file (EOF)
  - -1 indicates error
  - May be less than nbytes
    - File size
    - Network buffering
    - Reading from terminal: Usually one line per call is read
    - When interrupted by signal

… or close of TCP connection by communication peer, cf. ch. 3

Files API – write

#include <unistd.h>

ssize_t **write**(int *fildes*, const void **buf*, size_t *nbyte*);

- Returns the number of bytes written
  - -1 indicates error
  - may be less than nbyte
    - Disk full
    - Quota exceeded
- If file was opened with O_APPEND
  - Write always appends to end of file

Files API – close and unlink

#include <unistd.h>

int close(int *filedes*);

- Closes file
  - No further operations on filedes possible
- All files are automatically closed when process exits

int unlink(const char **path*);

- Removes name from file system -> no more opens possible
- Data is accessible until no process has file open
  - Reading/Writing still possible
  - File is removed after last process calls close

Files API – lseek

```
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
```

- Moves file pointer for next read/write operation
  - Pointer can be set beyond the current end of the file
- offset can be positive or negative!
- whence: relative to what
  - SEEK_SET: beginning of file
  - SEEK_CUR: current position
  - SEEK_END: end of the file
- Returns absolute position after seek
  - Use lseek(filedes, 0, SEEK_CUR) to obtain current position
  - -1 on error, e.g., no regular file

Files API – Example: Append Timestamp to File

includes and get_timestamp() implementation omitted

```
int main (int argc, char* argv[])
{
      /* The file to which to append the timestamp. */
      char* filename = argv[1];
      /* Get the current timestamp. */
      char* timestamp = get_timestamp ();
      /* Open the file for writing. If it exists, append to it;
        otherwise, create a new file. */
      int fd = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
      /* Compute the length of the timestamp string. */
      size_t length = strlen (timestamp);
      /* Write the timestamp to the file. */
      write (fd, timestamp, length);
      /* All done. */
      close (fd);
      return 0;
}
```

Example Output

```
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
Thu Feb 1 23:25:47 2001
```

(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)

File Permissions: https://en.wikipedia.org/wiki/File_system_permissions

Portable (Buffered) File API

- #include <stdio.h>
- fopen, fread, fwrite, fseek corresponding calls
  - Call `open`, `read`, `write`, `lseek` internally
- Buffered Streams
- In most cases you should use the stream version
  - Platform-independent
  - Easier to use

Stream API – Example: Append Timestamp to File

includes and get_timestamp() implementation omitted

```
int main (int argc, char* argv[])
{
      /* The file to which to append the timestamp. */
      char* filename = argv[1];
      /* Get the current timestamp. */
      char* timestamp = get_timestamp ();
      /* Open the file for writing. If it exists, append
        otherwise, create a new file. */
      FILE fp = fopen (filename, "w+");
      /* Compute the length of the timestamp string. */
      size_t length = strlen (timestamp);
      /* Write the timestamp to the file. */
      fwrite (timestamp, sizeof(char), length, fp);
      /* All done. */
      fclose (fp);
      return 0;
}
```

Example Output

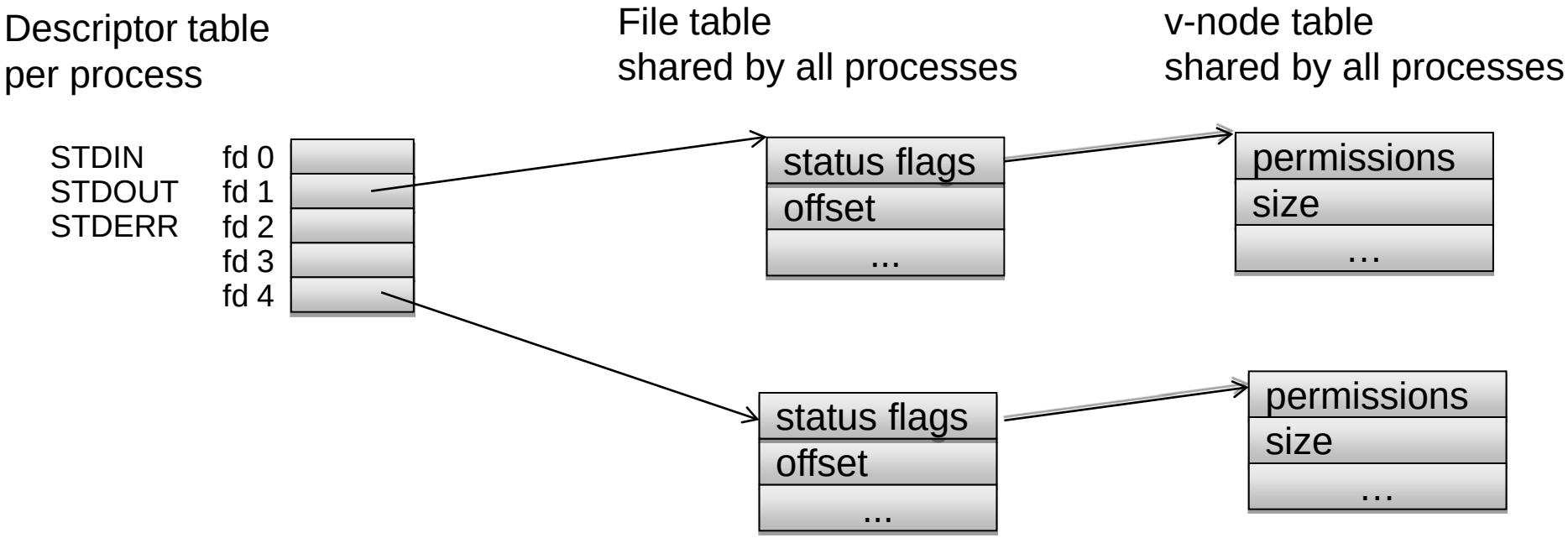
```
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
Thu Feb 1 23:25:47 2001
```

Source

(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)

