System-Programmierung 8: IPC mit Pipes

Ablauf heute

⅔ Vorlesung,

⅓ Hands-on,

Feedback.

Slides, Code & Hands-on: tmb.gr/syspr-8

Interprozesskommunikation

Interprozesskommunikation (IPC) umfasst sowohl Kommunikation & Synchronisation als auch Signale.

Es geht um Datenaustausch zwischen Prozessen oder Threads, und das Synchronisieren ihrer Aktionen.

Wir betrachten *POSIX* Interprozesskommunikation, nicht die ältere, komplexere *System V* IPC Variante.

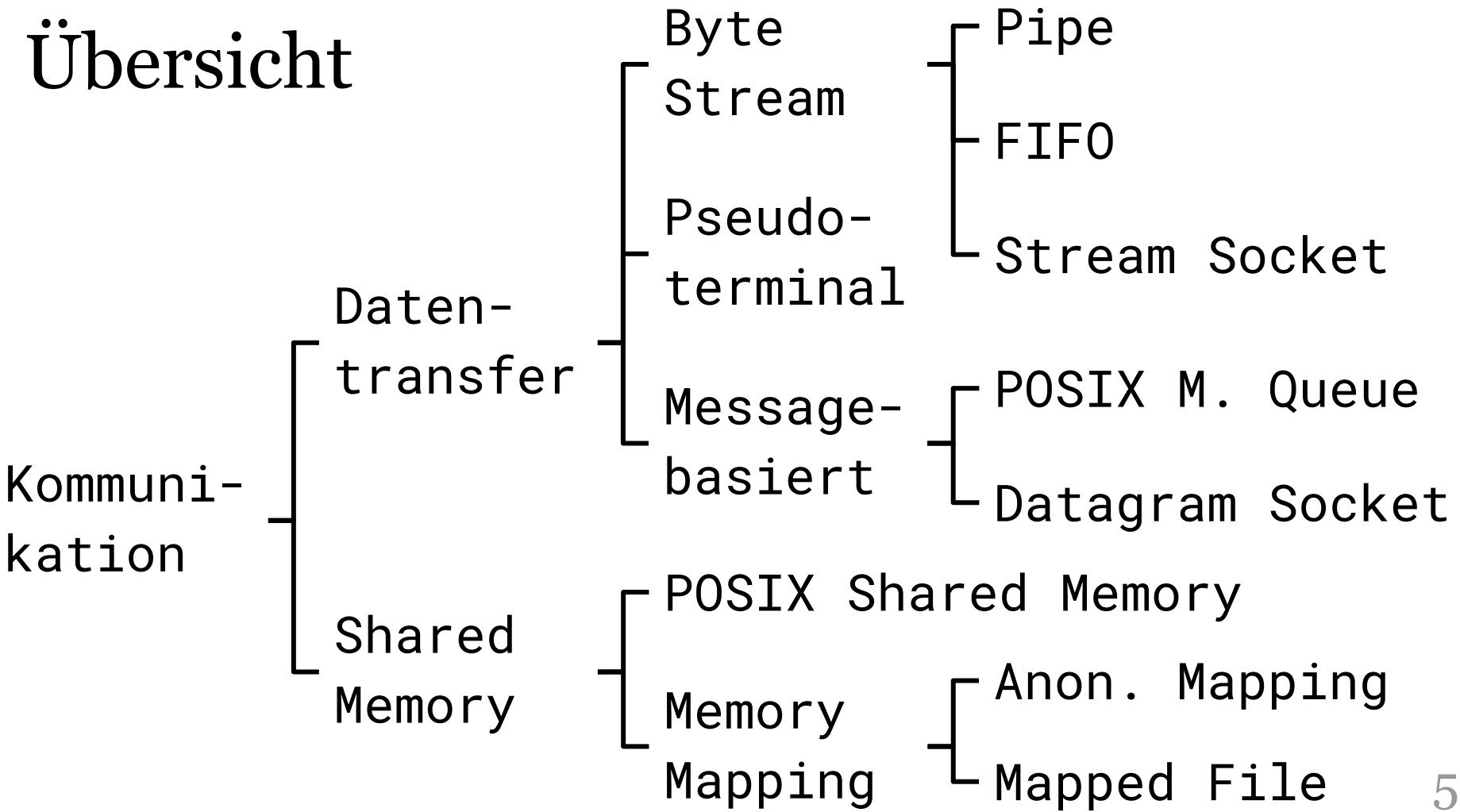
Kommunikation

Kommunikations-Mechanismen erlauben Prozessen (und im Prinzip auch Threads) den Datenaustausch.

Eine Kategorie sind *Datentransfers*, die mit *read()* und *write()* System Calls via Kernel kommunizieren.

Die zweite Art zu kommunizieren ist mittels *Shared Memory*, direkt im Speicher, ohne System Calls und daher wesentlich schneller.

Übersicht


Kommuni-
kation
Daten-
transfer
Byte
Stream
Pipe
FIFO
Pseudo-
terminal
Stream Socket
Message-
basiert
POSIX M. Queue
Datagram Socket
Shared
Memory
POSIX Shared Memory
Memory
Mapping
Anon. Mapping
Mapped File

Datentransfer

Datentransfer kann via Byte Stream, Message-basiert oder mit spezialisierten *Pseudoterminals* geschehen.

