Betriebssysteme und Netzwerke Vorlesung 5

Artur Andrzejak

Wiederholung Vorlesung 4

Prozesswechsel?

Prozesshierarchien, Boot?

Übergänge zw. Zuständen – wann?

Zustände: Waiting und Ready

IPC – zwei Grundtypen?

SM API: shmget, shmat?

MP – synchron vs. asynchron?

Scheduling, Dispatcher, I/O-bound, CPU-bound?

Warteschlangen - effizient?

Umfragen: https://pingo.coactum.de/301541

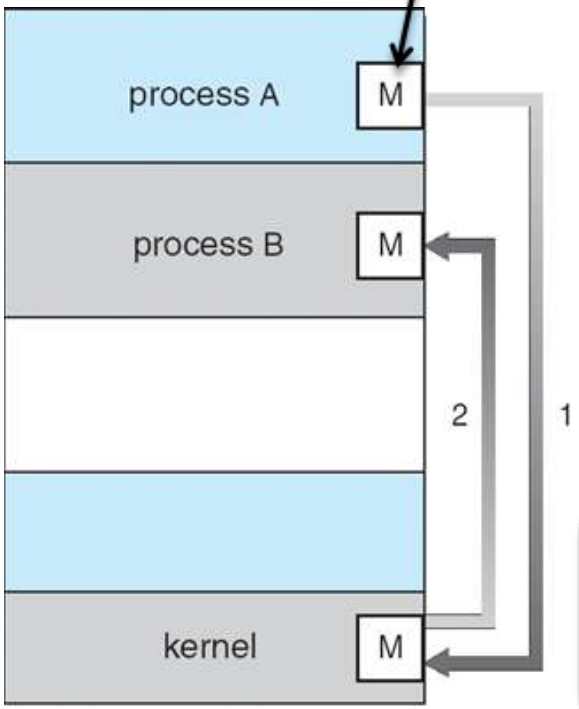
Inter-Prozess Kommunikation (IPC)

Wiederholung

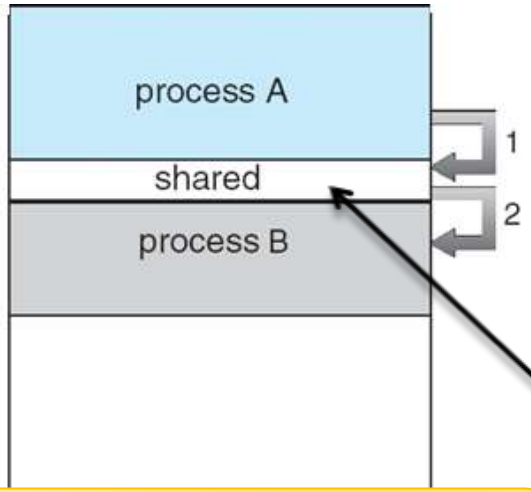
- Es gibt zwei prinzipielle Arten von IPC
  - **1: Nachrichtenübermittlung - message passing (MP)**
  - **2: gemeinsamer Speicher - shared-memory (SM)**

Ein kleiner Speicherbereich von A (Nachricht M) wird durch das BS in den Adressraum von B kopiert

1



2



Der gemeinsame Speicher kann in den Adressraum mehrerer Prozesse eingeblendet werden, als ob dieser ein Teil des Speichers des Prozesses wäre

Was ist effizienter? Was wird häufiger benutzt?

IPC hat Diverse APIs/Implementationen

- **Datenströme**: Kanäle, die die Daten sequentiell von einem Prozess zu einem anderen schicken
  - Pipes, Sockets, Message Queues, Terminal, …
- **Ereignisse**: Spezielle Nachrichten, ggf. ohne Daten
  - Interrupts, Signale, Windows DDE, Apple Events
- **Entfernte Funktionsaufrufe**: Aufruf von Funktionen innerhalb anderer Prozesse
  - Remote Procedure Call/Remote Method Invocation, CORBA, DCOM, Java RMI
- **Shared Memory**: APIs für gemeinsamen Speicher mehrerer Prozesse

Inter-Process Kommunikation in Android

- Einfachere Mechanismen
- **Sandboxing**
  - Gleiches “privates Dateisystem” für mehrere Prozesse
  - Video: #2 Android Interprocess Data Exchange Part 1 [HD 1080p], https://www.youtube.com/watch?v=4u-xpB1RFfs, von 0:15 bis ca. 3:30 (min:sec)[04b]
- **Binder** ist eine ressourcenschonende Form von IPC mittels Shared Memory
  - Programme tauschen per Nachrichten lediglich Referenzen auf Objekte aus, die im Shared Memory abgelegt sind

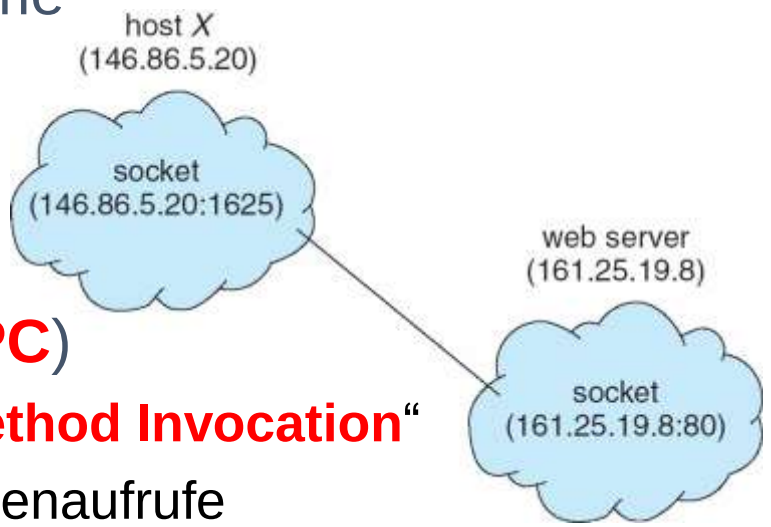
Message Passing (MP)