File Management Structures

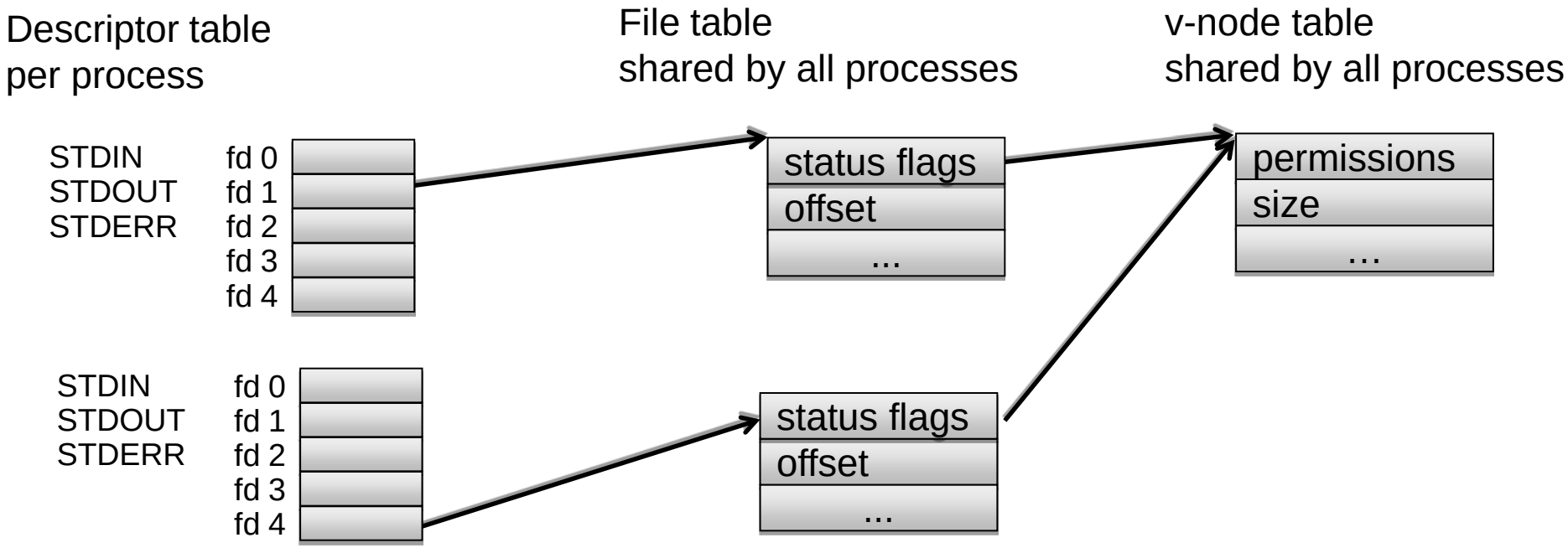
- File descriptor: file descriptor flags
- File table: offset, flags from open, …
- V-node table: information about file on disk



Weitere Infos: https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L09/Class.html

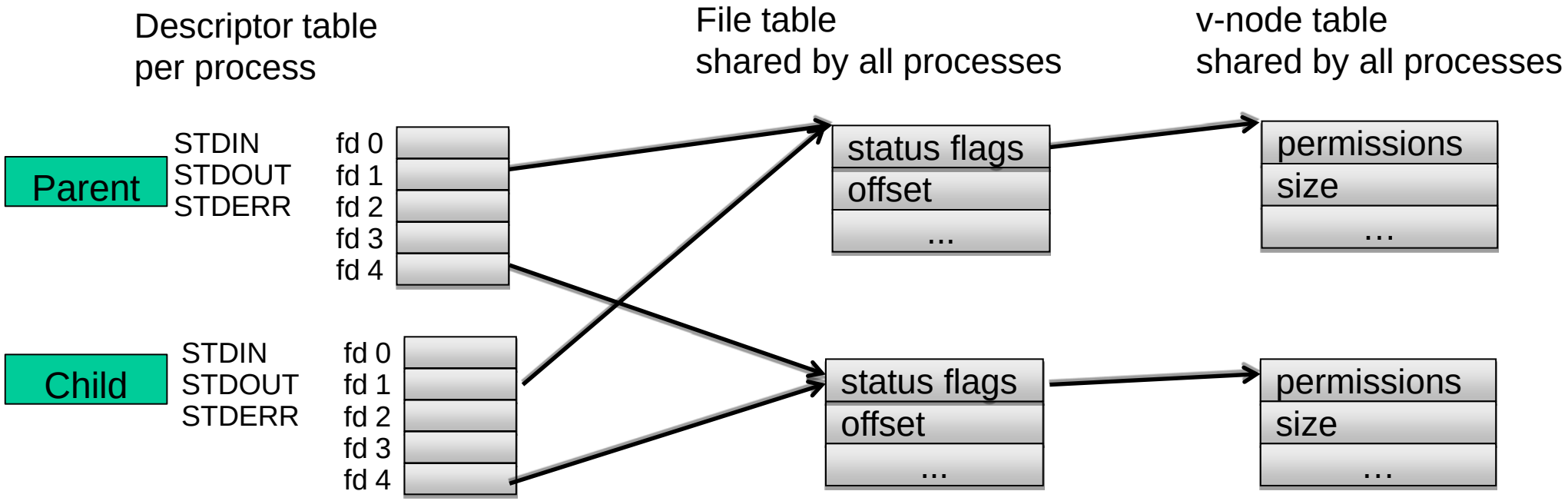
File Sharing (1)

- Two processes open the same file
- Similar for two open calls for same filename



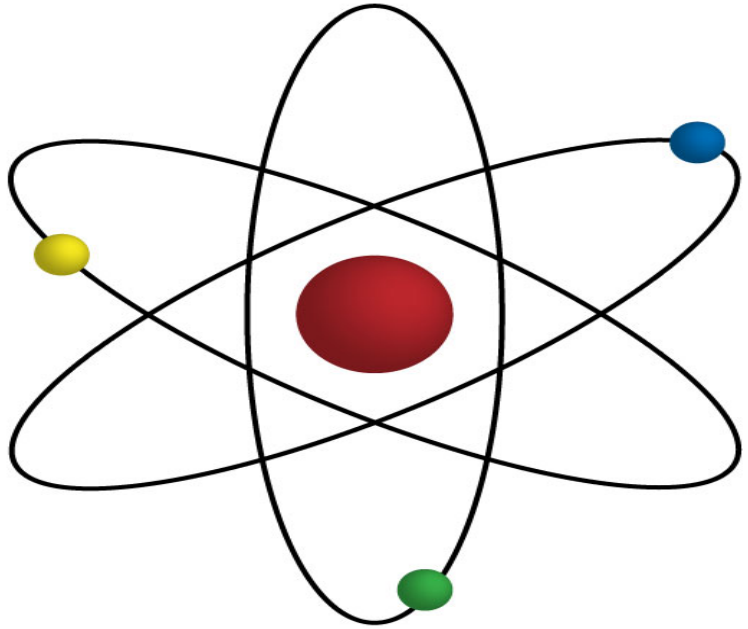
File Sharing (2)

- After fork
- File offset is shared!
- Important, e.g., for STDOUT: writes interleave but do not overwrite



Atomicity of Operations

- All operations are generally atomic
  - Read/write: data of one operation is not interleaved with data of other process
- No guarantees between calls
  - O_APPEND necessary instead of lseek+write



2.3.3. Message Passing – 2.3.3.1. Pipes (1)

- Well known from shell usage, e.g., who | sort
  - Use output of one command as input for another
- API perspective: 2 connected file descriptors
  - Write in one, read from the other
- Oldest form of Unix IPC
- Half-Duplex* (information flows only in one direction)*
  - Modern versions support also full-duplex
- Only possible for processes with same ancestor
  - Usually: parent-child

* unfortunately, mismatch of using terms simplex, half-duplex and full-duplex
cf. http://www.tldp.org/LDP/lpg/node9.html "Pipes ... a method of one-way communications (hence the term half-duplex) …"

vs. http://en.wikipedia.org/wiki/Duplex_%28telecommunications%29 "… *half-duplex* (HDX) system provides communication in both directions, but only one direction at a time (not simultaneously)."

