

Betriebssysteme

Lehrveranstaltung

Vorlesung

Dozent: Thomas Fahringer (thomas.faringer@uibk.ac.at)

Radu Prodan (radu.prodan@uibk.ac.at)

Termin: **Mittwoch 15:15 – 16:00, Hörsaal A**

Freitag 10:15 – 12:00, Hörsaal D

Klausur-Termin: **05.07.2017, 15:0 – 17:00, Hörsaal A**

Anmeldung ab 31.05.2017

Links:

- https://lfoonline.uibk.ac.at/public/lfoonline_lv.details?lvnr_id_in=703012&sem_id_in=17S
- <http://dps.uibk.ac.at/teaching/>
- <http://dps.uibk.ac.at/teaching/>

Proseminar

Dozent: Peter Thoman ([???](#))

Termin: **Mittwoch 16:15 – 18:00, RR 15**

Prüfungen: **Zwei schriftliche Tests**

Prüfungstermine: ???

Anzahl erlaubter, unentschuldigter Fehlstunden: **0**

Links:

- https://lfoonline.uibk.ac.at/public/lfoonline_lv.details?lvnr_id_in=703012&sem_id_in=17S
- <http://dps.uibk.ac.at/teaching/>
- <https://lms.uibk.ac.at/auth/RepositoryEntry/4154589249>
- https://github.com/Juanjdurillo/intro_unix
- https://github.com/PeterTh/bs_ue_2017

Kompendium

Inhaltsverzeichnis

| | |
|---|----|
| Lehrveranstaltung..... | 1 |
| Vorlesung..... | 1 |
| Proseminar | 1 |
| Betriebssysteme..... | 5 |
| Einführung..... | 5 |
| Hardwareabstraktion..... | 5 |
| Virtuelle Maschine..... | 5 |
| Multiplexing..... | 5 |
| Eigenschaften..... | 5 |
| Benutzerfreundlichkeit | 6 |
| Infrastruktur | 6 |
| Kosten | 6 |
| Anpassungsfähigkeit..... | 6 |
| Warum wir Betriebssysteme lernen..... | 6 |
| Entwicklung | 6 |
| Arten von Kernels..... | 7 |
| Arten von Betriebssystemen | 9 |
| Systemaufrufe | 9 |
| Fehlerbehandlung..... | 10 |
| Systemkontakte..... | 10 |
| Kontextübergänge | 10 |
| POSIX | 10 |
| Prozesse | 10 |
| Einführung..... | 10 |
| Durchsatz | 10 |
| Wartezeit (Latenz)..... | 11 |
| Prozesshierarchien..... | 11 |
| Prozessperspektive auf das System..... | 11 |
| Prozess-Verwaltung aus Programmierersicht | 11 |
| Kernelsicht von Prozessen..... | 15 |
| Context Switch | 16 |
| Preemptions..... | 16 |
| Cooperative..... | 16 |
| Visualisierung eines Context Switches..... | 17 |

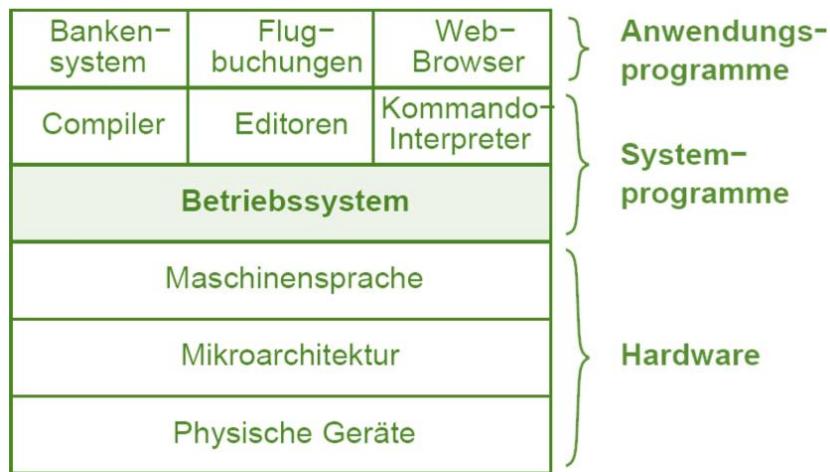
| | |
|---|----|
| Scheduling..... | 17 |
| Grundlegende Begriffe | 18 |
| Scheduling Policy | 18 |
| Prozessor und andere Geräte | 22 |
| Fortgeschrittenes Scheduling | 23 |
| Scheduling Implementierungen..... | 25 |
| Spezialfälle..... | 25 |
| Synchronisierung..... | 26 |
| Nebenläufigkeit | 26 |
| Gemeinsame Ressourcen | 27 |
| Kritische Region | 28 |
| Semaphore | 32 |
| Monitor | 33 |
| Klassische Synchronisationsprobleme | 35 |
| Deadlocks..... | 37 |
| Interprozesskommunikation | 37 |
| System V IPC vs POSIX IPC..... | 38 |
| Signale (Unix) | 38 |
| Synchrone Signale..... | 39 |
| Asynchrone Signale..... | 39 |
| Beispiel-Signale | 39 |
| Pipes | 39 |
| Coprozesse..... | 41 |
| Benannte Pipes (FIFOs) | 41 |
| Kommandozeile | 41 |
| FIFOs in C | 41 |
| Regeln für FIFOs | 42 |
| Beispiele..... | 42 |
| XSI IPC..... | 43 |
| Einrichten neuer IPC-Objekte | 44 |
| Verbindung herstellen..... | 44 |
| Löschen von Objekten | 44 |
| Zugriffsrechte und Limits | 45 |
| Kommunikationsmöglichkeiten | 45 |
| Nachrichtenwarteschlangen..... | 45 |
| Message-Get..... | 46 |
| Message-Send | 46 |
| Message-Send | 46 |

| | |
|--|----|
| Message-Control | 48 |
| Shared Memory (Gemeinsamer Speicher) | 48 |
| System V-Interface..... | 49 |
| POSIX-Interface | 51 |
| Semaphore..... | 52 |
| Synchronisierung mit mehreren Semaphoren | 52 |
| Arten von Semaphoren | 53 |
| SVR 4 Semaphore..... | 53 |
| Status und Limits..... | 54 |
| System V-Interface..... | 54 |
| POSIX-Interface | 56 |
| Threads..... | 57 |
| Der Thread | 58 |
| Eigenschaften..... | 58 |
| Vorteile / Nachteile..... | 58 |
| POSIX-Threads..... | 59 |
| Wichtige Informationen | 59 |
| POSIX-Interface | 59 |
| Beispiel..... | 62 |
| Nebenläufigkeitsprobleme..... | 63 |
| Beispiel..... | 63 |
| Mutual Exclusion | 63 |
| Deadlocks und Backoff-Algorithmus | 65 |
| Bedingungsvariablen..... | 65 |
| POSIX-Auswirkung auf UNIX..... | 66 |
| Fork..... | 66 |
| Exec und Exit..... | 66 |
| Thread-Safety & Eintrittsinvariante Funktionen | 66 |
| Signale..... | 67 |
| Weitere Synchronisierungsmöglichkeiten..... | 67 |
| Semaphoren..... | 67 |
| Read/Write-Sperren..... | 67 |
| Barrier..... | 67 |
| Spinlocks | 67 |
| Zusammenfassungen | 67 |
| Übersichten | 70 |

Betriebssysteme

Einführung

Was ist ein Betriebssystem?



Ein Betriebssystem kontrolliert die Ausführung von Programmen/Prozessen, sprich wann welcher Prozess ausgeführt wird. Es verteilt die vorhandenen Ressourcen und holt sie auch wieder zurück.

Hardwareabstraktion

Hardware stellt meist umfangreiche, sehr komplexe Funktionalität zur Verfügung, weil sie billig, schnell und zuverlässig sein muss. Dies resultiert in einer schwierigen Programmierung.

Das OS abstrahiert diese (meist sehr unterschiedlichen) Funktionalitäten der Hardware und bietet „vereinheitlichte“ Funktionen zur Verfügung, über die auf die Hardware zugegriffen werden kann.

Virtuelle Maschine

Ein OS ist grundsätzlich eine virtuelle Maschine – weil es, wie oben geschrieben, eine einheitliche Sicht auf die Hardware für Programme realisiert.

Multiplexing

Während man früher noch zu jedem Zeitpunkt auf einem Rechner nur eine Anwendung laufen lassen konnte, kann man heute einen Rechner im Mehrprozess- und Mehrbenutzerbetrieb verwenden. Damit können mehrere Anwendungen durch mehrere verschiedene Benutzer „gleichzeitig“ ausgeführt werden.

Dafür ist eine „gerechte“ Verteilung der Ressourcen, so wie ein Schutz der Anwendungen und Benutzer voreinander und untereinander nötig.

Eigenschaften

Ein Betriebssystem stellt im Idealfall eine einfache und produktive Umgebung bereit:

- ∅ häufig gebrauchte Subroutinen
- ∅ Zugriff auf Hardware
- ∅ „Abstrakte“ Funktionalität

Dabei sind 4 Eigenschaften wichtig:

Benutzerfreundlichkeit

Dafür ist eine **Robustheit** (z.B.: falsche Inputs), **Proportionalität** (einfache Aktionen einfach, teure Aktionen schwierig), **Komfort** und **Widerspruchsfreiheit** (Konventionen) nötig

Infrastruktur

Ausreichend für den **Verwendungszweck**, **vollständig** und **angemessen** soll sie sein.