Bei allen davon sind Leseoperationen destruktiv, d.h. konsumierte Daten sind nicht für Andere verfügbar.

Synchronisation geschieht automatisch, wenn keine Daten verfügbar sind blockiert der lesende Prozess.

Datentransfer via Byte Stream

Daten, die via Pipes, FIFOs, und Stream Sockets ausgetauscht werden sind ein unbegrenzter *Byte Stream*.

Jede Leseoperation kann eine beliebige Anzahl Bytes lesen, unabhängig von der Blockgrösse bei *write()*.

Dieses Modell entspricht dem bekannten UNIX Modell eines "Files als Sequenz von Bytes".

Message-basierter Datentransfer

Daten, die via POSIX Message Queues und Datagram Sockets übertragen werden, sind klar abgegrenzt.

Jede Leseoperation liest eine ganze Message, wie sie vom schreibenden Prozess geschrieben wurde.

Es ist nicht möglich, nur einen Teil einer Message zu lesen, und den Rest später abzuholen. Man liest auch nie mehr als eine Message auf's mal.

Shared Memory

UNIX Systeme bieten POSIX (und System V) *Shared Memory* und *Memory Mappings*, mehr dazu später.

Shared Memory ist zwar schnell, aber man muss den Zugriff darauf synchronisieren, was ein Nachteil ist. Meistens werden dazu Semaphoren verwendet.

Shared Memory ist für alle Prozesse sichtbar, die sich den Speicher teilen, das Lesen ist nicht destruktiv.