Message Passing: Synchronisation

- Nachrichtenübermittlung kann **blockierend** oder **nicht-blockierend** sein
- **Blockierend** = **synchron**
  - Der Aufruf von „send“ wird den Sender blockieren (Routine in Bearbeitung), bis die Nachricht empfangen wird
- **Nicht-blockierend** = **asynchron**
  - Der Sender ruft „send“ auf, und die Routine kehrt sofort zurück
- Hat Vorteile und Nachteile – welche?
  - Der Sender kann weiterhin arbeiten, anstatt zu warten (+)
  - Der Sender braucht bei „send“ eine Referenz (Handle), um später abzufragen, ob die Nachricht zugestellt wurde (-)
  - Es ist eine zusätzliche Nachricht für die Bestätigung nötig (-)

Verteilte Inter-Prozess Kommunikation

- Die IPC wird auch benutzt, wenn sich die Prozesse auf verschiedenen Rechnern befinden
- Es gibt ein großes Arsenal an Umsetzungen
  - **Sockets**: imitieren Dateisysteme



  - **Remote Procedure Calls** (**RPC**)
    - unter Java genannt „**Remote Method Invocation**“
    - Imitieren Prozedur- bzw. Methodenaufrufe
  - **Web Services**: ähnlich wie RPC, aber hat universelle Standards, um zwischen Organisationen zu kommunizieren

Beispiel: Message Passing in Mach

- Das BS **Mach** wurde an der Carnegie Mellon University (CMU) entwickelt und ist Teil von **Mac OS X**
- Mach ist „streng“ Nachrichtenbasiert
  - Sogar Systemaufrufe werden so umgesetzt!
  - Nachrichten werden zwischen Postfächern (in Mach sog. **ports**) ausgetauscht (=> indirekte Kommunikation)
- Mit einem neuen Prozess werden zwei Ports erzeugt:
  - kernel port: für Kommunikation Kern (d.h. BS) ⇔ Prozess
  - notify port: für Benachrichtigungen über neue Ereignisse
- Es sind nur drei Operationen nötig
  - `msg_send()`, `msg_receive()`, `msg_rpc()`
  - Erläuterung: **RPC** = remote procedure call
  - Für zusätzliche Ports: `port_allocate()`

Shared-Memory (SM) und Memory-Mapped-Dateien

SM: IPC via Gemeinsamen Speicher in POSIX

- Prozess erstellt zunächst ein **Segment** des gem. Speichers (Details der Parameter)

```
segment_id = shmget (IPC_PRIVATE, size,
S_IRUSR | S_IWUSR);
```

- Prozesse, die Zugang auf dieses Segment wollen, müssen diesem gem. Speicher „abonnieren“ („attach to“) (Parameter)

```
shared_memory =(char *) shmat
(segment_id, NULL, 0);
```

- Aber: wie erfahren die "Abonnenten" den Wert von segment_id?

SM: IPC via Gemeinsamen Speicher in POSIX

- Schreiben / Lesen erfolgt durch "normales" Kopieren in Speicher
  - `sprintf(shared_memory, "Writing here…");`
  - `char read_value = *shared_memory;`

- Wenn Prozess fertig ist, trennt er sich (**detach**) von dem gemeinsamen Speicher (Parameter)
  - `shmdt (shared memory);`

Einblenden von Dateien in den Speicher

- Idee: ein Prozess blendet eine Datei in einen Teil seines Adressenraumes ein
  - Das ergibt ein alternatives Modell der Ein-/Ausgabe
- Anstatt auf Dateien mit open(), read(), write() zu arbeiten, kann man auf die Datei wie auf ein großes Feld (array) zugegriffen werden
- Effizienz
  - Schreibzugriffe müssen nicht sofort auf die Festplatte (FP) geschrieben werden
  - Wenn Prozess terminiert oder Datei geschlossen wird, werden alle Veränderungen auf Festplatte geschrieben

Nutzung für Shared Memory

- **Memory-Mapped-Dateien** (MMD) werden auch für die Inter-Prozess-Kommunikation verwendet
  - Leichter ein Dateipfad zu teilen als ein Segment_ID
  - Unter Linux und Unix nicht ganz üblich
- Unter MS Windows benutzt man diesen Mechanismus häufiger
  - Man erzeugt zunächst ein **Datei-Mapping** (**file mapping**)
  - Dann wird eine **Ansicht** (**view**) der Datei in dem logischen Adressraum erzeugt
  - Ein weiterer Prozess kann dann eine weitere Ansicht in seinem Adressraum kreieren
- Nachfolgender Code nur zur Illustration, kein Klausurstoff