Kosten

Services sollten **effizient** sein, **gute Algorithmen** implementieren und **wenig Overhead** produzieren (nichts tun sollte nichts kosten).

Anpassungsfähigkeit

Das Betriebssystem sollte sich **an** das **Umfeld anpassen** (nicht umgekehrt), **Veränderungsfähig** über die Zeit und **erweiterbar** sein.

Warum wir Betriebssysteme lernen

Keine Anwendung (Ausnahmen jaja) verbraucht soviel Ressourcen wie das Betriebssystem. Sie sind (aufgrund der Anbindung zu verschiedenen Hardware-Geräten) hochkomplex und notwendig für jeden Computer.

Außerdem gibt es immer noch viele ungelöste Probleme, z.B. Ressourcenverbrauch in embedded systems (wie Batterie), Sicherheit und Optimierungen.

Entwicklung

Dieser Teil ist Prüfungsirrelevant (glaub i) – aber cool zu wissen

Bis 1955 gab es noch **keine Betriebssysteme**, man hat Programme manuell (für jede Ausführung) in den Speicher laden müssen.

Erst ab **1955 bis 1965** wurde mit der **Stapelverarbeitung** (Lochkarten, die den Programmcodes enthalten) erste Betriebssysteme realisiert. Das OS hat dabei über einen Compiler die Karte „kompiliert“, das Kompilat dann ausgeführt und das Ergebnis zurückgegeben. Man konnte so mehrere Lochkarten wie in einem Stapel (hence „Stapelverarbeitung“) hintereinander ausführen.

Hierbei war das OS aber sehr primitiv, d.h.: es gab kaum Sicherheit für die Funktionalität, es konnte immer nur ein Programm zu jedem bestimmten Zeitpunkt ausgeführt werden, folglich auch keine „gefährlichen“ Benutzer oder Programme auftreten. Problematisch war allerdings die Performance – z.B.: musste die CPU idlen während das Programm auf den Abschluss der IO-Operation wartete.

Zwischen **1965 und 1980** kamen dann immer mehr „Rechnerfamilien“ in den Vorschein, welche gleiche Befehlssätze verwendeten – diese konnte das Betriebssystem nun abstrahieren. Auch wurde ein **Mehrprogrammbetrieb** eingeführt werden, das heißt, musste das OS auf den Abschluss einer Operation warten (CPU idlet) dann wurden die Ressourcen einfach einem anderen Programm zugeteilt. Auch eine interaktive Nutzung über Terminals, statt über Lochkarten

wurde eingeführt – genauso wie der Mehrbenutzerbetrieb. Man musste also auch für einen gegenseitigen Schutz sorgen.

Ab 1980 wurden **Mikroprozessoren**, also kleine/billige Rechner, sowie zunehmende **Vernetzung** der Rechner immer moderner. Auch musste immer mehr Hardware abstrahiert werden.

Die Zukunft wird (laut Fahringer) in die Virtualisierung (also mehrere OS gleichzeitig aktiv) gehen.

Mehrprogrammbetrieb

Wenn ein Prozess blockt, führe einen anderen aus -> Scheduling.

Aber was, wenn ein böswilliger Prozess in Betrieb genommen wird, der z.B. in den Speicherbereich anderer Prozesse schreibt oder einfach nur eine Endlosschleife ausführt -> andere Prozesse blockt?

Dafür mussten **Preemptions** und **Speicherbereichsschutz** eingeführt werden. Mehr dazu unten

Mehrbenutzersysteme

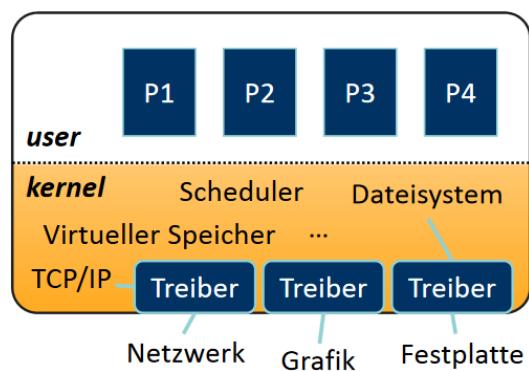
Die Idee war, dass Ressourcen nur dann an einen Nutzer zugewiesen werden, wenn sie wirklich benötigt werden – ansonsten würden n Nutzer das System n mal langsamer machen.

Dabei kann aber immer noch ein User zu viele Ressourcen verbrauchen, bzw. der Gesamtspeicherverbrauch zu hoch werden. Es musste also eine

Verteilungsstrategie, sowie eine **Speicher-Virtualisierung** eingeführt werden.

Arten von Kernels

Man nennt daher die Ausführung von Anwendungen „unprivilegiert“ (auch User-Mode), das Betriebssystem selbst „privilegiert“ (auch Kernel-Mode). Gewisse Operationen sind dann nur im Kernel-Mode erlaubt.

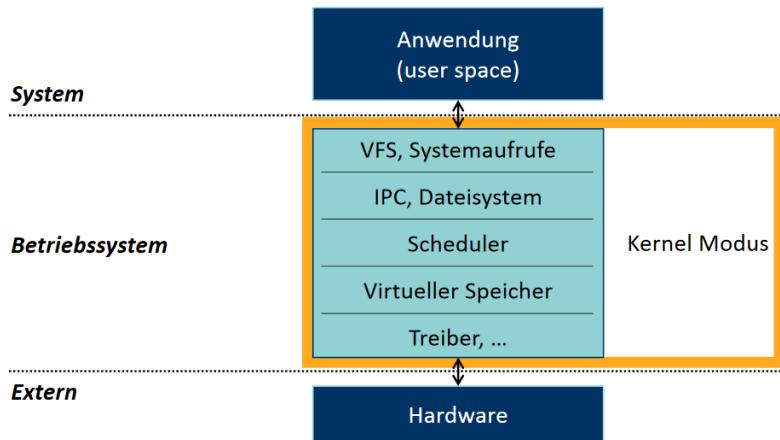


Grundsätzlich lässt sich die Betriebssystemarchitektur auf drei verbreitete Möglichkeiten eingrenzen:

Monolithische Kernels

Dabei läuft das gesamte Betriebssystem im privilegierten Modus, weshalb kein Schutz zwischen den Komponenten garantiert werden kann. Dafür kann das OS mit dynamisch geladenen Modulen erweitert werden, es muss kein Kontextwechsel innerhalb des OS erfolgen und eine allgemein gute Performance kann erzielt werden.

Linux, Unix und Windows (nur alle 9x Versionen) verwenden einen monolithischen Kernel.



Mikrokernels

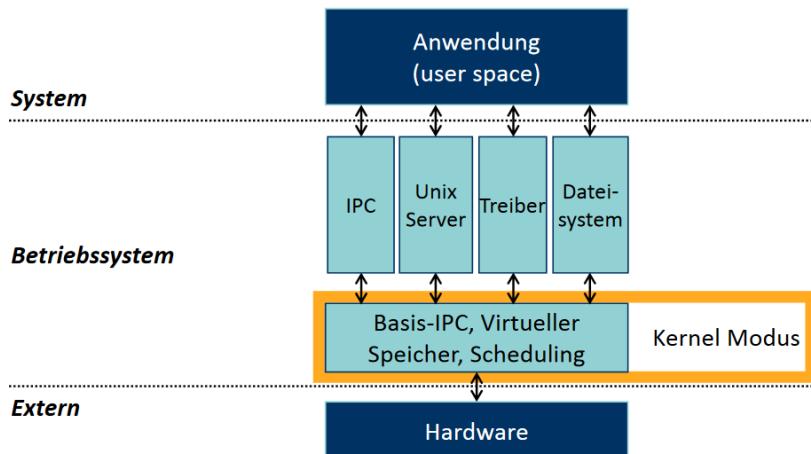
Die Kernels stellt nur Minimalfunktionalität bereit, während andere Dienste in Serverprozesse ausgelagert werden, welche wiederum im User-Mode ausgeführt werden. Damit eignet sich der Mikrokernel gut für verteilte Systeme, da diese über Nachrichten kommunizieren können.

Der Mikrokernel realisiert dabei den Prozesswechsel, maschinennahe Speicherverwaltung, Interprozesskommunikation und die Verwaltung von E/A-Adressen und Interrupts.

Der Vorteil dieser Architektur besteht in der einheitlichen Schnittstelle für Dienste auf Benutzer- und Kernebene, die Erweiterbarkeit durch neue Dienste und die Flexibilität. Mikrokernels sind auch sehr Zuverlässig und Portabel.

Dafür gibt es viele Kontextwechsel und teilweise zu viel Kommunikation → Overhead.