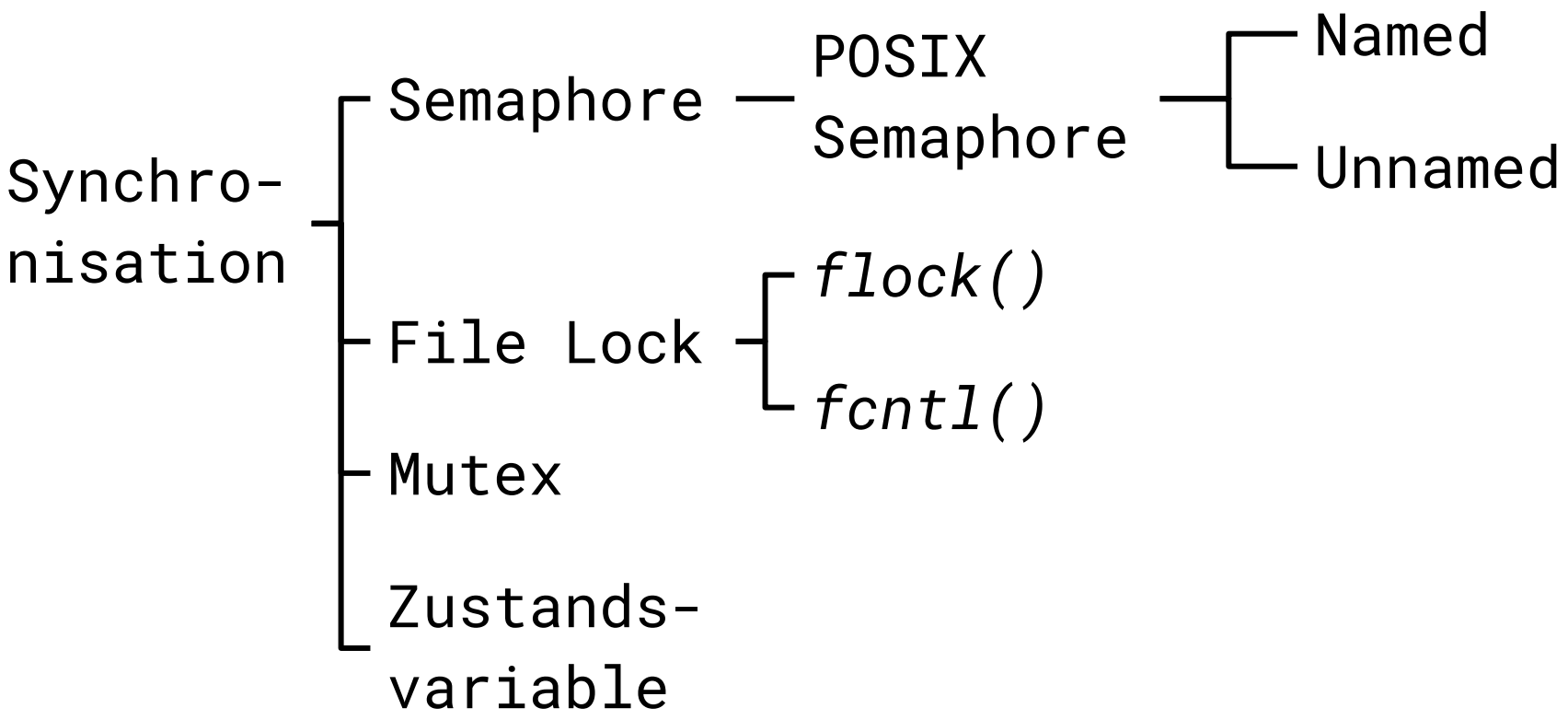
Synchronisation

Synchronisation erlaubt Prozessen ihre Aktionen zu koordinieren, um Inkonsistenzen zu verhindern, z.B. beim gleichzeitigen Update einer Speicherstelle.

Für Dateizugriffe werden oft *File Locks* benutzt, für andere geteilte Ressourcen eher *Semaphoren*.

Kommunikationsmechanismen können auch zur Synchronisation verwendet werden, z.B. Pipes.

Übersicht


Synchro-
nisation
Semaphore
POSIX
Semaphore
Named
Unnamed
File Lock
flock()
fcntl()
Mutex
Zustands-
variable

Synchronisation mit Semaphoren

Ein *Semaphor* ist eine Zahl deren Wert nicht unter `0` fallen kann. Beim Dekrementieren eines Semaphors das `0` ist, wird der Aufrufer vom Kernel blockiert.

Sobald ein anderer Prozess das Semaphor wieder um eins erhöht, kann der blockierte Prozess weiterlaufen.

Semaphoren sind oft binär, mit Werten `0` und `1`, je nach Anwendung kann der Wert auch höher sein.

Synchronisation mit File Locks

File Locks koordinieren gleichzeitige File-Zugriffe.

Read Locks werden geteilt, *Write Locks* sind exklusiv.