Sender-Prozess (Producer)

```
int main(int argc, char *argv[])
{
HANDLE hFile, hMapFile;
LPVOID mapAddress;

    // first create/open the file
    hFile = CreateFile("temp.txt",
                       GENERIC_READ | GENERIC_WRITE,
                       0,
                       NULL,
                       OPEN_ALWAYS,
                       FILE_ATTRIBUTE_NORMAL,
                       NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        fprintf(stderr,"Could not open file temp.txt (%d).\n",GetLastError());
        return -1;
    }

    // now obtain a mapping for it

    hMapFile = CreateFileMapping(hFile,
                                 NULL,
                                 PAGE_READWRITE,
                                 0,
                                 0,
                                 TEXT("SharedObject"));

    if (hMapFile == NULL) {
        fprintf(stderr,"Could not create mapping (%d).\n", GetLastError());
        return -1;
    }
```

Sender-Prozess (Producer)

```
    // now establish a mapped viewing of the file

    mapAddress = MapViewOfFile(hMapFile,FILE_MAP_ALL_ACCESS,0,0,0);

    if(mapAddress == NULL) {
        printf("Could not map view of file (%d).\n", GetLastError());
        return -1;
    }

    // write to shared memory

    sprintf((char *)mapAddress,"%s","Shared memory message");

    while (1);
    // remove the file mapping
    UnmapViewOfFile(mapAddress);

    // close all handles
    CloseHandle(hMapFile);
    CloseHandle(hFile);
}
```

Empfänger-Prozess (Consumer)

```
#include <stdio.h>
#include <windows.h>

int main(int argc, char *argv[]) {
HANDLE hMapFile;
LPVOID lpMapAddress;

hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,          // read/write permission
                           FALSE,                        // Do not inherit the nam
                           TEXT("SharedObject")); // of the mapping object.

if (hMapFile == NULL)
{
   printf("Could not open file mapping object (%d).\n", GetLastError());
   return -1;
}

lpMapAddress = MapViewOfFile(hMapFile,            // handle to mapping object
                             FILE_MAP_ALL_ACCESS, // read/write permission
                             0,                   // max. object size
                             0,                   // size of hFile
                             0);                  // map entire file

if (lpMapAddress == NULL)
{
   printf("Could not map view of file (%d).\n", GetLastError());
   return -1;
}

printf("%s\n",lpMapAddress);

UnmapViewOfFile(lpMapAddress);
CloseHandle(hMapFile);

}
```

Umfrage (https://pingo.coactum.de/301541)

Sie sind ein Entwickler an einem großen, sicherheitskritischem Projekt (z.B. MCAS bei Boeing ☺). Wenn Sie für IPC **Shared Memory** (**SM**) statt **Message Passing** (**MP**) verwenden, (i) steigern Sie Ihre street credibility bei den anderen Entwicklern, aber (ii) die Quality Assurance Abteilung (die „Tester“) wird Sie hassen. Warum? Grund für nur (i) oder (ii) OK.

- A. Durch die Tweets von Donald Trump ist MP bei den Entwicklern in Verruf geraten
- B. Bei SM kann es zu sehr schweren und seltenen Defekten kommen
- C. Bei MP formatiert das BS die Nachrichten, so dass man sie leichter Debuggen kann
- D. SM ist schneller, da u.a. keine Systemaufrufe (Abgesehen von der Initialisierung) nötig sind

IPC via Pipes

Motivation /1

- Angenommen, wir wollen alle Bilder (*.jpg - Dateien) in einer Verzeichnishierarchie (rekursiv) durchzählen
- Effektives Vorgehen (Linux)?
- Shell-Befehle bzw. Skripte!
- **ls** –R <Dir>
  - Liste (rekursiv) Inhalte des Verzeichnisses <Dir> auf
- **grep** <Optionen> <Muster> <Input-Datei(en) D>
  - Suche Textmuster in allen Zeilen der Input-Datei(en) D
  - Default: Per gefundene Stelle wird eine Zeile ausgegeben
  - Optionen:
    - -c, --count: nur zählen, keine Zeilen von D ausgeben
    - -i, --ignore-case: Klein/Großschreibung ignorieren

Motivation /2

- 1. Wir leiten die Ausgabe von **ls** in eine Datei um
  - Mit: Befehl **>** Dateiname
- 2. Wir verarbeiten die Datei mit **grep**

- ls –R **>** tmp-file.txt
- grep –ci '\.jpg$' tmp-file.txt
  - Ausgabe bei mir (/home/Artur/Galleries): 4603
- Störend: die temporäre Datei tmp-file.txt
- Eigentlich ist das ein Fall für IPC!
- „We should have some ways of connecting programs like **garden hose** – screw in another segment when it becomes necessary to massage data in another way.“
  - – DOUGLAS MCILROY (Memo zu Multics-Projekt, 1964)

Pipes in den Shells

- Shells wie bash erlauben es, zwei Prozesse via ein „Pipe“ zu verbinden – Zeichen **|** (engl. „Pipe“)
- ProA **|** ProB
  - Die Standard-Ausgabe von Prozess A (ProA) wird an die Standard-Eingabe von Prozess B (ProB) weitergeleitet
  - Standard-Ausgabe: normalerweise Konsole (Terminal)
  - Standard-Eingabe: normalerweise Tastatur
- Wir können also das Shell-Programm:

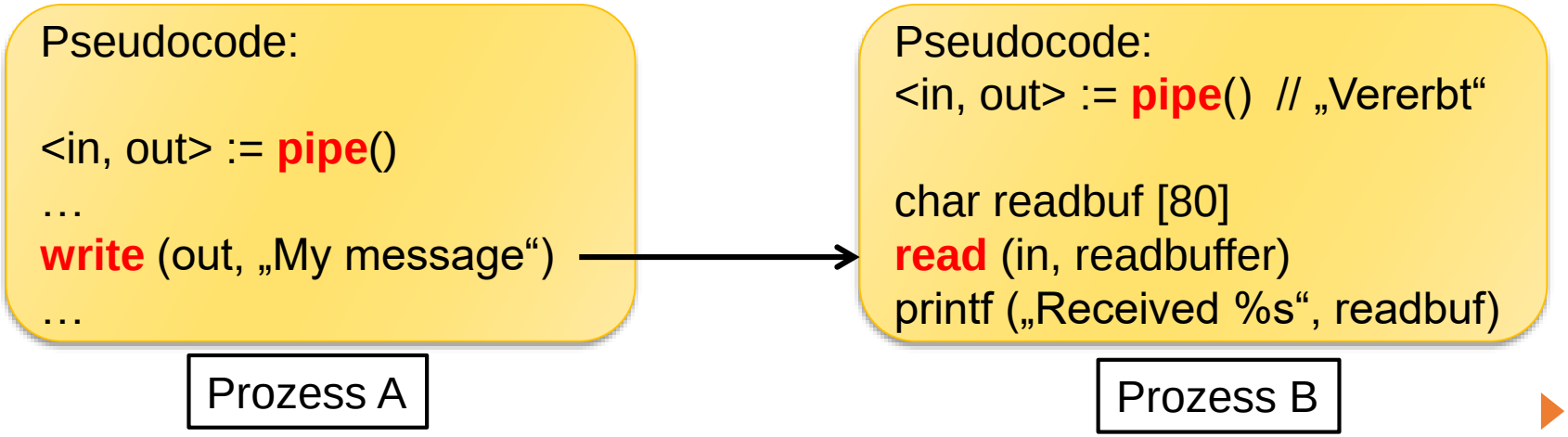
```
ls –R > tmp-file.txt
grep –ci  '\.jpg$' < tmp-file.txt
```

- Wie vereinfachen?

```
ls –R | grep –ci  '\.jpg$'
```

Pipes via Systemaufrufe

- Ein **Pipe** (Rohr, Röhre) ist ein Mechanismus für einen *gepufferten Datenstrom zwischen zwei Prozessen*
  - Ein Prozess erzeugt Datenstrom, der andere liest diesen
  - Für Daten gilt das „**FIFO**“ (**First In - First Out**)-Prinzip
- Die Pipes werden als eine spezielle Datei behandelt
  - Die Programme können diese mit normalen Dateioperationen **read**() und **write**() benutzen



Video zu Pipes

- **AT&T Archives: The UNIX Operating System**
  - https://www.youtube.com/watch?v=tc4ROCJYbm0&list=PLqE63EN7m04eoD84hdrHK7rt9-zsZHe78
  - Ab 6:00 bis 10:50 (min:sec) [01b]

Anonyme Pipes: Eigenschaften

- „Privat“ für die beteiligten Prozesse
- Unidirektional: ein Prozess schreibt, der andere liest
- Die maximale Datenmenge, die ein Pipe puffern kann, ist relativ klein
- Sie terminieren, wenn der Erzeuger-Prozess terminiert
- Anwendung
  - Ein Prozess erzeug ein Kind mit fork() und nutzt Pipe, um mit dem Kind zu kommunizieren
- Benennung und Systemaufrufe
  - POSIX: **ordinary** pipes, erzeugt mit **pipe**()
  - Win32: **anonymous** pipes

Anonyme Pipes aus Programmen

- Erzeugung unter POSIX via **int pipe(int fd [2])**
  - **fd** ist ein Paar von **Dateideskriptoren** (d.h. „Identifikatoren“ der geöffneten Dateien, später)
  - fd[0] : aus dieser Datei liest ein Prozess die Daten
  - fd[1] : in diese Datei schreibt ein Prozess

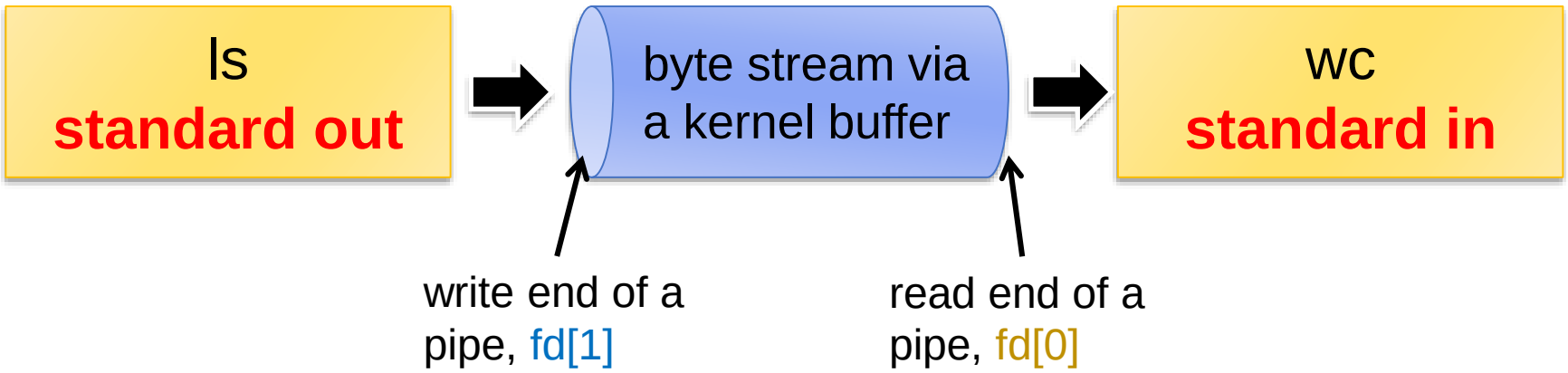
Code für Sender und für Empfänger in C

- *int fd[2]; **pipe** (fd);* // im Elternprozess
- Empfänger:
  - *read (fd[0], readbuf, sizeof(readbuf));* // fd[0] ist „Ausgang“
- Sender:
  - *write (fd[1], string, strlen(string)+1);* // fd[1] ist „Eingang“