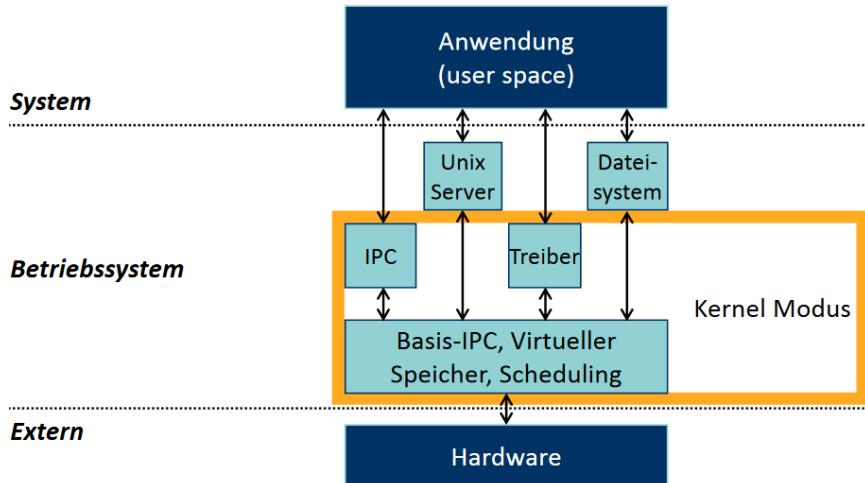
Mach Unix von CMU realisiert bspw. einen Mikrokernel.



Hybrid-Kernels

Ein Hybrid-Kernel verwendet Mikrokernell-Architektur, führt aber manche Komponenten ähnlich den monolithischen Kernels direkt aus. Damit soll eine bessere Performance, Sicherheit und Verwendbarkeit ermöglicht werden.

Diese Architektur wurde von Windows NT, BeOS und NetWare verwendet.

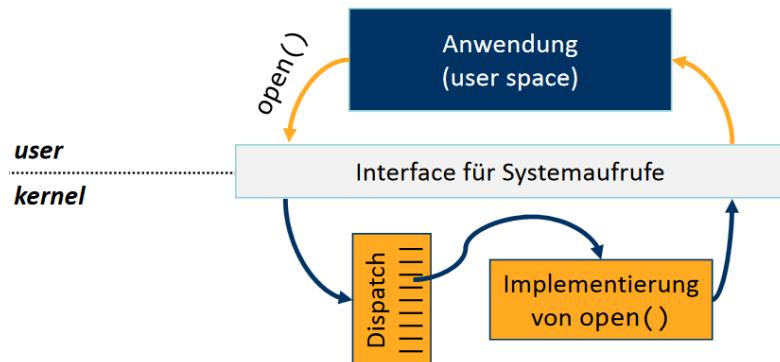


Arten von Betriebssystemen

- ∅ Personal Computer-OS
- ∅ Server-OS
 - viele Benutzer gleichzeitig, Netzwerkanbindung
- ∅ Multiprozessor-OS
 - für Parallelrechner (heute genereller Fall)
- ∅ Echtzeit-OS
 - Scheduling erlaubt Zeitgarantien
- ∅ Anwendungsspezifische OS
 - z.B.: Für eingebettete Systeme (Autos, Industrie, etc.)
 - Spielekonsolen, Handys, Tablets

Systemaufrufe

System Calls ermöglichen es Anwendungen die Kernel-Funktionalität zu aktivieren. Das heißt, die Anwendung übergibt die Kontrolle freiwillig an den Kernel, welcher den passenden Handler au ruft.



Damit lassen sich also Aktionen durchführen, die im user-mode nicht möglich wären. Dazu muss aber der Kernel aber ein SystemCalls-Interface anbieten, über welches die Anwendungen Argumente setzen können und sogenannte Kernel Traps auslösen können. Der Kernel führt die gewünschte Aktion aus und gibt dann das Resultat zurück.

So ist beispielsweise `printf()` in C eine user-level Erweiterung vom System-Call `write()`.

Fehlerbehandlung

Damit man mit Fehler umgehen kann, muss ein SystemCall auch eine Rückgabe haben. Diese werden beispielweise in sys/errno.h gelistet.

Systemkontakte

Grundsätzlich gibt es mehrere Kontexte in einem OS:

- ⌚ User-Level: führt eine Anwendung aus
 - ⌚ Kernel-Prozess: z.B.: Systemaufruf
 - ⌚ Kernel Code wird ohne speziellen Prozess ausgeführt
 - Timer interrupt
 - Device interrupt
 - ⌚ Context-Switch (Kontextübergänge)
 - ⌚ IDLE (Nix zu tun → CPU in low power state)

Kontextübergänge

| | |
|--|------------|
| User → Kernel-Prozess: | Syscall |
| User/Prozess → Interrupt handler: | Hardware |
| Kernel-Prozess → User/Kontextübergang: | Return |
| Kernel-Prozess → Kontextübergang: | Sleep |
| Kontextübergang → User/Kernel-Prozess: | Scheduling |

POSIX

In den folgenden Kapiteln wird oft der POSIX-Standard referenziert und verwendet. Der POSIX-Standard definiert eine einheitliche API über mehrere Systeme hinweg, die von der Implementierung unabhängig sind. Sie wurden eingeführt, weil die verschiedenen Systeme (vor allem die UNIX-Distros) alle eigene APIs definierten. POSIX war dann ein „vertraglicher“ Standard.

Prozesse

Einführung

Ein Prozess ist eine laufende Instanz eines Programms. Das Betriebssystem ist dabei für die Prozesse verantwortlich (Erzeugung, Terminierung, Scheduling, Synchronisierung, Kommunikation, Deadlocks). Damit das OS all diese Aufgaben bewältigen kann, braucht es eben Prozesse, denen es Ressourcen zuweisen kann und sie voneinander schützen kann.

Durchsatz

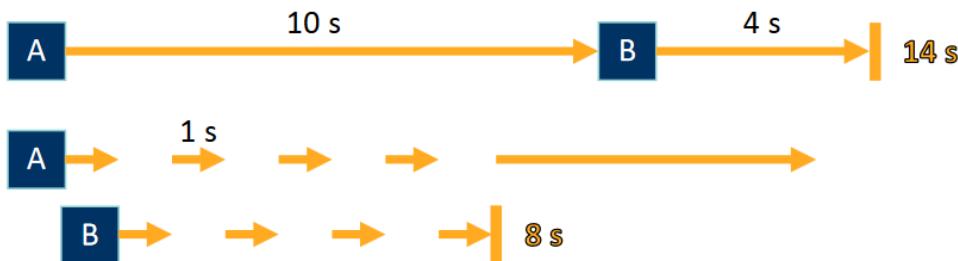
Dabei verbessern mehrere Prozesse die CPU-Auslastung:



Muss ein Prozess warten (z.B.: Benutzerinteraktion) kann ein anderer die CPU nutzen.

Wartezeit (Latenz)

Multi-Prozess-OS können die Latenz verringern:



Werden Prozesse hintereinander ausgeführt, dauert es 14 Sekunden bis B fertig ist.
Werden sie gleichzeitig ausgeführt, kann Prozess B bereits nach 8 Sekunden fertig sein.

Prozesshierarchien

Ein Parent-Prozess kann einen Child-Prozess erzeugen, dieser wiederrum kann selbst wieder einen Prozess erzeugen.

In UNIX wird dadurch eine Hierarchie (man nennt es dort „Prozessgruppe“) definiert. Windows hingegen kennt keine Prozesshierarchien – dort sind alle Prozesse gleichwertig!

Prozessperspektive auf das System

Jeder Prozess sieht das (Gesamt-)System auf seine eigene Sichtweise:

- ∅ Eigener Adressraum
 - so ist “*(int*) 0xc000” in unterschiedlichen Prozessen ein unterschiedlicher Wert
- ∅ Eigene Datei-Handler
- ∅ Eigene „virtuelle“ CPU (durchgesetzt durch Preemptions)

Prozess-Verwaltung aus Programmierersicht

Prozess erzeugen

Um aus einem Elternprozess einen Kindprozess zu erzeugen, können wir

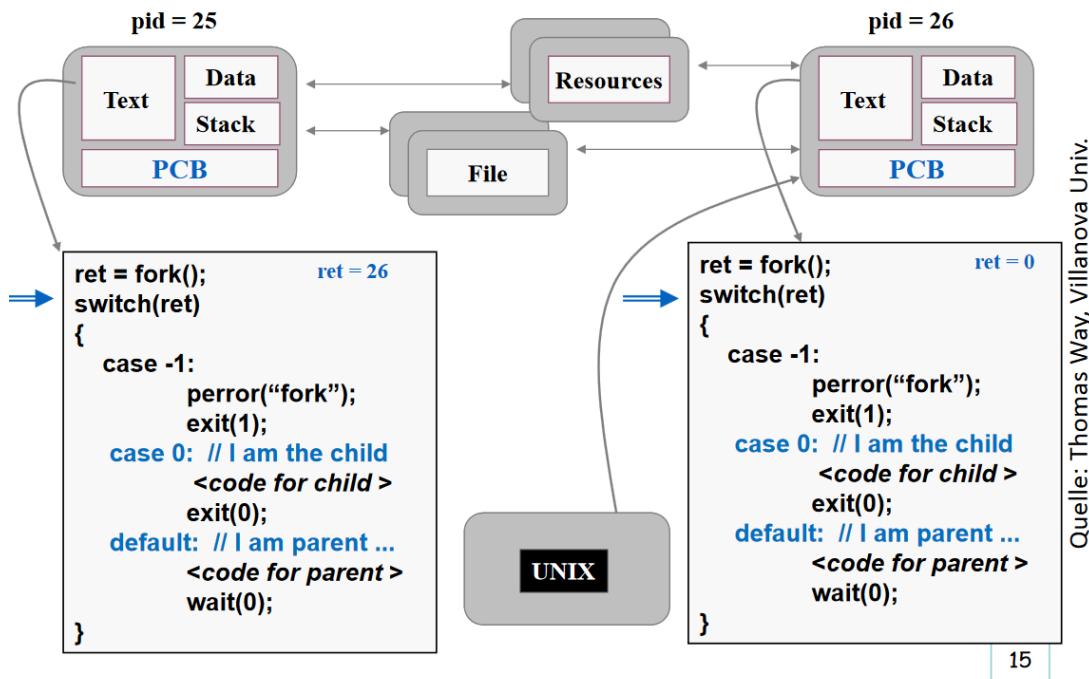
```
fork(void);
```

verwenden. Damit wird eine Kopie des Speichers und der File-handler (von offenen Files) erzeugt und beide Prozesse (Kind und Parent) führen den Code simultan ab dem Fork aus.