Wenn ein Prozess ein *Write Lock* auf ein File hat, darf kein anderer Prozess ein *Write* od. *Read Lock* haben.

Linux bietet dazu die *flock()* und *fcntl()* System Calls, wobei *fcntl()* den Zugriff auf einzelne *Records* regelt.

Mutex und Zustandsvariablen

Ein *Mutex* schützt den Zugriff auf geteilte Ressourcen.

Zustandsvariablen (condition variables) erlauben es Threads, sich gegenseitig über Zustandsänderungen einer gemeinsam genutzten Ressource zu informieren.

Diese Mechanismen werden oft im Zusammenhang mit Threads verwendet und wurden bereits behandelt.

Vergleich von IPC Mechanismen

IPC Mechanismen unterscheiden sich durch:

Art der "Handles" um den Mechanismus zu nutzen.

Read/Write Datentransfer vs. Shared Memory Modell.

Netzwerk-Kommunikation, UNIX vs. Internet Sockets.

Portierbarkeit, Zugriffsrechte, Persistenz, Performanz.

Einige dieser Aspekte schauen wir genauer an.

IPC Identifiers und Handles

Um mit einem IPC Objekt zu arbeiten braucht es einen *Identifier*, z.B. Pfadname oder Adresse, und nach dem öffnen ein *Handle*, z.B. File Deskriptor oder Pointer.

Manche Mechanismen funktionieren ohne Identifier.

Nachfolgend eine Tabelle zur Übersicht.

IPC Identifiers und Handles

Pipe	Kein Name	File Deskriptor
FIFO	Pfadname	File Deskriptor
UNIX Socket	Pfadname	File Deskriptor
Internet Socket	IP Adresse und Port	File Deskriptor
Message Queue	IPC Pfadname	*mqd_t*
Semaphore	Kein / IPC Pfadname	*sem_t* *
Shared Memory	IPC Pfadname	File Deskriptor
M. Mapped File	Pfadname	File Deskriptor
flock(), fcntl()	Pfadname	File Deskriptor

IPC Zugriffsrechte und Persistenz

Für IPC Objekte im Filesystem gelten die normalen File Zugriffsrechte für *owner*, *group* und *other*.

Manche IPC Objekte sind nur für *verwandte* Prozesse zugänglich, wobei verwandt "per *fork()*" gemeint ist.

Für ein *File Lock* braucht es "File *open()*" Rechte.

Für Internet Sockets gibt es keine Einschränkung.

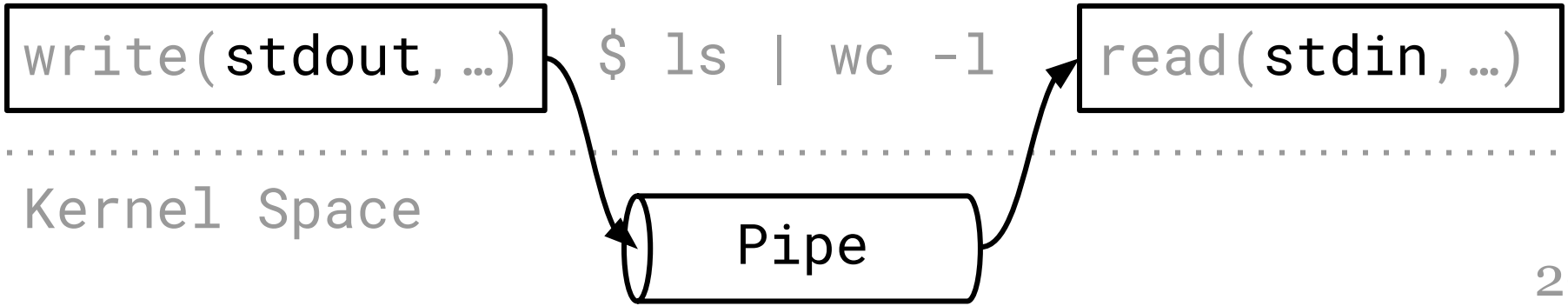
IPC Zugriffsrechte und Persistenz

Pipe	Verwandte Prozesse	Prozess
FIFO	File Zugriffsrechte	Prozess
UNIX Socket	File Zugriffsrechte	Prozess
Internet Socket	Jeder Prozess	Prozess
Message Queue	File Zugriffsrechte	Kernel
Semaphore	File Zugriffsrechte	Kernel
Shared Memory	File Zugriffsrechte	Kernel
M. Mapped File	File Zugriffsrechte	File System
flock(), fcntl()	File *open()* Rechte	Prozess