Pipes (2)

- Uses Files API with exceptions
  - Creation: pipe()
  - No seeking possible
  - Read blocks when no data is available

    -> Perfect solution for producer-consumer pattern

    -> No explicit synchronization required
- Multiple readers and writers possible
  - Rules for file descriptors apply
  - Read “removes” data from buffer -> Only one obtains data
  - Not a common use case

Pipe Creation (1)

```
#include <unistd.h>
int pipe(int fd[2]);
```

- fd must be an existing array of file descriptors
- After successful call
  - fd[0]: open for reading
  - fd[1]: open for writing
- Return value
  - 0 OK
  - -1 Error

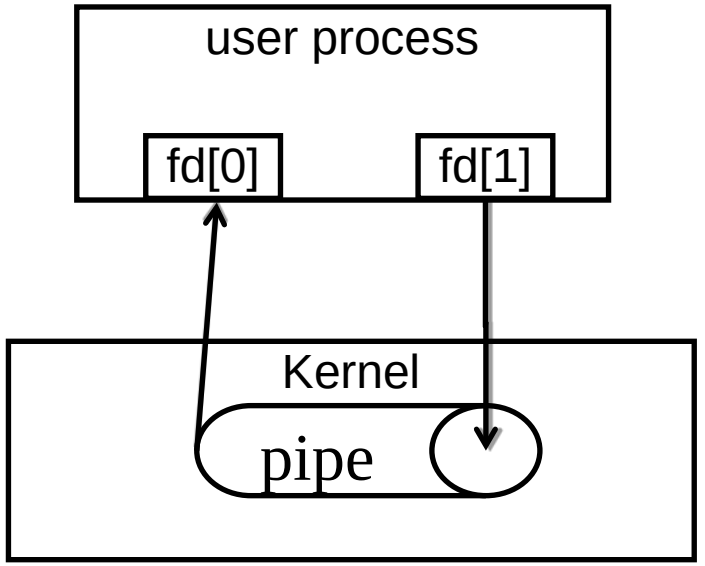
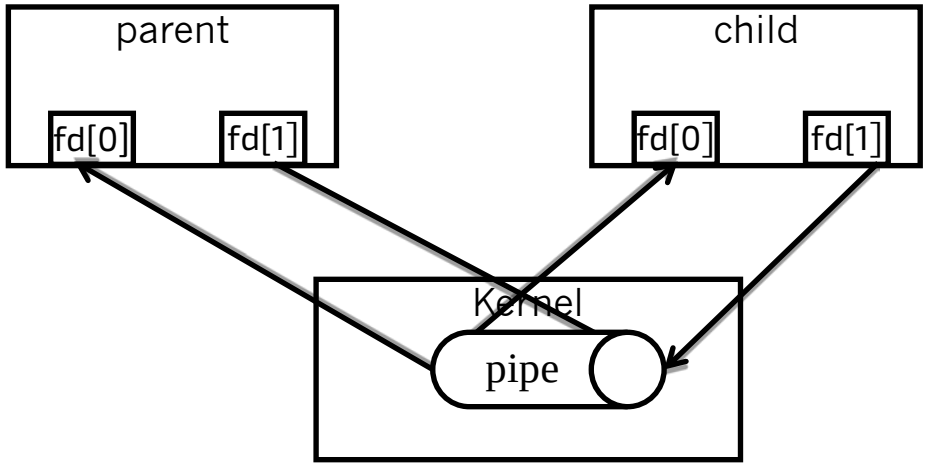


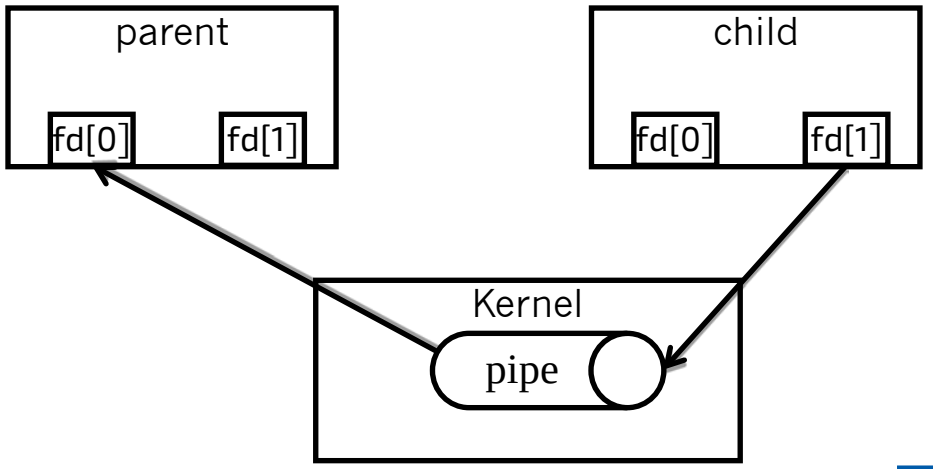
Image from [Stevens1999]

Pipe Creation (2)

- After fork



- Parent and child each close one file descriptor -> uni-directional (half-duplex) communication channel



Images from [Stevens1999]

(cf. fork() with sockets in chapter 1)

Pipe read/write

- read/write operations from file API
- All writers close pipe
  - read returns 0 (=EOF)
  - **-> Reader process must close write file descriptor**
- All readers close pipe
  - Signal SIGPIPE is generated on write

- Atomicity
  - Constant PIPE_BUF specifies kernel buffer size
  - Read/write only atomic if nbytes <= PIPE_BUF
  - PIPE_BUF can be obtained with pathconf

Pipe example – Send Data from Parent to Child

```c
int main(void) {
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
        close(fds[1]);
} else {                        /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Source

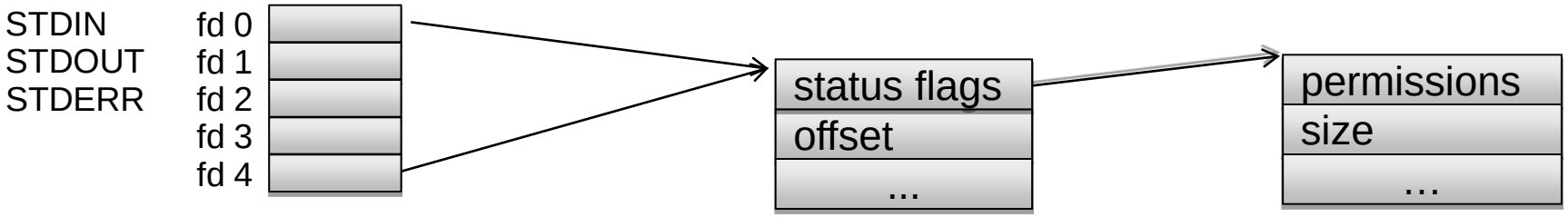
(Source from: [Stevens1999])

Redirection

- How does the shell…
  - who | sort
  - ls > filelist.txt
- Redirection of standard input/output/error

-> dup, dup2

- Duplicate existing file descriptor



dup, dup2

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

- Both functions duplicate existing file descriptor
- Returns new file descriptor, -1 in error case
- dup returns next free file descriptor; after call pointing to the same file table entry
- dup2 closes filedes2 and then copies filedes to filedes2
  - Operation is atomic -> do not replace dup2 by close();dup();
  - If filedes == filedes2 -> returns filedes2, filedes2 is not closed
  - If filedes2 is not valid -> returns -1

Example: printf to File