Der Rückgabewert ist

- ∅ die Prozess-ID des neuen Prozesses im Parent-Prozess
- ∅ 0 im Child-Prozess.

Funktionsweise/Nutzung von Fork:



Quelle: Thomas Waw, Villanova Univ.

fork (und vfork [*siehe unten*]) bieten ein sehr einfaches Interface, um ohne Argumente einen Prozess zu erzeugen – manchmal ist es aber sinnvoll, ein manuelles fork durchzuführen (was allerdings sehr aufwändig ist)

Manuelles erzeugen

Beispiel aus der Windows-API:

```

BOOL WINAPI CreateProcess(
    _In_opt_     LPCTSTR lpApplicationName,
    _Inout_opt_   LPTSTR lpCommandLine,
    _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_          BOOL bInheritHandles,
    _In_          DWORD dwCreationFlags,
    _In_opt_     LPVOID lpEnvironment,
    _In_opt_     LPCTSTR lpCurrentDirectory,
    _In_          LPSTARTUPINFO lpStartupInfo,
    _Out_         LPPROCESS_INFORMATION lpProcessInformation
);

```

Wenn du mehr wissen willst, geh googlen oder manpagen.

Auf Prozess warten

Waitpid wartet auf das Ende (oder ein Signal) von Kindprozessen.

```
int waitpid(int pid, int *stat, int opt);
```

- ∅ pid: Prozess-ID des Prozesses, auf den gewartet werden soll
 - -1: beliebiger Kindprozess
 - 0: beliebiger Kindprozess in der gleichen Prozessgruppe
 - >0: Prozess mit dieser ID
- ∅ stat: Speicherstelle für zusätzlichen Rückgabewert des Programms
 - Rückgabewert: Prozess-ID oder -1 bei Fehler (z.B. Kind bereits terminiert)

- ∅ opt: Generell 0 oder WNOHANG (wartet nicht)

Als Rückgabewert wird entweder die Prozess-ID des Kindprozesses, auf den gewartet wurde – oder aber 0 falls WNOHANG gesetzt und kein Kindprozess terminiert wurde.

Prozess beenden

```
void exit(int status);
```

Aktueller Prozess hört auf zu existieren. Es werden alle offenen Files und Verbindungen geschlossen und der Speicher freigegeben.

Status ist der Wert, der z.B. an waitpid übergeben wird. Dabei ist als Konvention 0 == erfolgreich, sonst Fehler-ID (8-Bit)

```
int kill(int pid, int sig);
```

Sendet das signal sig an den Prozess pid

SIGTERM wird häufig verwendet, um den Prozess zu beenden, kann aber vom Prozess selbst abgefangen werden.

SIGKILL ist „stärker“, es beendet den Prozess zuverlässig.

```
Kill -N pid
```

Sendet signal über Kommandozeile, z.B.: „kill -9 pid“ (9 = SIGKILL)

Signal Handler / Bearbeiten von Signalen

```
void(*signal(int sig, void (*func)(int)))(int);
```

Installiert einen Signalhandler „ func() “ für das Signal sig.

Sobald das Signal sig auftritt, wird die Funktion func() aufgerufen, wobei es viele Ausnahmesituationen gibt:

- ∅ Signal während Signal verarbeitet wird
- ∅ Verfügbarkeit von Ressourcen während Fehler/Abbruchsignal
- ∅ Weiterverarbeitung nach Fehlersignal

Hierbei ist wichtig, dass manche Ausnahmesituationen genau spezifiziert, andere aber abhängig von der Implementierung sind. Genauso muss das OS eine Signalmaske pro Prozess verwalten.

Beispiel

```
void handler(int signal) {
    // z.B. Datenstrukturen aufräumen
}

int main(int argc, char **argv) {
    signal(SIGINT, &handler);
    // ...
}
```

Programme ausführen

```
int execve(char *prog, char **argv, char **envp);
```

- ∅ prog: Pfad zu dem auszuführenden Programm
- ∅ arg: Array aus Argumenten für main() in prog

∅ envp: Umgebungsvariablen (wie PATH)

execv ist ein System-Call, der den aufrufenden Prozess in einen neuen Prozess umwandelt und das Programm prog ausführen lässt.

execv = execve, aber aktuelle Umgebungsvariablen werden verwendet

Es folgt kein return zum aufrufenden Prozess – das Return wird durch ein exit() ersetzt, welches das wait() vom aufrufenden Prozess freisetzt. Das heißt für uns ganz einfach: Es wird kein neuer Prozess erzeugt, sondern ein neues Programm in den aktuellen (aufrufenden) Prozess geladen.

Dabei gibt es noch verschiedene Wrapper-Funktionen:

```
int execvp (char *prog, char **argv); //Sucht prog in PATH, verwendet aktuelle Umgebungsvariablen  
int execlp(char *prog, char*arg, ...); //Liste von Argumenten, endet mit NULL
```

Generisches Erzeugen eines Prozesses

Mit den neu gelernten Befehlen von oben können wir ein „generisches“ Code-Fragment für den Elternprozess definieren:

```
int childPid;  
char * const argv[ ] = {...};  
  
main {  
    childPid = fork();  
    if(childPid == 0)  
    {  
        // I am child ...  
        // Do some cleaning, close files  
        execv(argv[0], argv);  
    }  
    else  
    {  
        // I am parent ...  
        <code for parent process>  
        wait(0);  
    }  
}
```

Damit lässt sich allgemein ein neuer Prozess aufrufen, in welchem ein anderes Programm gestartet wird.

Den meisten Aufrufen von fork() folgt exec*(). Folglich ist das Erstellen der Kopie des Elternprozesses reiner Overhead, der sofort wieder zerstört wird. Deshalb haben sich schlaue Menschen gedacht – warum nicht ein fork() implementieren, welches einen Prozess ohne Speicherkopie erzeugt?

```
pid_t vfork(void);
```

fork() blieb aber bestehen, weil es auch dafür einige Anwendungszwecke gibt – z.B. um früheren Zustand wiederherzustellen (Backup von Elternprozess).

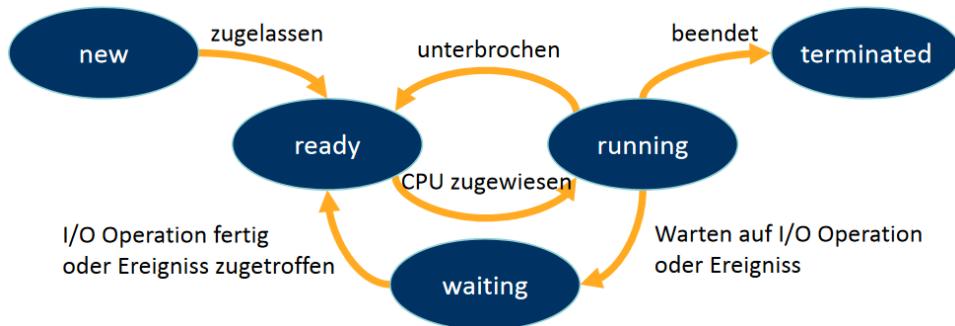
Kernelsicht von Prozessen

Prozesszustände

In Abhängigkeit von internen und externen Ereignissen kann ein Prozess in verschiedenen – aber zu jedem Zeitpunkt in genau einem – Zustand befinden.

Typische Zustände (die von OS zu OS unterschiedlich sind) wären:

- ∅ New: Prozess wurde soeben erzeugt
- ∅ Ready: Prozess wartet auf Prozessor
- ∅ Running: Instruktionen werden ausgeführt
- ∅ Waiting: Prozess wartet auf ein Ereignis (z.B.: I/O)
- ∅ Terminated: Prozess ist beendet



Nun tritt die Frage auf, welcher Prozess ausgeführt werden soll, wenn mehrere Prozesse den Zustand „Ready“ ausweisen? Genau dafür ist **Scheduling**.

Process Control Block (PCB)

Für die Implementierung eines Prozesses durch das Betriebssystem eine Datenstruktur für jeden Prozess angelegt werden. Diese wird **Process Control Block** (PCB) genannt und in UNIX durch proc, in Linux durch task_struct definiert.

Der PCB speichert den Zustand von Prozessen, verwaltet alle Informationen die für die Ausführung notwendig sind (Register, virtueller Speicher, ...) sowie zusätzliche Daten wie Rechte, Priorität, etc.