Pipes - Video

- Video zu (anonyme) Pipes: „[Linux.conf.au 2013] - An Introduction to Linux IPC Facilities”, [05a], Link: http://goo.gl/6DF71v (ab 8:45 bis ca. 11:40, min:sec)

ls | wc -l



Codebeispiel

fd[**0**] : aus dieser Datei liest man
fd[**1**] : in diese Datei schreibt man

```
int main(void) {
    int fd[2], nbytes;                  pid_t  childpid;
    char  string[] = "Hello!\n";        char   readbuffer[80];
    pipe (fd);
    if( (childpid = fork()) == -1) {    perror("fork");  exit(1);      }
    if(childpid == 0) {
        close (fd[0]);      /* ????? */
        /* ????? */
        write (fd[1], string, (strlen(string)+1));    exit(0);
    }  else {
        close (fd[1]);      /* ????? */
        /* ??????? */
        nbytes = read (fd[0], readbuffer, sizeof(readbuffer));
        printf ("Received string: %s", readbuffer); }
    return(0); }
```

Quelle: http://tldp.org/LDP/lpg/node11.html

Codebeispiel

fd[**0**] : aus dieser Datei liest man
fd[**1**] : in diese Datei schreibt man

```
int main(void) {
    int fd[2], nbytes;                  pid_t  childpid;
    char  string[] = "Hello!\n";        char   readbuffer[80];
    pipe (fd);
    if( (childpid = fork()) == -1) {    perror("fork");  exit(1);      }
    if(childpid == 0) {
        close (fd[0]);      /* Child process closes up input side of pipe */
        /* Send string through the output side of pipe */
        write (fd[1], string, (strlen(string)+1));    exit(0);
    }  else {
        close (fd[1]);      /* Parent process closes up output side of pipe */
        /* Read in a string from the input side of pipe */
        nbytes = read (fd[0], readbuffer, sizeof(readbuffer));
        printf ("Received string: %s", readbuffer); }
    return(0); }
```

Quelle: http://tldp.org/LDP/lpg/node11.html

Benannte Pipes (**Named** Pipes)

- Jeder Prozess, der den Namen einer benannten Pipe kennt, kann über diesen Namen die Verbindung zur Pipe und damit zu anderen Prozessen herstellen
- Sie können zur Kommunikation zwischen Prozessen auf unterschiedlichen Rechnern eingesetzt werden
  - => Flexibler als anonyme Pipes
- Bidirektional und Vollduplex-fähig: erlauben die gleichzeitige Kommunikation in beide Richtungen
- Benennung und Systemaufrufe
  - UNIX – gennant **FIFO**s, erzeugt von **mkfifo**()
  - Windows – named pipe, erzeugt von **CreateNamedPipe**()

Beispiel: mkfifo in bash

- Öffne zwei Shells (Terminals) und wechsle zu /tmp/

Shell A:

- mkfifo myPipe
- echo “ExampleText“ >myPipe

Shell B:

- while read line; do echo “Received: ${line}"; done<myPipe

Pipes und Dateideskriptoren

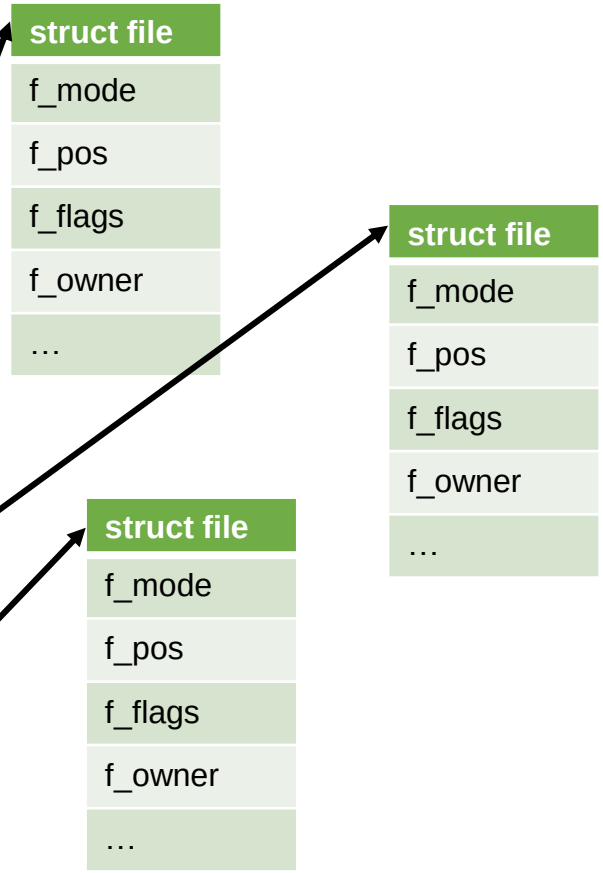
Dateidescriptoren bzw. Handles

- Ein **Dateideskriptor** (**file descriptor**) (**DD**) ist der *Index* (Identifikator) *einer geöffenen Datei*
- **int open (const char ****pathname***, int** *flags***)** liefert ein DD (integer), der später für read/write benutzt wird
  - DD sind „lokal“, d.h. privat für jeden Prozess
- Wichtig: der DD unter POSIX ist tatsächlich ein Integer (0, 1, 2, 3, …) und kein Zeiger oder Objekt!
  - Tatsächlich: ein Index zu einer Datenstruktur im BS
- Unter Windows oder Std C Library entspricht das einem **file handle** (i.A. kein integer)
- Gute Erklärung: https://www.bottomupcs.com/file_descriptors.xhtml