```c
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>

int main()
{
    int file_desc = open("tricky.txt",O_WRONLY | O_APPEND);

    // here the newfd is the file descriptor of stdout (i.e. 1)
    dup2(file_desc, 1) ;

    // All the printf statements will be written in the file
    // "tricky.txt"
    printf("I will be printed in the file tricky.txt\n");

return 0;
}
```

popen and pclose

- pipe + fork + system: common use case supported by standard library

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);
```

- Execute cmdstring as different process with system and redirect input/output
- Returns file pointer, NULL if error
- type
  - “r”: STDOUT of child is redirected; parent can read
  - “w”: STDIN of child is redirected; parent can write

```
int pclose(FILE *fp);
```

- Closes pipe and returns termination status of child

FIFOs (Named Pipes)

- "Pipes with a name"
- FIFO is created in file system
- Processes can open FIFO in read or write mode only
  - Half-duplex*
- No common ancestor between processes required
- Does not work over a network file system
  - Kernel structure
- Data discarded after last process closes

Persistence ???

FIFO Creation

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- Returns 0 for OK, -1 for error
- mode and pathname as for **open** call
- Implies O_CREAT | O_EXCL
  - FIFO will be created and error if it already exists

(cf. slide 26, files)

- User command mkfifo also exists
  - Create FIFO from command line
  - Synopsis: **mkfifo** [OPTION] NAME...

FIFO open

- Use **open** to get file descriptor for FIFO
  - either O_RDONLY or O_WRONLY
- open **blocks** until FIFO is opened for writing and reading
  - Not possible to use from one (single-threaded) process
  - Sequence of open important if multiple FIFOs are used, e.g., for full-duplex communication channel
    - **Deadlock** problem
    - Similar to synchronization traps

FIFO Reading and Writing

- Rules for **read/write operations** of PIPEs apply, especially
  - Atomic operation only if nbytes<=PIPE_BUF
  - All writers close pipe
    - read returns 0 (=EOF)
  - All readers close pipe
    - Signal SIGPIPE is generated

- Multiple readers and writers supported
  - Read “removes” data from buffer -> only one reader obtains data

FIFO Closing and Deleting

- Close function as for files
  - Process can not access FIFO anymore
  - FIFO is not removed from system
  - Data is discarded if all processes close

Persistence ???

- Removal of pipe

```
#include <unistd.h>
int unlink(const char *path);
```

- Same function as for files
  - Returns 0 for OK, -1 for error

FIFO example: Client-Server Communication

- Well known FIFO for server
- Client opens own FIFO (e.g., /tmp/$PID) and sends name and request to server

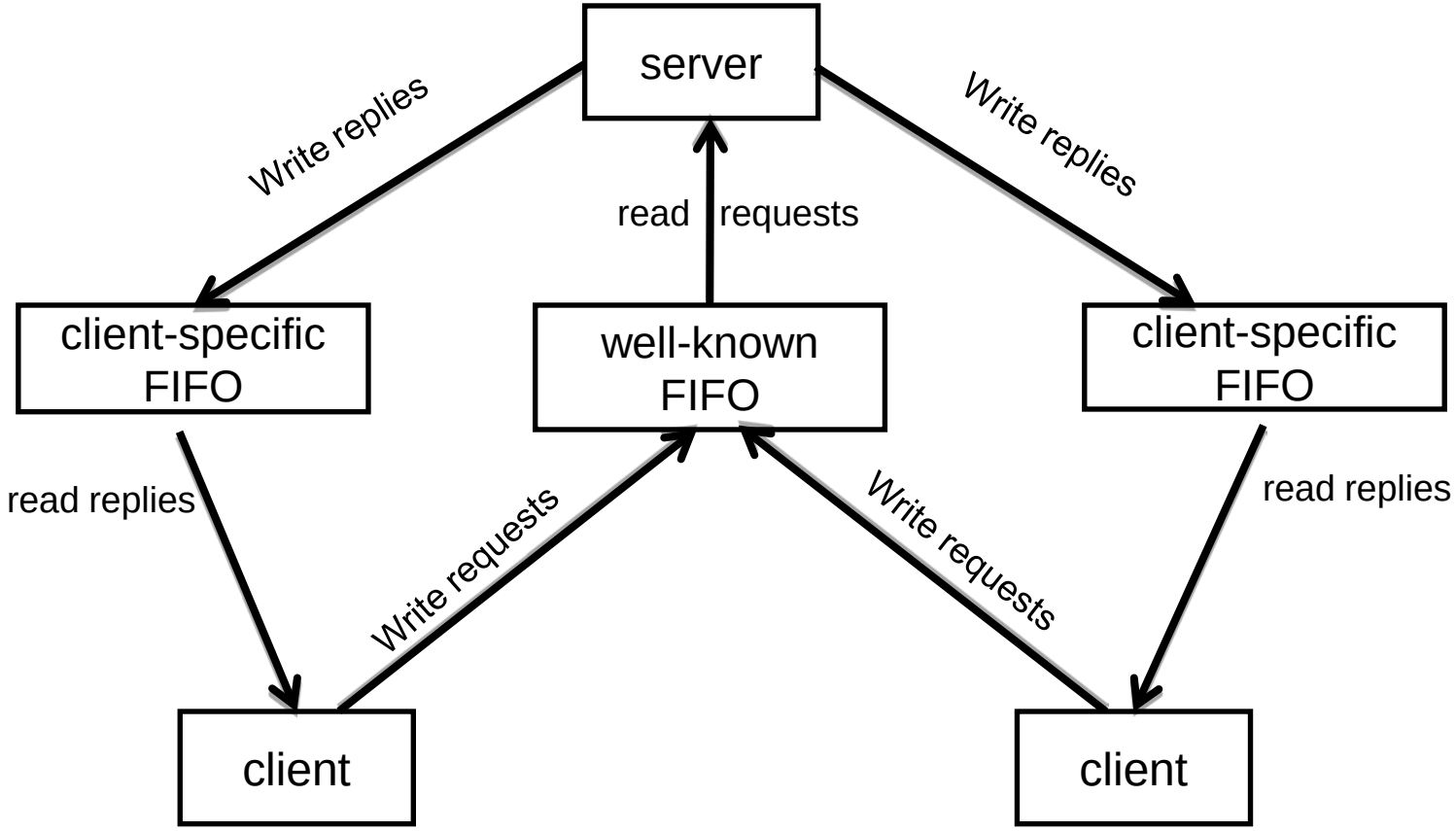


Image from [Stevens2005]

Non-Blocking Operations (1)

- Similar to non-blocking operations in socket programming
- FIFO: use flag O_NONBLOCK for open
- FIFO/pipe: use fcntl

```c
/* Set file descriptor to non-blocking */
int flags;
/* Get current flags */
if ((flags = fcntl(filedes, F_GETFL, 0)) < 0) {
        /* Handle error */
}

/* set non-blocking flag */
flags = flags | O_NONBLOCK;

if (fcntl(filedes, F_SETFL, flags) < 0) {
        /* Handle error */
}
```

```c
/* Set file descriptor to blocking */
int flags;
/* Get current flags */
if ((flags = fcntl(filedes, F_GETFL, 0)) < 0) {
        /* Handle error */
}

/* clear non-blocking flag */
flags &= ~O_NONBLOCK;

if (fcntl(filedes, F_SETFL, flags) < 0) {
        /* Handle error */
}
```