Verwaltung

Die Verwaltung erfolgt über eine Prozesstabelle. Häufig werden die Prozesse in Warteschlangen, basierend auf den Prozesszustand abgspeichert.

| |
|------------------------|
| Prozess-ID |
| Prozesszustand |
| Benutzer-ID |
| Program counter |
| Register |
| Virtueller Addressraum |
| Offene Dateien |
| ... |
| PCB |

Erzeugung

Soll ein Prozess erzeugt werden, muss das Betriebssystem einen PCB für den Prozess erzeugen. Dabei werden die meisten Einträge des Vaterprozesses kopiert, Abrechnungsinformationen initialisiert und eine eindeutige PID generiert.

Danach muss ein Speicherbereich zugeteilt werden – eventuell auch alle/Teile der Speichersegmente des Vaterprozesses kopiert werden. Das Programmsegment ist nach dem Erzeugen ident mit dem Vaterprozess – wird also mit dem Parent geteilt – sofern kein neues Image geladen wird. Abschließend wird der PCB in die Prozesstabelle eingereiht.

Beenden

Normalerweise erfolgt das Beenden durch das Ausführen der letzten Instruktion.

In gewissen Fällen kann aber auch der Kernel seine Privilegien ausspielen und den Prozess terminieren, z.B. wenn die verfügbaren Ressourcen überschritten werden oder illegale Operationen ausgeführt werden.

Dabei schickt das OS ein Signal (SIGCHILD) an den Elternprozess, um ihn über das Prozessende zu informieren. Speicher & Ressourcen werden – wenn nach dem Signal nicht bereits durch den Parentprozess geschehen – freigegeben und der PCB in der „terminated“-Queue eingereiht.

Context Switch

Darunter versteht man den (Prozessor-)Wechsel von einem Prozess zu einem Anderen. Es muss also der Zustand des aktuellen Prozesses gespeichert und der Zustand des neuen Prozesses geladen werden. (Dazu gehören Register, Adressraumwechsel, Cache, ...)

Wie du dir denken kannst, ist dieser Context-Switch sehr, sehr kritisch! Er kann nur durchgeführt werden, wenn der Kernel über die CPU verfügt.

Dazu kann der Prozess durch Preemptions – also ein Trap (Auftreten eines Fehlers) oder ein Interrupt (z.B. wegen Nachricht) – oder auch durch Cooperatives, also explizite Aufgabe der CPU durch Benutzerprozess (wie I/O Operationen) die Kontrolle verlangen.

Preemptions

Damit ist die Möglichkeit, Ressourcen aktiv zurückzunehmen, wenn sie anderweitig gebraucht werden, gemeint. Als Zwischenstück zwischen Ressourcen und Anwendungen ist dies ein Privileg des OS/Kernels.

So kann der Kernel z.B.: einen Hardware-Timer programmieren, der die Ausführung alle 10ms unterbricht. Die dazu nötigen I/O-Register dürfen nur im Kernel Mode geschrieben werden (also ein Programm kann den Timer nicht ändern). Wird der Interrupt ausgelöst, bekommt ein anderer Prozess die CPU.

Andere Beispiele für Preemptions wären Device-Interrupts (Paket im Netzwerk angekommen, ...) und Fehler (page fault, illegal instruction, ...)

Cooperative

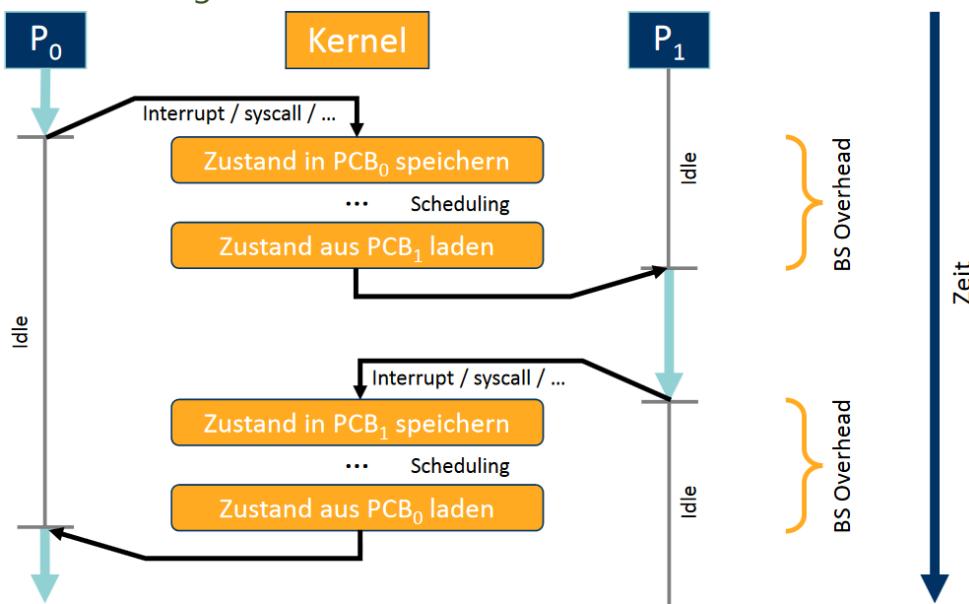
Wie der Name es schon vermuten lässt, ist dies ein vom jeweiligen Prozess kooperatives Unterfangen – System Calls, Sleep oder das explizite Starten eines anderen Prozesses führt dazu, dass der Kernel die Kontrolle freiwillig bekommt.

```
int main() {
    int x = 5+8;
    x = x*x;
    x = -x;
    printf("Bla %d\n", x);
    int y = 5;
    yield();
    x = x+y;
    return 0;
}
```

Cooperative oder
Preemptive Context
Switch möglich

Preemptive Context
Switch möglich

Visualisierung eines Context Switches



Man kann vermutlich schon selber Schlussfolgern, dass ein Context-Switch sehr Hardwareabhängig ist. Programcounter, (Spezial-)Registerinhalte müssen abgespeichert werden, zusätzliche PCB-Werte (die ja auch wieder Hardwareabhängig sind) müssen auch abgespeichert werden – abschließend muss die eigene, virtuelle Adresstabelle geändert werden.

Daraus resultieren relativ hohe Kosten, für die man Lösungen braucht, z.B.:

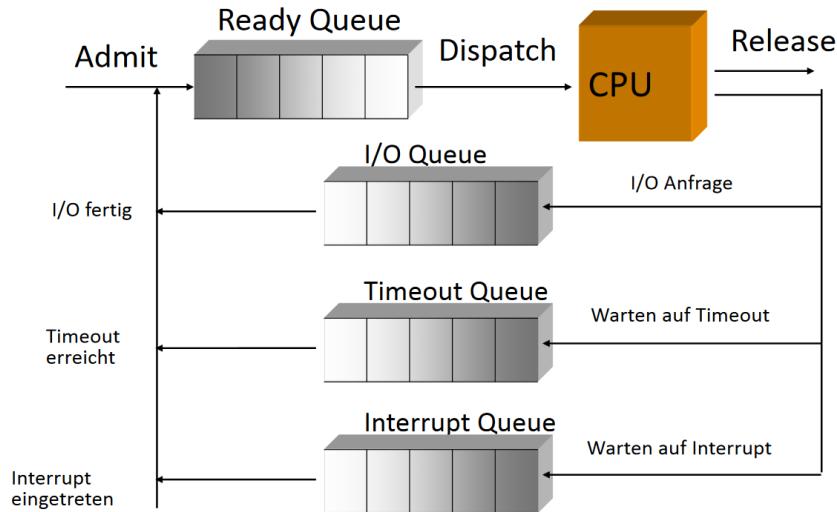
- Spezialregister nur speichern, wenn verwendet
- Werte aus dem TLB (Adressübersetzungshardware) nur löschen, wenn sie zu dem Prozess gehören.
- ...

Scheduling

Grundsätzlich muss die Entscheidung getroffen werden, welcher Prozess wann auf welchem Prozessor ausgeführt wird. Dabei müssen viele Faktoren berücksichtigt werden, wie Priorität, Zeit die ein Prozess schon wartet, etc.

Dabei erfolgt Scheduling bei den folgenden Zustandsübergängen

| | |
|---|-----------------------------|
| <i>Von Running in Waiting (zB I/O Anforderung)</i> | → non-preemptive-scheduling |
| <i>Von Running in Ready (zB Interrupt)</i> | → preemptive-scheduling |
| <i>Von Waiting in Ready (zB I/O Operation fertig)</i> | → preemptive-scheduling |
| <i>Von Ready in Terminated</i> | → non-preemptive-scheduling |



Scheduling = Auswahl eines Prozesses zur Exekution aus verschiedenen Warteschlangen

Grundlegende Begriffe

| Begriff | Erklärung |
|-----------------------------|-------------------------------------|
| Laufzeit | Zeit von Start bis Ende |
| Antwortzeit / Latenz | Zeit von Anfrage bis erster Antwort |
| Wartezeit | |
| Prozessorschub | |

Scheduling Policy

Bestimmt die Ausführungsreihenfolge (Optimierung) von Prozessen durch einen Scheduler. Dabei müssen viele Ziele für optimales Scheduling erreicht werden.