Dateideskriptoren in Linux - Umsetzung

Process Control Block (PCB)
(In Linux, task_struct)
pid;
…
fd[]; // => folgende Tabelle:

DD = Index in der Tabelle	**Zeiger auf *struct file***
0	0xba92c100
1	0xba94d200
…	…
5	0xba962100



Standard-Dateien und Deskriptoren

- Wenn unter Linux/Unix ein Prozess erzeugt wird, werden vom BS *automatisch* drei Dateien eröffnet:
  - DD-Wert — Konvention
  - 0 — Standard Input (stdin) — Standardeingabe
  - 1 — Standard Output (stdout) — Standardausgabe
  - 2 — Standard Error (stderr) — Standardfehler
- Als default ordnet das BS diesen Dateien bestimmte „Geräte“ zu
  - Standard**ein**gabe: als default die Tastatur
  - Standard**aus**gabe: als default die Konsole (Terminal)
  - Standard**fehler**: als “default” die Konsole (Terminal)

Standardkanäle und Konventionen

- Konvention: Programme nehmen immer die Datei mit DD = 0 für die Standardeingabe, usw.
  - Insbesondere Shell-Prg wie grep, ls usw.
  - Auf printf(..) schreibt immer in eine Datei mit DD = 1!

Sicht des Prozesses		Sicht des Betriebssystems	
Bedeutung	**DD-Wert (fest!)**	**DD-Wert**	**Default-“Gerät“**
Std Input	0	0	Tastatur
Std Output	1	1	Konsole
Std Error	2	2	Konsole

Umlenken von Std-Ein/Ausgabe

- Können wir dem BS anordnen, die default-Geräte (Tastatur,…) durch andere bzw. Dateien zu ersetzen?
- In der Shell können wir die Ein-/Ausgabe **umlenken**
  - Eingabe: Prozess „vorgaukeln“, dass die Daten von der Tastatur kommen, während sie aus einer Datei kommen
  - Analog bei Ausgabe: Prozess „vorgaukeln“, dass es auf die Konsole schreibt, aber er schreibt in eine Datei
- Eingabe mit **<** als Kommandozeitenparameter:
  - *wc* **<** *tmp-file.txt*
- Ausgabe mit **>**:
  - *ls –R* **>** *tmp-file.txt*

Umlenken von Std-Ein/Ausgabe /2

- In der Shell können wir die Ein-/Ausgabe **umlenken**
  - …
- Was passiert hier mit den Dateideskriptoren (DD)?

- **ls** schreibt auf Std.-Ausgabe (= Datei mit DD = 1) und glaubt, DD 1 gehört zur Konsole, ABER …
  - Shell hat tmp-file.txt (mit irgendeinem) DD **x** geöffnet
  - Dann hat sie (für ls) **x** durch 1 ersetzt …
  - Und die Datei mit dem bisherigem DD 1 (= Konsole) geschlossen

Ersetzen von Dateideskriptoren

- ls schreibt auf Std.-Ausgabe (= die Datei mit DD = 1) und [illegible] DD [illegible] Konsole, ABER
  - Shell hat tmp-file.txt (mit irgendeinem) DD x geöffnet
  - Dann [illegible] DD x [illegible] …
  - Und die Datei mit bisherigem DD 1 geschlossen

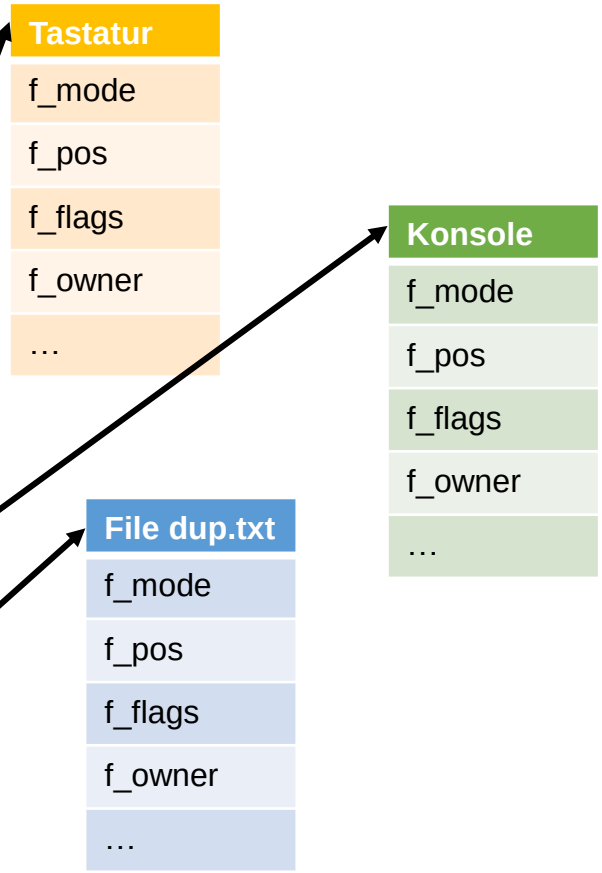
Wie ersetzt man die Dateideskriptoren?