Non-Blocking Operations (2)

Operation	Existing opens of pipe or FIFO	Blocking	Non-Blocking
Open FIFO for reading	Open for writing	Returns OK	Returns OK
	Not open for writing	Blocks until opened for writing	Returns OK
Open FIFO for writing	Open for reading	Returns OK	Returns OK
	Not open for reading	Blocks until opened for reading	Returns ENXIO
Read empty pipe or FIFO	Open for writing	Blocks until data available or all writers close	Returns EAGAIN
	Not open for writing	Returns 0 (EOF)	Returns 0 (EOF)
Write to pipe or FIFO	Open for reading	Returns OK	Returns OK
	FIFO/pipe full	Blocks until space available	Returns EAGAIN
	Not open for reading	Generate SIGPIPE	Returns EPIPE

Pipes and FIFOs Summary

- Simple API based on file access
- Perfect for producer-consumer pattern
- Extremely useful for shell programming
- Half-Duplex
- Process **Persistence**
  - Data in FIFOs lost when all file descriptors are closed
- Stream-oriented (in contrast to record oriented)
- Concurrency of reader and writer required
- Reasonable support only for one reader
- Pipes: Processes require same ancestor; no name
- FIFOs: name introduced in file system

Pipes vs. Channels

- Pipes simply store buffers
  - Atomicity only guaranteed if nbytes<=PIPE_BUF
  - Partial retrieval possible

- Channels store entire records (type-safe)
  - Can be buffered and unbuffered
  - Synchronized
  - Partial retrieval of records not possible

2.3.3.2. Posix Message Queues (MQ)

- Model: linked list of messages
- Messages can be put in and taken out in FIFO order
  - Additional support for priorities
- Record oriented (in contrast to stream oriented)
  - Each record has size, data and priority
- Indirect communication
  - Processes communicate only with mailbox
  - Concurrency of readers and writers **not** required
- Kernel persistence
  - Process can store messages in queue and exit:
    Data remains until kernel reboot

Posix Message Queues API

- API completely distinct from file access API (although similar functions)
- Message queue descriptor instead of file descriptor
  - Small integer
  - Different namespace
  - Used for all access functions
- Names as described above
  - Use “/somename” for Linux and Solaris
- Names can be mapped to file system
  - Solaris: yes
  - Linux: optional mounting
- Linking of real-time library required: -lrt

Message Queues – open and create

```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

open existing MQ

create new MQ

- oflag and mode as in open for files
- attr can be NULL or pointer to message queue attributes (cf., next slide)

Message Queues – Attributes

- Attribute of message queue

```
#include <mqueue.h>
mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *attr);
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *newattr, struct mq_attr
*oldattr);

struct mq_attr {
       long mq_flags;       /* Flags: 0 or O_NONBLOCK */
       long mq_maxmsg;      /* Max. # of messages on queue */
       long mq_msgsize;     /* Max. message size (bytes) */
       long mq_curmsgs;     /* # of messages currently in queue */
 };
```

- Only mq_flags field is changed by mq_setattr

Message Queues – Sending Messages

```
#include <mqueue.h>
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
```

- Arguments similar to file write
- Returns 0 for OK, -1 for error
- Message is always completely written (or not at all)
- Message is inserted after messages with same priority and before messages with lower priority
- Message size must be smaller than mq_msgsize from attributes

Send Message Example

```c
#include "unpipc.h"

int main(int argc, char **argv)
{
        mqd_t   mqd;
        void    *ptr;
        size_t  len;
        uint_t  prio;

        if (argc != 4)
                err_quit("usage: mqsend <name> <#bytes> <priority>");
        len = atoi(argv[2]);
        prio = atoi(argv[3]);

        mqd = Mq_open(argv[1], O_WRONLY);

        ptr = Calloc(len, sizeof(char));
        Mq_send(mqd, ptr, len, prio);

        exit(0);
}
```

Error checks hidden in “uppercase function names”
(see [Stevens1999] ch. 1.6)

e.g. Mq_open(…) is a “wrapper” function that calls mq_open(…) with error checks

send „empty“ message to MQ (Calloc sets to 0)

Example from [Stevens1999]

Source

Message Queues – Receiving Messages

#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);

- Arguments similar to file read
- Returns size of message, -1 if error
- The oldest of the high priority message(s) is received
- msg_prio will receive priority of message
- msg_len must be greater or equal than mq_msgsize of attributes

If not, returns immediately with error, regardless of actual message size

- Function removes message from queue -> Other readers cannot obtain same message

Receive Message Example

```c
int main(int argc, char **argv) {
        int             c;
        mqd_t   mqd;
        ssize_t n;
        uint_t  prio;
        void    *buff;
        struct mq_attr  attr;

        if (argc != 2) err_quit("usage: mqreceive <name>");

        mqd = Mq_open(argv[1], O_RDONLY);
        Mq_getattr(mqd, &attr); /* obtain maximum message size */

        buff = Malloc(attr.mq_msgsize);

        n = Mq_receive(mqd, buff, attr.mq_msgsize, &prio);
        printf("read %ld bytes, priority = %u\n", (long) n, prio);

        exit(0);
}
```

Error checks hidden in “uppercase function names”
(see [Stevens1999] ch. 1.6)

Source

Example adapted from [Stevens1999]

Message Queues Closing and Deleting

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

- Same rules as for file closing and removal apply

Message Queues – Notification (1)

- **Common problems** when implementing a server
  - Blocking operation -> no processing while waiting possible
  - Non-Blocking operation -> wasting cycles for checking queue

→Notification when message arrives

1. Generation of signal
2. Creation of thread and execution of callback

Message Queues – Notification (2)

- Only **one** process can register for notification
- Notification when queue transition from **empty** to **non-empty** occurs
- Registration is automatically removed when notification is executed
  - Re-register (if desired) before reading messages
- If thread is waiting with blocking receive, receive is satisfied and notification is omitted as if queue stayed empty

Message Queues – Notification (3)

```
#include <mqueue.h>
int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

- Returns 0 for OK, -1 for error
- If notification==NULL, process registration is removed
- struct sigevent on next slide

Message Queues – Notification (4)

```
union sigval {                          /* Data passed with notification */
      int sival_int;                    /* Integer value */
      void *sival_ptr;                  /* Pointer value */
};
struct sigevent {
      int sigev_notify;                 /* Notification method */
      int sigev_signo;                  /* Notification signal */
      union sigval sigev_value;         /* Data passed with notification */
      void (*sigev_notify_function) (union sigval); /* Function for thread
                                                       notification */
      void *sigev_notify_attributes; /* Thread function attributes */
};
```

Message Queues – Notification (5)