Fairness (kein Prozess sollte „verhungern“)

Priorität (relative „Wichtigkeit“ von Prozessen beachten)

Deadlines (Prozess x muss vor gewissen Zeitpunkt terminieren)

Durchsatz (throughput, System sollte gut ausgelastet werden)

Anzahl an fertigen Jobs / Zeit, folglich ist Höher = Besser

Laufzeit (turnaround time, minimiere Zeit -Start bis Ende von Prozessen)

Antwortzeit (latency, Zeit von Anfrage bis erster Antwort)

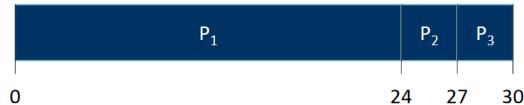
z.b: Mausbewegung bis Cursorbewegung, sollte also niedrig sein

Effizienz (Scheduler selbst sollte minimalen Overhead verursachen)

Leider gibt es keinen „besten“ Scheduler. Wir können aber die Kriterien allgemein ausdrücken durch:

*Maximiere Prozessorauslastung/Durchsatz,
Minimiere Laufzeit/Antwortzeit*

First-come First-Served (FCFS)



Jobs werden in der Reihenfolge ausgeführt in der sie eintreffen. Das ist logischerweise sehr einfach zu implementieren (FIFO-Queue) allerdings ist die Performance sehr dürftig.

$$\text{Durchsatz} = 3 \text{ Jobs} / 30 \text{ s} = 0,1 \text{ job/s}$$

$$\text{Laufzeit} = P_1: 24 \text{ s}, P_2: 27 \text{ s}, P_3: 30 \text{ s} \rightarrow \text{Durchschnitt: } 27 \text{ s}$$

Würden wir die Jobs umgekehrt ausführen, hätten wir zwar immer noch denselben Durchsatz, aber eine andere Laufzeit:

$$\text{Laufzeit} = P_1: 30 \text{ s}, P_2: 6 \text{ s}, P_3: 3 \text{ s} \rightarrow \text{Durchschnitt: } 13 \text{ s}$$

Somit kann ein guter Schedulingalgorithmus die Laufzeit reduzieren. Ein Verbessern der Wartezeit verbessert auch die Antwortzeit – allerdings ist bei FCFS-Scheduling die Wartezeit nicht optimal.

Auch Ressourcen können nicht parallel genutzt werden.

FCFC Konvoi Effekt

Ein prozessorlimitierter Job hält den Prozessor bis zum Ende – oder aber bis zu einer I/O-Operation. Letzteres kommt in prozessorlimitierten Jobs allerdings selten vor.

Daraus resultieren lange Perioden ohne I/O Anfragen – allerdings mit Prozessor.

Das ist eine schlechte Auslastung der I/O Geräte. Man nennt diesen Effekt

Konvoi-Effekt. D.h.: Alle Jobs warten auf den prozessorlimitierten Job, bis dieser den Prozessor freigibt.

Shortest Job First (SJF)

- Exekutiere den Job mit der geringsten (Rest)laufzeit
- Vorteile: Optimale durchschnittliche Laufzeit für Jobs mit variierender Länge
- Nachteile: Vorhersage, unfair

SJF setzt es sich zum Ziel die Laufzeit zu minimieren. Dafür wird ein Job gewählt, dessen nächster Prozessorschub der kürzeste ist. Sollten mehrere Jobs die gleiche Prozessorschubzeit haben, dann kommt FCFS zum Einsatz.

SJF sollte eigentlich „shortest next CPU burst“ genannt werden, weil es sich nicht auf die Gesamtlaufzeit von Jobs bezieht

Dabei wird SJF in zwei Varianten realisiert:

- ⊖ Non-Preemptive: Wurde ein Job gewählt, kann dieser nicht unterbrochen werden, bis sein Prozessorschub vorbei ist
- ⊖ Preemptive: Wenn ein neuer Prozess verfügbar wird, dessen Prozessorschublänge geringer ist als die verbleibende Zeit des aktuellen Prozesses ist, wird dieser unterbrochen
 - Man nennt dies auch *Shortest Remaining Time First (SRTF)*

Was optimieren SJF/SRTF?

Resultiert in der beweisbar, minimalen durchschnittlichen Wartezeit:

SJF für nichtunterbrechende Scheduling Algorithmen

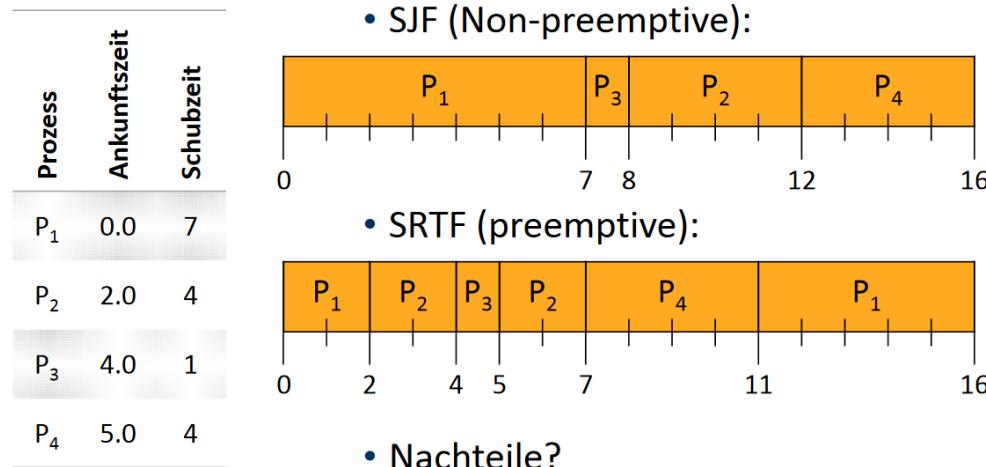
SRTF für unterbrechende Scheduling Algorithmen

SRTF ist mindestens so gut wie SJF

Dafür muss aber die Laufzeit aller Jobs bekannt sein und zu einem bestimmten Zeitpunkt vorliegen. Scheduling erfolgt dann ab diesem Zeitpunkt. Ziel ist es dann, die durchschnittliche Laufzeit für Jobs mit variierender Laufzeit zu minimieren --> SRTF, damit kurze Jobs nicht durch lange Jobs verzögert werden.

Das funktioniert, indem die Wartezeit minimiert wird, was häufig (aber nicht immer) auch in kürzeren Laufzeiten resultiert.

Beispiel



Nachteile

SJF minimiert nur die Wartezeit (und damit die Antwortzeit). Das kann aber zu einer unfairen Verteilung führen – man spricht auch vom „Verhungern“ von Jobs.

In der Praxis stellt sich aber ein noch viel größeres Problem: Zukünftige Prozessorschübe können nur schwer vorhergesagt werden. Die Abschätzung basiert auf bisherigem Verhalten des Jobs.

Abschätzung der Prozesslaufzeit

Prozesslaufzeiten können gemessen und gespeichert werden. Dann wird die Laufzeit auf Basis von historischen Messwerten vorhergesagt:

T_i i-te ($1 \leq i \leq n$) gemessene Ausführungszeit eines Prozesses
Zeit im Zustand „Running“

L_i vorhergesagter Wert für die i-te Instanz

L₁ vorhergesagter Wert für die erste Instanz; nicht berechnet

Somit lässt sich eine einfache Abschätzung der Ausführungszeit L durchführen mithilfe des Mittelwertes:

$$L_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i = \frac{1}{n} T_n + \frac{n-1}{n} L_n$$

Dabei lässt sich aber auch eine höhere Gewichtung von jüngeren Messungen miteinberechnen: **Exponentieller Mittelwert**. Diese sind meist genauere

Vorhersagen für aktuelle Bedingungen. Dazu führen wir eine Zeitreihe vergangener Werte ein und definieren α als konstanten Gewichtungsfaktor mit $0 < \alpha < 1$