- Man ersetzt die DD unter POSIX mit **dup2**()
  - **int dup2 (int srcDD, int targetDD)**
- Erzeugt neben existierendem DD srcDD ein zweites DD (targetDD) - ein Alias auf die geöffnete Datei zu srcDD
- Wert des targetDD wird vom Benutzer vorgegeben
  - Falls eine Datei mit targetDD für diesen Prozess schon geöffnet war, wird sie geschlossen, wie bei close(targetDD)

Wirkung von int **dup2** (int srcDD, int targetDD)

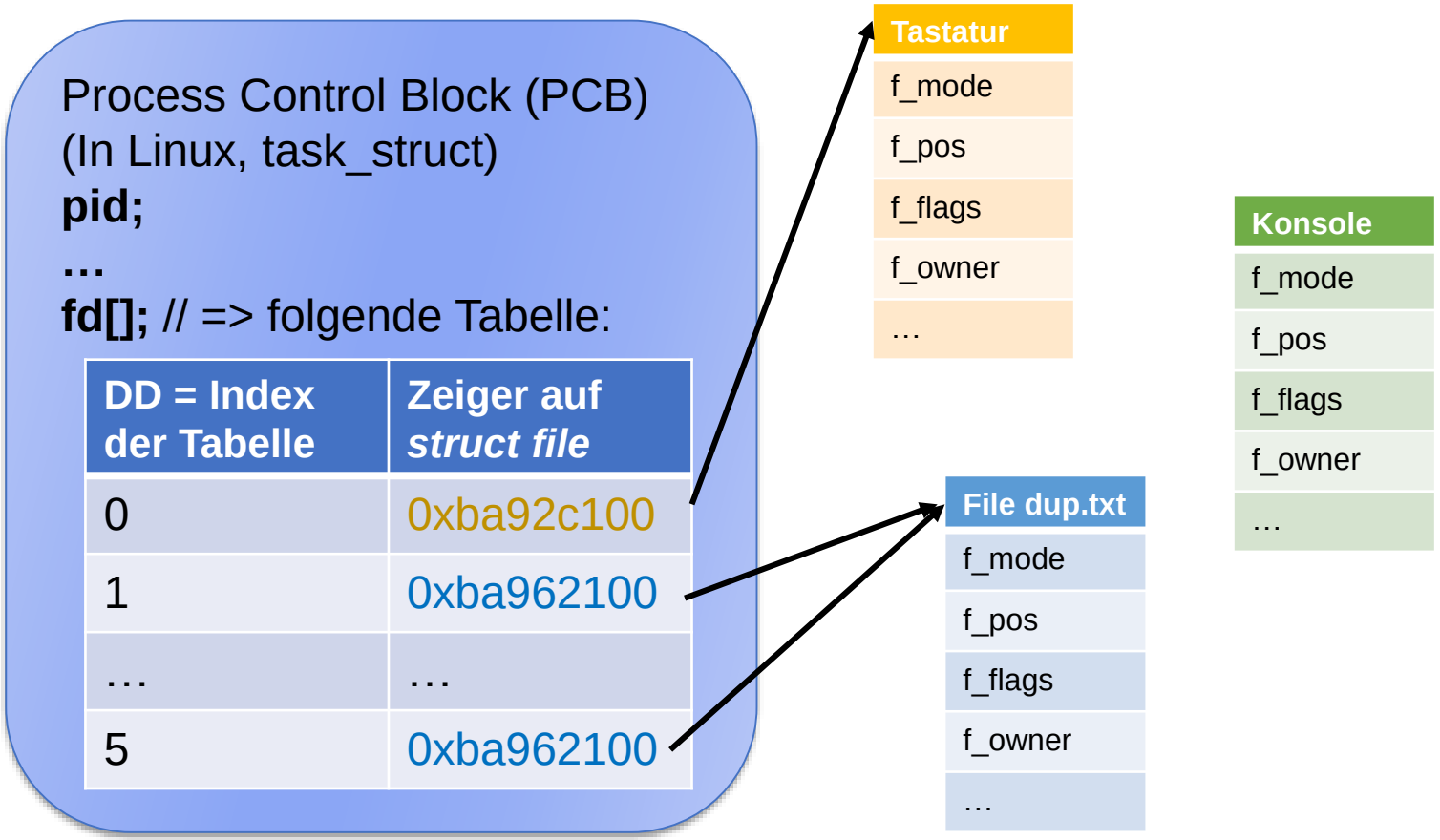
Process Control Block (PCB)
(In Linux, task_struct)
pid;
…
fd[]; // => folgende Tabelle:

DD = Index der Tabelle	Zeiger auf *struct file*
0	0xba92c100
1	0xba94d200
…	…
5	0xba962100




```
int srcDD = open ("dup.txt", O_WRONLY | O_APPEND);
dup2 (srcDD, 1) ;
printf (“What happens with this text?\n");
```

Wirkung von int **dup2** (int srcDD, int targetDD)




```
int srcDD = open ("dup.txt", O_WRONLY | O_APPEND);
dup2 (srcDD, 1) ;
printf (“What happens with this text?\n");
```

Umfrage (https://pingo.coactum.de/301541)

Welche Wirkung hat folgender Code?