- Notification method: int sigev_notify
  - **SIGEV_NONE**:

    no notification occurs

  - **SIGEV_SIGNAL**:

    signal notification->sigev_signo is sent

  - **SIGEV_THREAD**:

    new thread with function notification->sigev_thread_function is created

Message Queues – Example (1/2)

```c
#define die(msg) { perror(msg); exit(EXIT_FAILURE); }
static void tfunc(union sigval sv);     /* forward declaration */

int main(int argc, char *argv[]) {
        mqd_t mqdes;
        struct sigevent not;

    (*) assert(argc == 2);

        mqdes = mq_open(argv[1], O_RDONLY);
        if (mqdes == (mqd_t) -1) die("mq_open");

        not.sigev_notify = SIGEV_THREAD;
        not.sigev_notify_function = tfunc;
        not.sigev_notify_attributes = NULL;
        not.sigev_value.sival_ptr = &mqdes;    /* Arg. to thread func. */
        if (mq_notify(mqdes, &not) == -1) die("mq_notify");

        pause();    /* Process will be terminated by thread function */
}
```

- Register notification with command line argument (argv[1] is MQ name)
- Perform notification by creation of new thread

(*) assert - abort the program if assertion is false

#include <assert.h>
void assert(scalar expression);

Source

(Adapted from: man page of mq_notify)

Message Queues – Example (2/2)

```c
static void tfunc(union sigval sv) { /* Thread start function */
        struct mq_attr attr;
        ssize_t nr;
        void *buf;
        mqd_t mqdes = *((mqd_t *) sv.sival_ptr);

        /* Determine max. msg size; allocate buffer to receive msg */

        if (mq_getattr(mqdes, &attr) == -1) die("mq_getattr");
        buf = malloc(attr.mq_msgsize);
        if (buf == NULL) die("malloc");

        nr = mq_receive(mqdes, buf, attr.mq_msgsize, NULL);
        if (nr == -1) die("mq_receive");

        printf("Read %ld bytes from MQ", (long) nr);
        free(buf);
        exit(EXIT_SUCCESS);          /* Terminate the process */
}
```

- Output message
- Terminate process

Source

(Adapted from: man page of mq_notify)

Message Queues – Summary

- Indirect communication
- Kernel persistence
- Supports producer-consumer pattern
- Supports client-server pattern
  - Limited support for multiple readers
- Full-duplex
- Record oriented access
- Notifications
  - Efficient server implementations possible
  - Re-registration necessary

3.2.3.3. Sockets

- Will be discussed in detail in Chapter 3
- Process **persistence**
- Full-duplex
- Record or stream oriented
- Networking support
- Hostname + port as identifier
- Also usable on a single system

- Specific for single systems: Unix domain Sockets (in contrast to internet domain sockets)
  - No protocol overhead -> increased efficiency

Unix Domain Sockets

- Identical API to internet domain socket
- Different naming scheme: sockaddr_un instead of sockaddr_in

```
struct sockaddr_un {
        sa_family_t sun_family;                 /* AF_UNIX */
        char sun_path[108];                     /* pathname */
};
```

- Name will be mapped to file system, but can't be opened

Persistence ???

Unix Domain Sockets – Example “bind”

```c
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int main(void) {
    int fd, size;
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket failed");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("bind failed");
    printf("UNIX domain socket bound\n");
    exit(0);
}
```

“apue.h”

comes from http://www.apuebook.com/
Resources to Stevens et al. Advanced Programming in the Unix Environment

Source

2.3.4. Posix Shared Memory

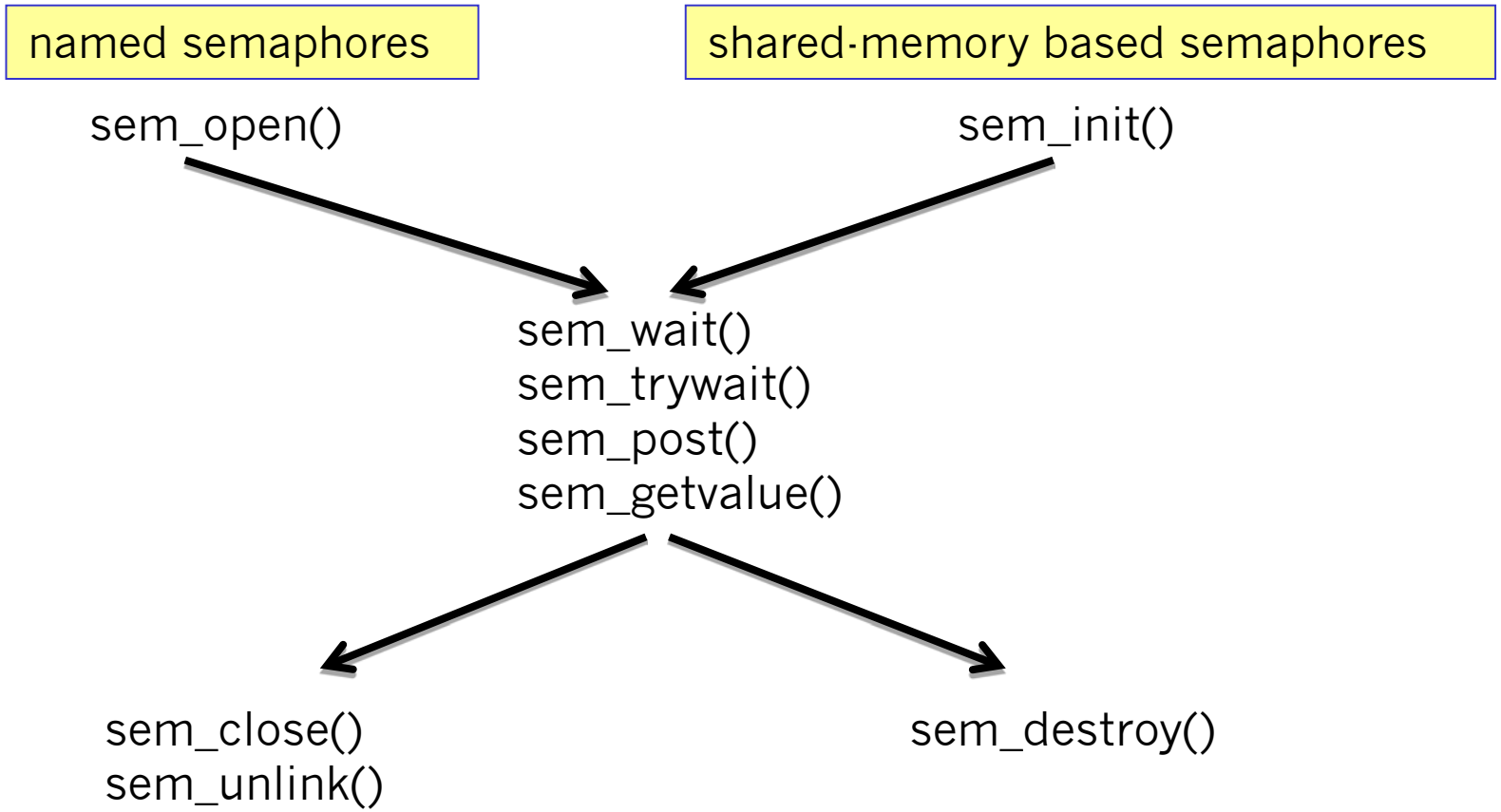
- Shared memory: identical to shared memory in threads
- More efficient than message passing mechanisms
  - Data does not have to be moved through kernel
- Explicit synchronization from processes required
  - Identical to threading challenges
  - IPC synchronization primitive semaphore discussed later
- Multiple possibilities for shared memory
  - Posix shared memory
  - Memory-mapped file
  - /dev/zero anonymous memory-mapping
  - Kernel **persistence**