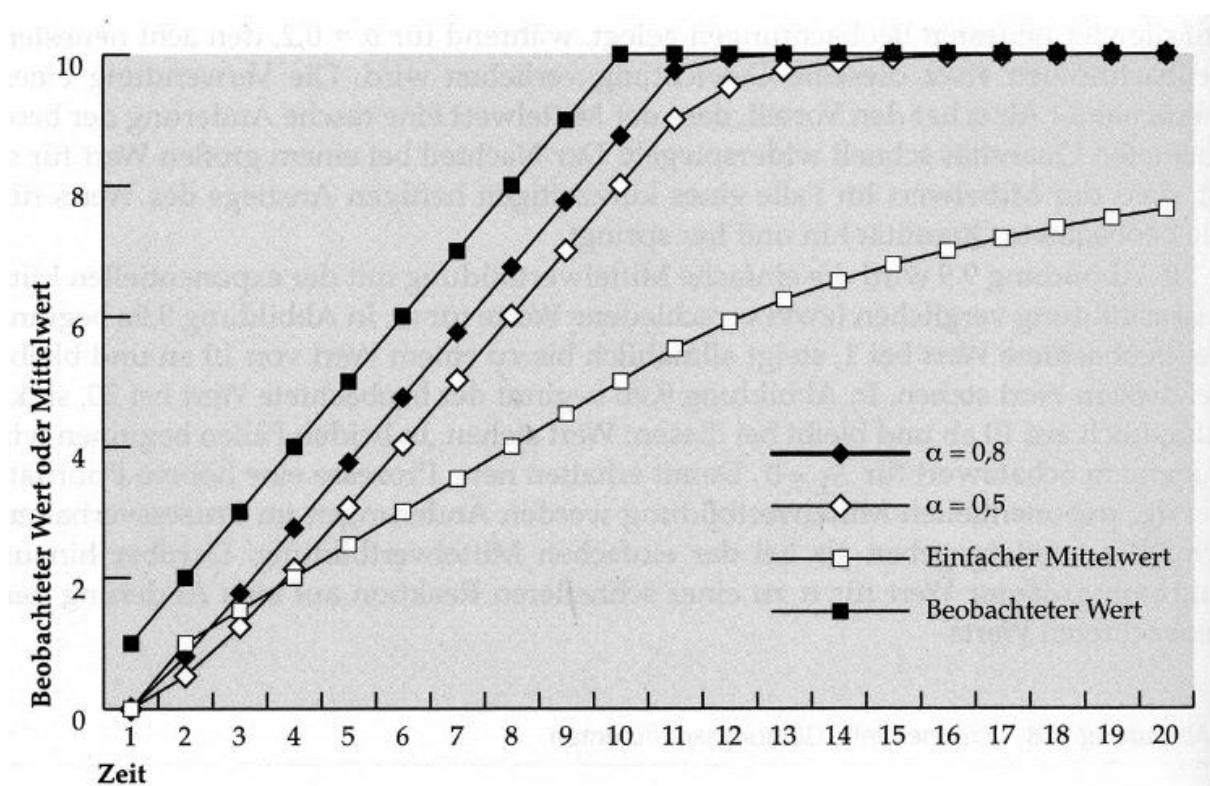
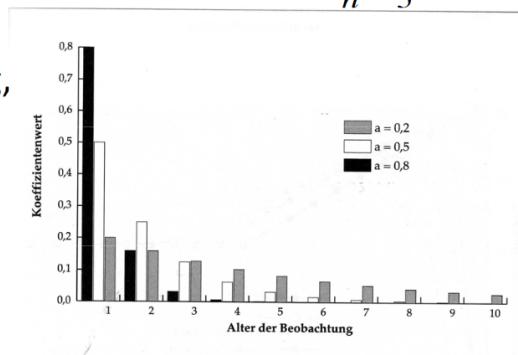
Die letzte Messung erhält also höhere Gewichtung als ältere Messungen:

$$L_{n+1} = \alpha T_n + (1 - \alpha)L_n$$

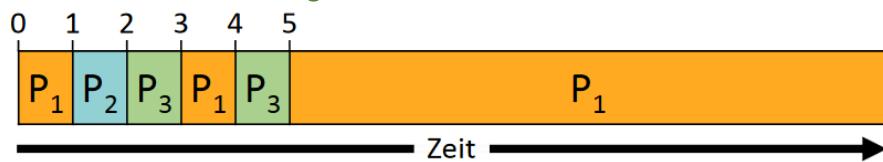
Beispiel: $\alpha = 0,8$

$$L_{n+1} = 0,8T_n + 0,16T_{n-1} + 0,032T_{n-2} + 0,0064T_{n-3} + \dots$$

Je älter die Beobachtung, umso weniger trägt der Wert zur Mittelwertbildung bei.



Round Robin Scheduling



Beim Round Robin Scheduling wird aus der Ready Warteschlange auf Basis eines Round-Robin Prinzips Jobs zyklisch auserwählt. Dies löst die Probleme der Fairness und des Verhungerns von Prozessen.

Prozesse werden nach fixem Zeitintervall (Slice oder Quantum) unterbrochen. Der Prozess wird dann ans Ende der FIFO Queue gestellt. Jobs, welche zwischenzeitlich entstehen, kommen vor diesem Job in die Queue. Lange Zeitintervalle realisieren FCFS (Antwortzeitproblematik), während kurze Zeitintervall die Zahl der Kontextwechsel erhöht und zu schlechterer CPU Auslastung führt.

Round-Robin wird aufgrund der fairen Verteilung der Prozessorressourcen auf Jobs und der guten Antwortzeit (falls wenig Jobs) sehr häufig in der Praxis verwendet.

Nachteile

Variierende Job-Größen funktionieren gut, allerdings nur, wenn die Jobs unterschiedlich groß sind. Hätten wir zwei Jobs mit je 100s Laufzeit:



Durchschnittliche Laufzeit mit RR: 199,5s

Durchschnittliche Laufzeit mit FCFS: 150s

Die durchschnittliche Wartezeit und Laufzeit längerer Jobs ist häufig sehr lange – selbst wenn der Context Switch gleich null wäre!

Zeitintervall / Quantum

| Jobzeit = 10 | Quantum | Context Switches |
|--------------|---------|------------------|
| 0 | 12 | 0 |
| 0 | 6 | 1 |
| 0 | 1 | 9 |

Das Quantum sollte so gewählt werden, dass ein Großteil der Prozessorschübe kürzer als das Quantum sei. Auch sollte das Quantum größer als der Zeitverlust durch Context-Switch sein.

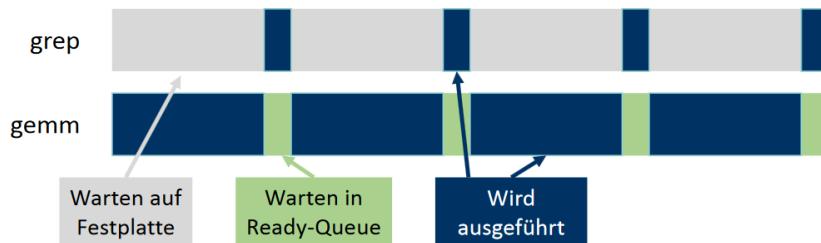
Es darf aber auch nicht so groß sein, dass das Resultat äquivalent zu FCFS ist (Antwortzeiten erhöhen sich), aber auch nicht zu klein (Durchsatz verschlechtert sich).

Typischerweise beträgt das Quantum 10-100ms. Damit ergibt sich eine Zeit von ca. 0,1ms – 1ms für die Context-Switches.

Prozessor und andere Geräte

Weil nicht nur der Prozessor ausgelastet sein sollte, sondern auch alle anderen Geräte, wird das Scheduling ein wesentlich schwierigeres Unterfangen (als es eh schon ist). So kann durch „perfektes“ Scheduling, wenn 1 Prozessor und n I/O-Geräte vorhanden sind, eine „ $n+1$ “-fache Performanceverbesserung erreicht werden.

Hier ein Beispiel mit einem Prozessor und einer Festplatte:



Durchsatz maximieren

Durch das Überlappen von I/O und Berechnungen mehrerer verschiedener Jobs können wir den Durchsatz maximieren. Das bedeutet, Prozessorauslastung & I/O Geräteauslastung wird maximiert, was nicht notwendigerweise gut für die Antwortzeiten sei.

Fortgeschrittenes Scheduling

Zwei-Stufen Scheduling

Das Zwei-Stufen Scheduling berücksichtigt den Umstand, dass ein Wechsel zu Prozessen, die auf die Festplatte ausgelagert sind, sehr zeitintensiv ist. Beispielsweise benötigt ein Festplattenzugriff 10ms. Bei einer 3GHZ CPU wären das 30 Millionen Cycles!

Das Zwei-Stufen Scheduling schaltet daher nur Prozesse im Hauptspeicher für „eine Weile“ – dann tauscht er (teilweise) Prozesse auf der Festplatte aus. Das Zwei-Stufen Scheduling muss daher ein Subset und eine Definition für „eine Weile“ voraussetzen --> Wähle „Speicherquantum“ >> Zeitkosten f. Festplattenzugriff.

Prioritäten Scheduling

Jedem Job wird eine numerische Priorität zugewiesen, wobei in UNIX eine kleinere Zahl, höhere Priorität bedeutet. Dann wird die CPU an den Prozess mit der höchsten Priorität (FCFS bei gleichen Prioritäten) zugewiesen. Dabei kann es wieder Preemptive oder Non-Preemptive stattfinden.

SJF ist dabei eine Sonderform des Prioritäten Scheduling (unter der Prämisse, dass Abschätzung der nächsten Prozessurschubzeit gleich der Priorität ist)

Auch hier entsteht wieder die Gefahr des Verhungerns von niedrig priorisierten Prozessen. Auch liefert Priority Scheduling nicht immer die beste durchschn. Wartezeit. Letzteres Problem lässt sich aber mit einer steigenden Prozesspriorität mit steigender Wartezeit verhindern.

Prioritätenfestlegung

Kann statisch, also bereits im Vorfeld durch vorhersagbares Verhalten von Jobs und Belastung, oder aber dynamisch stattfinden. Dabei basiert eine Priorität auf:

- ∅ Kosten für Benutzer
- ∅ Bedeutung des Jobs
- ∅ Alter des Jobs
- ∅ Prozessoranteil in den letzten x Minuten

Lotterie-Scheduling

- Jeder Thread bekommt eine prioritätenabhängige Zahl von Losen (kurze Jobs bekommen mehr Lose)
- Reserviere eine Mindestzahl von Losen für jeden Job um Fortschritt und Fairness zu gewährleisten.