Pipes

Pipes erlauben der Shell, Output und Input von zwei miteinander verwandten Prozessen zu verbinden.

Pipes sind unidirektionale Byte Streams vom *write()* zum *read()* Ende und haben eine begrenzte Kapazität.



Pipes sind Byte Streams

Daten können in beliebiger Blockgrösse in eine Pipe geschrieben, und am anderen Ende gelesen werden.

Lesen geht nur sequentiell, *lseek()* ist nicht möglich. Wenn keine Daten verfügbar sind, blockiert *read()*, nach dem Schliessen der Pipe liefert *read()* = `0`, *EOF*.

Daten können nur unidirektional, in eine Richtung fliessen, vom *read()* zum *write()* Ende der Pipe.

Pipes haben begrenzte Kapazität

Eine Pipe ist einfach ein Buffer im Kernel Speicher, und hat deshalb eine max. Kapazität (Linux: 64 KB).

Wenn eine Pipe voll ist, blockieren *write()* Aufrufe, bis am anderen Ende mit *read()* etwas gelesen wird.

Aus Effizienzgründen wird ein relativ grosser Buffer benutzt, da eine volle Pipe einen Kontext-Switch im Kernel erfordert, um den Lese-Prozess zu wecken.

Pipes kreieren mit *pipe()*

Der *pipe()* Aufruf kreiert eine Pipe von *fd[1]* zu *fd[0]*:

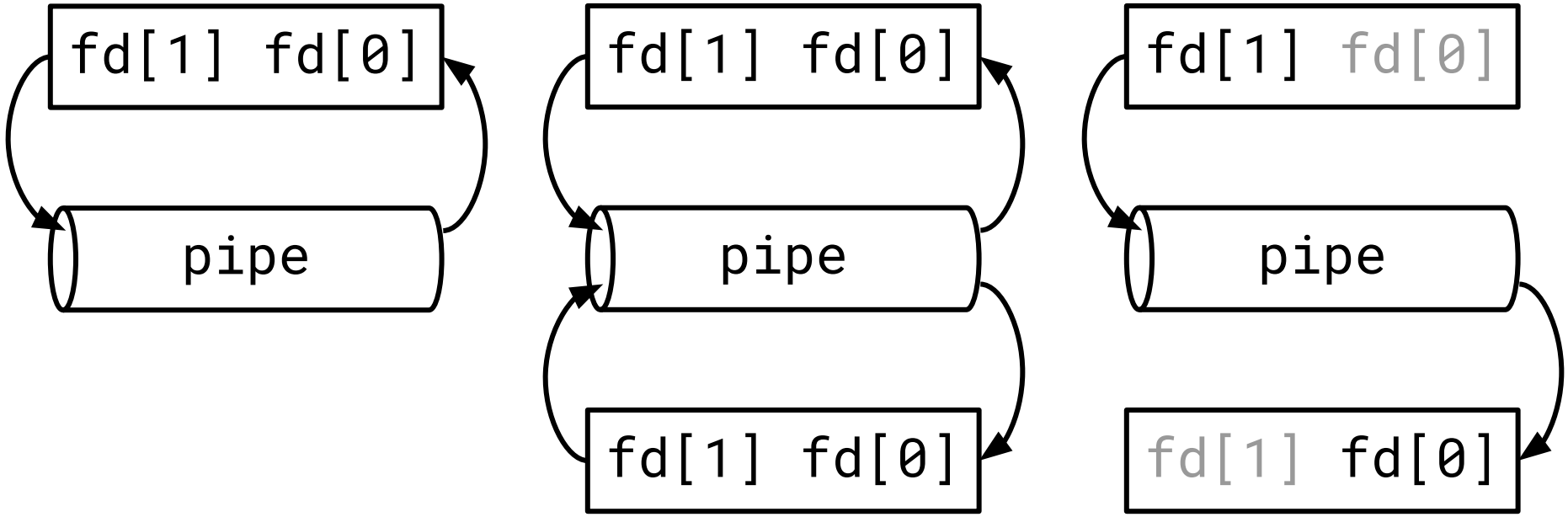
```
int pipe(int fd[2]); // 0 oder -1, errno
```