2.3.5. IPC Synchronization: Posix Semaphores

- Semaphore concept discussed in threading chapter
- Two flavors
  - Shared-memory based: discussed in threading chapter
  - Named: discussed below
- Both flavors can be used between processes
  - Shared-memory semaphore requires memory sharing (cf. 2.3.4.)
- Named Semaphores
  - Semaphore with well-known name can be used to synchronize processed that do not know each other
  - Can be used to protect process-external resources
- Same rules for names as for Posix message queue

Posix Semaphores API

- With the exception of creation/removal, same API for memory based and named semaphores



Posix Semaphores – open

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

open existing named sem

create new named sem

- Arguments equal to mq_open
- value: initial value
- Returns pointer to new semaphore, SEM_FAILED in case of error

Posix Semaphores – Closing and Removal

```
#include <semaphore.h>
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

- Same rules as for message queues apply

- sem_close has no effect on value of semaphore

  -> Deadlock possibility when process exits!

2.3.6. Signals – In Depth (1)

- Signals are asynchronous notifications
- Common use cases of explicit notifications from other processes
  - Terminate process
  - Pause/continue process
  - Inform daemon to re-read configuration file
- Common use cases of system notifications
  - Error notification: SIGSEGV, SIGPIPE, …
  - Notification of process events
    - Death of controlling process: SIGHUP
    - Death of child: SIGCHLD
  - Timer expired
  - Interruption of system call

Signals – In Depth (2)

- Normally, when a process receives a signal, it
  - first stops the current execution,
  - handles the signal, and then
  - resumes the original execution.

- Each signal type is specified by a signal number

- For most signal types, a program can
  - respond to the signal by registering a signal-handler function
  - ignore the signal

Unreliable Signals

- Signal handling in older versions of UNIX unreliable
- Signals could get lost
- Signal handler automatically deregistered after signal call
  - Re-registration in signal handler if desired
  - Short window between call of signal handler and re-registration exists
    - Default action taken, e.g., process termination
    - Signal gets lost

- Old API: signal()
  - Implementation defined if unreliable signals are used
  - **-> Don't use this!**

- On the following slides: reliable signals according to POSIX specification

Signals – Terminology

- Signal is *generated* (or *sent* to a process) when event occurs
- Signal is *delivered* if associated action is taken
- Signal is *pending* in between

- Signal action (also called disposition) can be
  - *default action* (often process termination)
  - *catch*, i.e., custom handler
  - *ignore*
- Signal delivery can be *blocked*
  - Only delivery, not generation can be blocked
  - Signal is pending until unblocked or action changed to ignore
  - Set of *blocked* signals is called *signal mask*

Signals – Generation of Multiple Signals

- One signal is generated multiple times while pending
  - Behavior is implementation dependent

  1. Signal is delivered multiple times (queuing)
  2. Signal is delivered only once

  - Most implementations deliver only once

- Multiple different signals are generated while pending
  - Order of delivery unspecified
  - Signal handler can be interrupted by another signal!

Signals – Interrupted System Calls

- System calls can return with error EINTR when signal is generated during execution

- Some implementations automatically restart system calls
  - Implementations that conform to the XSI extensions do **not** restart
  - Flag SA_RESTART for explicit specification in sigaction

Signals – Reentrant Functions

- If execution continues after signal handler, function calls in handler must be reentrant

- Similar to thread-safety requirements

- For list of reentrant functions see Single UNIX Specification
  - File and network IO
  - Signal functions

- **Never use malloc() and free() or dependent functions**

- Reentrancy also important for custom functions

Signals – Generating

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

- Return 0 for OK, -1 on error
- raise(signo) is equal to kill(getpid(), signo);
- Signal is sent if permissions allow
  - pid>0: send to process with ID==pid
  - pid==0: send to process group of current process
  - pid<0: send to process group abs(pid)
  - pid=1: send to all processes

Signals – Catching (1)

- Call of user-defined function
- Signal is blocked for additional delivery during execution of handler
  - Can be disabled with SA_NODEFER
- Signal mask: automatically block selected signals during execution of handler
- System calls are not automatically restarted (XSI conformance)
  - Can (and should) be enabled with flag SA_RESTART
- Emulate unreliable signal behavior with flag SA_RESETHAND
  - Signal action automatically reset to default when invoking handler

Signals – Catching (2)

- Signal handler prototype

void handler(int signo);

- Alternative signal handler, enabled with flag SA_SIGINFO

void handler(int *signo*, siginfo_t **info*, void **context*);

- info: additional information about the signal
  - Contents only partially standardized
  - Posix requirements next slide
- context: process context when signal was generated
  - Contents mostly opaque
  - Beyond this lecture

Signals Catching – Information for Handler

```
struct siginfo {
        int     si_signo;     /* signal number */
        int     si_errno;     /* if nonzero, errno value from <errno.h> */
        int     si_code;      /* additional info (depends on signal) */
        pid_t   si_pid;       /* sending process ID */
        uid_t   si_uid;       /* sending process real user ID */
        void    *si_addr;     /* address that caused the fault */
        int     si_status;    /* exit value or signal number */
        long    si_band;      /* band number for SIGPOLL */
        /*possibly other fields also */
};
```

- Examples for si_code on next slide

Signals Catching – si_code Example Values

Signal	Code	Reason
SIGILL Illegal Instruction	ILL_ILLOPC ILL_ILLOPN ILL_ILLADR ILL_ILLTRP ILL_PRVOPC ILL_PRVREG ILL_COPROC ILL_BADSTK	illegal opcode illegal operand illegal addressing mode illegal trap privileged opcode privileged register coprocessor error internal stack error
SIGFPE Floating Point Exception	FPE_INTDIV FPE_INTOVF FPE_FLTDIV FPE_FLTOVF FPE_FLTUND FPE_FLTRES FPE_FLTINV FPE_FLTSUB	integer divide by zero integer overflow floating-point divide by zero floating-point overflow floating-point underflow floating-point inexact result invalid floating-point operation subscript out of range
SIGPOLL Asynchronouse IO	POLL_IN POLL_OUT POLL_MSG POLL_ERR POLL_PRI POLL_HUP	data can be read data can be written input message available I/O error high-priority message available device disconnected

Signals – Modify Signal Handler

#include <signal.h>

int sigaction(int *signo*, const struct sigaction **act*, struct sigaction **oact*);

- If act !=NULL, action is changed
- If oact != NULL, old action is copied