- int srcDD = open ("dup.txt", O_WRONLY | O_APPEND);
- **dup2** (srcDD, 1) ;
- printf (“What happens with this text?\n");

A. Der String „ What happens ..“ wird sowohl auf die Konsole als auch in die Datei „dup.txt“ geschrieben

B. Der Prozess kann in die Datei „dup.txt“ über den DD Nr. 5 schreiben, z.B. `write(5, „abcd“, 5);`

C. Der Prozess könnte via `write(1, „abcd“, 5);` auf die Konsole schreiben

D. Kein anderer Prozess kann in die Datei „dup.txt“ schreiben, solange sie in diesem Prozess noch geöffnet ist

Details zu dup2() und dup()

- Details zu int **dup2** (int oldfd, int newfd)
  - If the descriptor newfd was previously open, it is silently closed before being reused
  - If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed
  - If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd
- Was macht int **dup** (int srcFD)?
  - “The dup() system call creates a copy of a file descriptor
  - It uses the lowest-numbered unused descriptor for the new descriptor
  - If the copy is successfully created, then the original and copy file descriptors may be used interchangeably
  - They both refer to the same open file description and thus share file offset and file status flags”
- Aus: https://www.geeksforgeeks.org/dup-dup2-linux-system-call/

Ein Weiteres Beispiel zu dup2()

- int fid = open (“/tmp/nowDate.txt”, **O_WRONLY**);
- int status = **dup2 (fid, 1)**;
- // ruft Kommando “date” auf und schreibt auf stdout
- execlp (“date",“date",NULL);
- Was passiert hier?
- Eine Datei wird geöffnet (write only), man erhält fid (DD)
- Zusätzlich zum bisherigen DD (fid) der Datei nowDate.txt bekommt die Datei einen weiteren DD mit Wert „1“
- => Prozess-Ausgabe, die in die Datei mit DD „1“(= bis jetzt Konsole) geht, geht nun in die Datei nowDate.txt
- Dasselbe mit einer Shell?

```
date > /tmp/nowDate.txt
```

Pipes mit Shell (Wiederholung)

- Wir haben bis jetzt gesehen, wie man via BS-Aufrufe (einfache) Pipes benutzen kann
- Hilft uns das, unsere Bilder zu zählen?
  - Nein, man müsste ein Programm in C schreiben
- Zum Glück erlaubt die Shell, zwei Prozesse via ein Pipe zu verbinden – Zeichen **|** (engl. „Pipe“)
  - ProA **|** ProB
  - Die Std.-Ausgabe von ProA wird mit der Std.-Eingabe von ProB verbunden
- Vereinfachung davon?

```
ls –R > tmp-file.txt
grep –ci  '\.jpg$' < tmp-file.txt
```

```
ls –R | grep –ci  '\.jpg$'
```

Pipes in UNIX – Beispiel: wie „**who | sort**“

1. Was macht „who | sort“ in der Shell?
2. Wie viele Prozesse sind hier beteiligt?

pipe_verbindung[0] : aus dieser Datei liest man
pipe_verbindung[1] : in diese Datei schreibt man

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void){
   int pipe_verbindung[2];
   pipe (pipe_verbindung);
   if ( fork()==0 ) {
      dup2 (pipe_verbindung[1], 1);        // was macht diese Zeile (A1)?
      close (pipe_verbindung[0]);          // was macht diese Zeile (A2)?
      execlp ("who","who",NULL);
   } else if ( fork()==0 ) {
      dup2 (pipe_verbindung[0], 0);        // was macht diese Zeile (B1)?
      close (pipe_verbindung[1]);          // was macht diese Zeile (B2)?
      execlp ("sort","sort",NULL);
}}
```

Pipes in UNIX – Beispiel: wie „**who | sort**“

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void){
    int pipe_verbindung[2];
    pipe (pipe_verbindung);
    if ( fork()==0 ) {                          // Kindprozess 1
        dup2 (pipe_verbindung[1], 1);           // Ersetze std-out durch Pipe-in
        close (pipe_verbindung[0]);             // „Leseende“ schließen
        execlp ("who","who",NULL);              //  „who“ ausführen
    } else if ( fork()==0 ) {                   // Kindprozess 2
        dup2 (pipe_verbindung[0], 0);           // Ersetze std-in durch Pipe-out
        close (pipe_verbindung[1]);             // „Schreibende“ schließen
        execlp ("sort","sort",NULL);            //  „sort“ ausführen
}}
```

1. „who | sort“ listet die Benutzer, die eingeloggt sind, und sortiert diese nach login-Namen
2. Es sind drei Prozesse (Hauptprogramm + 2 „fork()“)

Weitere IPC Möglichkeiten unter Linux

- Linux bietet einige weitere Möglichkeiten für Inter-Process Communication (IPC)
- Eine gute Übersicht findet man am Anfang des Videos „[Linux.conf.au 2013] - An Introduction to Linux IPC Facilities", Link: http://goo.gl/6DF71v
  - Ab 2:18 (Min:Sec)
- Im gleichen Video: IPC mit **Message Queues**, ab 18:25 (min:sec)
  - MQs ist einer wichtiger und häufig genutzter IPC Mechanismus (vom Typ Message Passing)

Zusammenfassung

- IPC via Shared Memory / Memory-Mapped Dateien
- IPC via Pipes
- Dateideskriptoren und Ein-/Ausgabeumleitung

- Quellen
  - Silberschatz et al., Kapitel 6
  - Tanenbaum Kapitel 2.3
  - Wikipedia