Üblicherweise kommt nach *pipe()* direkt ein *fork()*.

Das Array *fd* enthält *nach* dem Aufruf 2 Deskriptoren, Child und Parent Prozess schliessen je einen davon.

So entsteht eine Pipe vom Parent zum Child, oder umgekehrt, vom Child zum Parent Prozess.

pipe(), fork(), close() simple_pipe.c TLPI



Das *close()* ist wichtig für korrektes Funktionieren.

Pipes schliessen mit *close()*

Ist der schreibende Prozess fertig, muss dieser *close()* aufrufen, dass am anderen Ende *EOF* erscheint (wenn alle *fd* zu sind), sonst würde *read()* ewig blockieren.

Wenn nach einem "*read()*-seitigen" *close()* weiterhin auf eine Pipe geschrieben wird, sendet der Kernel das *SIGPIPE* Signal. Falls das Signal ignoriert wird, gibt *write()* den *EPIPE* "broken pipe" Fehler zurück.

Hands-on, 15': Pipes

pipe.!c

Schreiben Sie ein Programm *my_pipe.c*, welches eine Pipe von Child zu Parent Prozess erstellt.

Der Child Prozess soll die Nachricht "hello" an den Parent Prozess schicken, über diese Pipe.

Synchronisieren mit Pipes pipe_sync.c TLPI

Parent kreiert Pipe und forkt mehrere Child Prozesse.

Jedes Child schliesst das *read()* Ende der Pipe sofort, das *write()* Ende der Pipe, wenn die Arbeit fertig ist.

Der Parent Prozess blockiert beim *read()*, bis nach dem Schliessen des letzten Deskriptors *EOF* kommt.

Das *write()* Ende des Parents muss auch zu sein.

Filter verbinden mit Pipe pipe_ls_wc.c TLPI

Ein *Filter* liest von *stdin* und schreibt auf *stdout*. Mit folgenden Schritten verbindet die Shell zwei Filter:

```
pipe(fd); // Parent, "sh", Pipe in "ls | wc -l"
// fork(); … Child 1, "ls", schreibt auf stdout
dup2(fd[1], STDOUT_FILENO); // fd[1] umbiegen
close(fd[1]); // fd[1] ist unbenutzt
// fork(); … Child 2, "wc -l", liest von stdin
dup2(fd[0], STDIN_FILENO); // fd[0] umbiegen
close(fd[0]); // fd[0] ist unbenutzt
```

FIFOs

FIFOs sind eine Variation von Pipes. Sie erlauben den Datentransfer zwischen zwei beliebigen Prozessen.

Daten fliessen unidirektional, "first in first out".

FIFOs werden mit *mkfifo(filepath, ...)* kreiert und genau wie reguläre Files mit *open()* geöffnet.

Daher sind FIFOs auch als *Named Pipes* bekannt.

FIFO kreieren mit *mkfifo()*

Der *mkfifo()* Aufruf kreiert ein neues FIFO File:

```
int mkfifo(const char *filepath, mode_t mode);
```

Der *mode* Parameter ist derselbe wie bei File *open()*.

Das Öffnen mit *open()* und *O_RDONLY* blockiert, bis ein zweiter Prozess *open()* mit *O_WRONLY* aufruft.