```
struct sigaction {
void          (*sa_handler)(int); /* addr of signal handler, */
                                  /* or SIG_IGN, or SIG_DFL */
sigset_t      sa_mask;            /* additional signals to block during handler */
int           sa_flags;           /* signal options; examples discussed above*/
/* alternate handler */
void          (*sa_sigaction)(int, siginfo_t *, void *);
};
```

- Values as explained above
- SIG_IGN: ignore signal, SIG_DFL: reinstate default action

Signal Handler Example

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

int main (int argc, char *argv[])
{
struct sigaction act;

memset (&act, '\0', sizeof(act));

/* Use the sa_sigaction field because the handles has two additional parameters */
act.sa_sigaction = &hdl;

/* The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field, not sa_handler. */
act.sa_flags = SA_SIGINFO;

if (sigaction(SIGINT, &act, NULL) < 0) {
        perror ("sigaction");
        return 1;
}

while (1)
        sleep (10);

return 0;
}
```

```c
static void hdl (int sig, siginfo_t *siginfo, void *context)
{
printf ("Sending PID: %ld, UID: %ld\n",
                (long)siginfo->si_pid, (long)siginfo->si_uid);
}
```

Blocking Signals (1)

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

- Change or query signal mask

- sigprocmask only for single-threaded processes
  - Behavior undefined in multi-threading

- pthread_sigmask is thread safe
  - Signals and threads details follow

Blocking Signals (2)

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

- *how*:
  - **SIG_BLOCK**: additionally block specified signals
  - **SIG_SETMASK**: replace old mask by new mask
  - **SIG_UNBLOCK**: unblock specified signals

- set == NULL -> no modification, how is irrelevant
- oset != NULL -> old signal mask is copied

} Use for querying

Blocking Signals (3)

- Only signals that can be ignored can also be blocked
  - not SIGKILL and SIGSTOP

- Blocking of SIGFPE, SIGILL, SIGSEGV or SIGBUS
  - Programming errors; execution cannot be continued
  - Behavior undefined unless explicitly generated by kill()

Pending Signals

```
#include <signal.h>
int sigpending(sigset_t *set);
```

- Obtain the set of pending signals

Signal Sets

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

- All return 0 for OK, -1 on error

```
int sigismember(const sigset_t *set, int signo);
```

- Returns 1 for true, 0 for false, -1 for error

Signals Example – Print Signal Mask

```
void pr_mask(const char *str) {
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;       /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");
    /* remaining signals can go here */

    printf("\n");
    errno = errno_save;
}
```

string parameter …

… will be printed before list of signals in signal mask

(Adapted from: [Stevens1999])

Source

Waiting for Signals

```
#include <unistd.h>
int pause(void);

#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

- Both functions suspend thread until signal with handler is caught or signal that terminates the process
- Always return -1 for Error (errno=EINTR: system call interrupted)
- sigsuspend
  - Replaces current signal mask with sigmask
  - Signal mask is restored after return
  - Atomic operation!

Signals – Wait for Queued Signals

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

- Selects an *already pending* signal from set and returns
- Returns 0 if OK, -1 on error
- Signal number is stored in *sig
- Original signal action is **not** taken (contrary to sigsuspend)
- Signals in set must be part of signal mask (blocked)
- Call blocks if no signal is pending
- If multiple signals are in set and pending, order is not defined

Signals and Threads

- Disposition is valid for **all threads** of a process
- Mask is valid for each thread **separately**
  - Inherited at thread creation
- Synchronously generated signals
  - Generated by thread execution itself, e.g., SIGSEGV, …
  - Is handled by the generating thread **if not masked**
    - If masked, behavior from asynchronous signals apply
- Asynchronously generated signals
  - Generated from outside the process, e.g. SIGTERM
  - Behavior next slide

Asynchronous Signals and Threads

- A signal is handled by **exactly one** thread if caught
- Thread must

1. Be blocked in sigwait for the signal
2. Does not block the signal (is not in signal mask)

- Sigwait has preference

- If multiple threads are eligible, selected thread is implementation dependent

- If no thread is eligible, signal is pending

Signals & Threads – Dedicated Signal Handler

- Common pattern: dedicated signal handler thread
  - Easy to understand
  - Removes complexity from actual worker threads
- Pattern:
  - All threads block all signals
  - Dedicated handler thread uses sigwait() function
  - Additional advantage: sigwait() allows synchronous programming
  - Easier to understand and write correctly

```
thr_sigsetmask(mask);
while( (signo = sigwait(mask)) > 0) {
      /*handle signal type */
}
```

Inter-thread Signalling

```
#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
```

- Signal *sig* is delivered to thread thread
- Returns 0 if OK, -1 on error
- If sig is 0
  - Error checking is performed
  - No signal is sent
  - Can be used to check validity of thread argument

Signals in Shell Programming

- Signals can also be caught in shell scripts
- Important for cleanup, e.g., of temporary files
- Use: trap command signals
- command=- for removing trap

```
#!/bin/bash
# traptest.sh

trap "echo Booh!" SIGINT SIGTERM
echo "pid is $$"

while :                 # This is the same as "while true".
do
        sleep 60        # This script is not really doing anything.
done
```

(Source from: Bash Guide for Beginners)

Signals – Summary

- Efficient solution for event notification
- Can be used for process synchronization
- Old implementation/specification troublesome
  - Unreliable
  - Automatic de-registration of signal handler
  - When reading books/tutorials: make sure the new type is covered
    - Telltale signs: sigaction
- Signals and Threads
  - Disposition per process
  - Mask per thread

Interprocess Communications – Summary

- Various ways for interaction between processes
- Various persistence schemes
- Various solutions for producer-consumer pattern

- Powerful IO API
  - For files, pipes, FIFOs and limited for networking

- Processes vs. threads
  - Sophisticated solutions for communication vs. shared memory only
  - General synchronization primitive vs. sophisticated solutions

IPC - Summary (2)

- What did we learn in this section?
  - *„Everything is a file“* (2.3.2.)
  - some details on files and File API
  - **Pipes and FIFOS** (named pipes) => **File API** partly used (2.3.3.1.)
  - Message Queues=>API completely distinct from File API (but similar functions) (2.3.3.2.)
  - Sockets (ref. to ch. 1) – Unix Domain Sockets (short) (2.3.3.3.)
  - Shared Memory (short) (2.3.4.)
  - IPC semaphores (short) (2.3.5.)
  - **Signals** – in Depth (2.3.6.)

2.3. IPC – Outline

2.3.1. Basics

2.3.2. Files

file IO – as probably already known from basic programming

2.3.3. Message Passing

- .1 Pipes and FIFOs (Named Pipes)
- .2 Message Queues
- .3 Sockets

sockets for network communication as discussed in ch. 3.

2.3.4. Shared Memory

2.3.5. Semaphores

semaphores with processes as already mentioned in ch. 2.2.

2.3.6. Signals

Persistence of various types of IPC objects

Type of IPC	Persistence
Pipe	process
FIFO (named pipes)	process
Posix mutex	process
Posix condition variable	process
Posix read-write lock	process
fcntl record locking	process
Posix message queue	kernel
Posix named semaphore	kernel
Posix memory-based semaphore	process
Posix shared memory	kernel
System V message queue	kernel
System V semaphore	kernel
System V shared memory	kernel
TCP socket	process
UDP socket	process
Unix domain socket	process

from [Stevens1999]