Öffnen von FIFO mit *O_RDWR* ist nicht definiert.

FIFOs synchronisieren Lese- und Schreib-Prozess.

Hands-on, 15': FIFO

fifo_r.!c, _w.!c

Schreiben Sie ein Programm *my_fifo_r.c*, das ein neues FIFO File erzeugt, und zum Lesen öffnet.

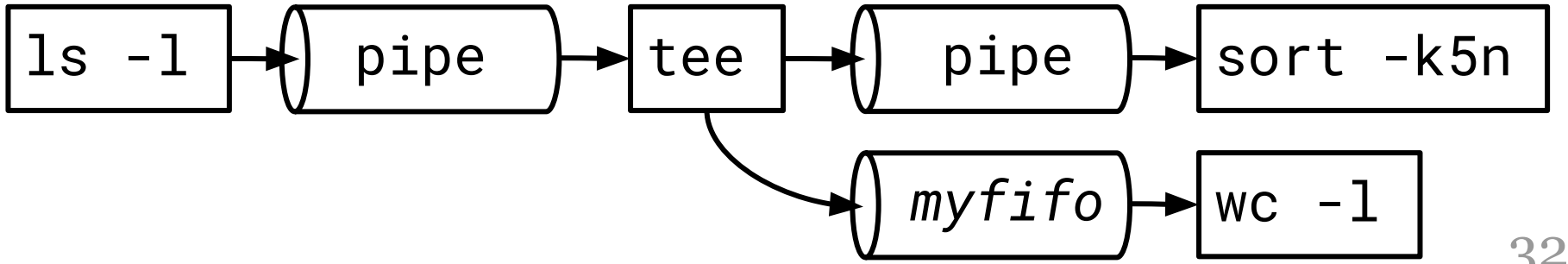
Und ein zweites Programm *my_fifo_w.c*, welches dasselbe File öffnet und eine Nachricht überträgt.

Was ist der Unterschied zu einem regulären File?

FIFOs und das *tee* Kommando

Das *tee* Kommando gibt seinen Input doppelt aus, auf *stdout*, und auf das als Argument übergebene File:

```
$ mkfifo myfifo # FIFO a.k.a. named pipe
$ wc -l < myfifo & # read() blockiert
$ ls -l | tee myfifo | sort -k5n
```



Non-blocking I/O

Manchmal will man beim *open()* nicht blockieren:

```
fd = open(filepath, O_RDONLY | O_NONBLOCK);
```
geht auch, wenn das andere Ende nicht bereit ist.

```
fd = open(filepath, O_WRONLY | O_NONBLOCK);
```
retourniert *-1* und *errno* = *ENXIO*, wenn "zu früh".

Non-blocking I/O ist praktisch um Deadlocks zu verhindern, beim Öffnen mehrerer FIFOs.

Non-blocking *read()* und *write()*

Das *O_NONBLOCK* Flag beeinflusst *read()* / *write()*:

Falls nichts zu Lesen, gibt *read()* -1, *errno* = *EAGAIN*, falls keine Daten und *write()* Ende "zu", kommt *EOF*.

Und *write()*, falls *read()* Ende nicht bereit, und *buffer* in *PIPE_BUF* kein Platz hat, gibt *-1*, *errno* = *EAGAIN*.

Ohne *O_NONBLOCK* würden beide Calls blockieren.

Selbststudium, 3h: Sockets

Lesen Sie diese kurzen Einführungen in Sockets: https://www.unixtutorial.org/unix-sockets-tutorial/, https://www.tutorialspoint.com/unix_sockets/socket_quick_guide.htm

Schauen Sie sich die *man* Pages für Sockets an, sowie die für UNIX Sockets, für Linux Sockets und für IP.

Feedback oder Fragen?

Gerne auf https://fhnw-syspr-fs20.slack.com/

Oder per Email an thomas.amberg@fhnw.ch

Slides, Code & Hands-on: tmb.gr/syspr-8