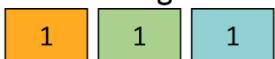
Man hat sich dabei an ökonomischen Grundsätzen inspirieren lassen. Der Scheduler verteilt „CPU-Lose“ an die Jobs und zieht zufällig ein Ticket. Dabei kann ein Job mehrere Lose bekommen (Prioritäten) – die Chance das nächste Quantum zu „gewinnen“ liegt also bei der Anzahl an Lose pro Prozess / Gesamtanzahl an Lose. Lose werden aber durch die Ziehung nicht verbraucht.

Das Hinzufügen/Löschen von Jobs beeinflusst dann alle existierenden Jobs proportional. Solange jeder Job mind. 1 Los hat, gibt's auch kein Verhungern.

Beispiel: 4 Jobs, 1 Los pro Job, jeder Job 1/4 CPU



1 Job wird gelöscht → verbleibende haben je 1/3 CPU



| # short jobs/ # long jobs | % of CPU each short job gets | % of CPU each long job gets |
|------------------------------|---------------------------------|--------------------------------|
| 1/1 | 91% | 9% |
| 0/2 | N/A | 50% |
| 2/0 | 50% | N/A |
| 10/1 | 9.9% | 0.99% |
| 1/10 | 50% | 5% |

Gibt es zu viele kurze Jobs, kann keine vernünftige Antwortzeit gewährleistet werden. Ist der Load nahe 100%, fällt die Performance. Man muss den Load also immer niedrig halten (z.B.: Nutzer abmelden).

Übertragung von Losen

Lose dürfen zwischen Jobs auch getauscht/übertragen werden. Das ist gut geeignet für Prozesskommunikation. Wartet zum Beispiel ein Client auf die Antwort des Servers, braucht er seine Lose nicht – er kann sie dem Server geben.

Kompensationslose

Nutzt ein Job B nur einen Anteil f seines Quantums, ein Job A aber das ganze Quantum voll, und haben beide gleich viele Lose, gewinnen beide gleich viel Prozessorzeit. Job A bekommt Anteil 0,5 – f der Prozessorzeit von B --> nicht fair.

Diesem Problem widmet sich das Kompensationslos. Wenn B nur Anteil f nutzt, erhöhe temporär Lose von B um $1/f$, bis der Prozessor von B wieder gewonnen wird.

Weil des jetzt wahrscheinlich nit kapiert hast, hier ein Beispiel:

Quantum = 100ms

Job B gibt CPU nach 20ms frei

d.h. $f = 0,2 = 1/5$

Kompensation: Alle Werte der Lose von B werden mit $1/0,2 = 5$ multipliziert, bis B die CPU erhält. Sobald B die CPU erhält, fallen die Kompensationslose wieder weg.

Kompensationslose bevorzugen I/O limitierte und interaktive Jobs und helfen diesen Jobs einen fairen Anteil der CPU zu bekommen.

Nachteile

Antwortzeit ist unvorhersagbar und man muss mit Fehlern rechnen. Die erwarteten Fehler belaufen sich dabei auf ungefähr $\sqrt{n_a}$ für n_a Entscheidungen. Das heißt, Job A sollte 1/3 der CPU bekommen, hat aber nach 1 Minute nur 19 Sekunden. Auch lassen sich diese Fehler kategorisieren in absolute Fehler (hier 1 Sekunde) und relative Fehler (1/60).

Zudem ist die Wahrscheinlichkeit k von n CPU-Quanten zu erhalten, binomial verteilt.

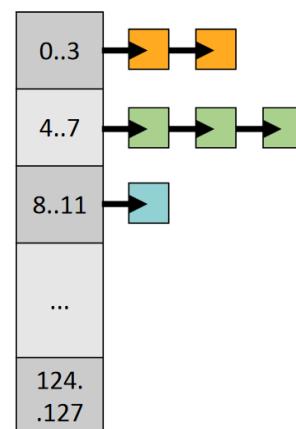
Scheduling Implementierungen

BSD Scheduling

- Mehrere Warteschlangen mit verschiedenen Prioritäten
- Automatische Anpassung von Prioritäten um SJF/SRTF zu erreichen.

Wird durch Multilevel Feedback Queues realisiert. Jeder lauffähige Job wird in einer von 32 Queues (Run Queues) eingereiht. Kernel wählt dann den ersten Job auf höchster Ebene. Mittels Round-Robin Scheduling wird zwischen Jobs auf derselben Ebene geschedult.

Prioritäten werden dynamisch berechnet. Jobs werden dann zwischen Queues bewegt, wobei Unterbrechungen stattfinden, falls ein neuer Job eine höhere Priorität als der aktuelle Job erhält. Damit wird die Idee, interaktive Prozesse (welche wenig CPU brauchen) zu bevorzugen.



Spezialfälle

Echtzeitscheduling

Es gibt zwei Kategorien:

Soft real-time

Falls die Deadline nicht erreicht wird, klingt die Musik kurz komisch.

Hard real-time

Falls die Deadline nicht erreicht wird, stürzt das Flugzeug ab.

Das System muss also periodische und aperiodische Events behandeln, z.B: Prozesse A, B und C müssen je alle 100, 200 und 500ms (Periode) ausgeführt werden, benötigen immer 50, 30 und 100ms (CPU).

Scheduling ist daher möglich, falls $\sum \frac{CPU}{Periode} \leq 1$ (ohne Context-Switch).

Dafür gibt es spezielle Scheduling-Strategien, wie z.B. First Deadline First. Dann hält der Job mit der am nächsten liegenden Deadline die höchste Priorität.

Multiprozessor Scheduling

Damit treten viele neue Probleme auf:

- ∅ Welcher Prozess auf welchem Prozessor?
- ∅ Bewegung zwischen Prozessoren bewirkt (Mehr-)Kosten
- ∅ Kommunizierende Prozesse/Threads auf demselben Prozessor
- ∅ ...

mehr dazu im Kapitel Multiprozessorsysteme.

Verteiltes Scheduling

Wir nehmen an, wir haben ein verteiltes System von unabhängigen Knoten. Wir möchten dann einen Job auf einem Knoten mit geringem Load ausführen. Allerdings ist jeden Knoten zu fragen sehr aufwändig und im Vergleich zu generellem Scheduling ist der Load unbekannt.

Wir wählen also einen Knoten zufällig – so funktionieren viele Dienste im Internet.

Besser ist es aber sogar, 2 Knoten zufällig zu wählen. Dann wird der Job an denjenigen mit weniger Load geschickt. Das Resultat ist nahezu optimal. Allerdings ist es nicht wesentlich vorteilhafter einen dritten Knoten abzufragen, weil es exponentiell konvergiert.

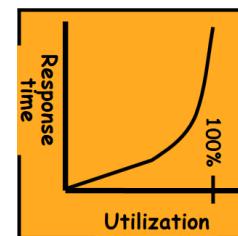
Scheduling Algorithmen in der Praxis

Batch-Systeme verwenden FCFS und SJF.

Interaktive Systeme hingegen sind normalerweise unterbrechend, d.h. Scheduling-Entscheidungen finden bei jedem neuen Quantum statt. Die Leistungskriterien für Interaktive Systeme sind minimale Antwortzeit und Ausgeglichenheit.

Vertreter sind Round Robin, Prioritäten-Scheduling, Shortest Next CPU-Burst, Lotterie-Scheduling und Multi-Level Scheduling.

In der Praxis sind Scheduling-Algorithmen relevant, wenn wenig Ressourcen vorhanden sind. Dabei sind die meisten Scheduling-Algorithmen nur im linearen Bereich (siehe Diagramm) sehr gut – ab dann versagen sie. Im „Knie“ des Diagramms sollte man dann mehr Ressourcen kaufen.



Synchronisierung

Nebenläufigkeit

Unter Nebenläufigkeit versteht man die gleichzeitige Ausführung von Prozessen oder Threads und ist ein grundlegendes Prinzip für die Entwicklung von Betriebssystemen. Nebenläufigkeit ist von zentraler Bedeutung für die Verwaltung von Prozessen und Threads.

- ∅ **Mehrprogrammbetrieb:** mehrere Prozesse in einem Einprozessorsystem.
- ∅ **Mehrprozessorbetrieb:** mehrere Prozesse in einem Mehrprozessorsystem.
- ∅ **verteilte Verarbeitung:** mehrere Prozesse, die auf verteilten Rechnersystemen ausgeführt werden.

Während Nebenläufigkeit hauptsächlich im **Mehrprogrammbetrieb** von Bedeutung ist, ist es weiteres auch in folgenden 2 Bereichen von Bedeutung:

- ∅ **strukturierte parallele Anwendungen:** Anwendung, die sich als Gruppe von nebenläufigen Prozessen strukturieren und programmieren lassen.
- ∅ **Betriebssystemstruktur:** Betriebssysteme werden häufig als Gruppe von nebenläufigen Prozessen oder Threads implementieren.

Zentrale Konzepte innerhalb der Nebenläufigkeit sind **Kommunikation zwischen Prozessen, gemeinsame Nutzung von Ressourcen, Wettbewerb um Ressourcen, Synchronisierung der Aktivitäten mehrerer Prozesse** und Zuteilung von Prozessorzeit zu Prozessen (**Prozessscheduling**).

| Nebenläufigkeit: "zu viel Milch Problem" | | |
|--|---|--------------------------|
| time | You | Your Roommate |
| 3:00 | Arrive home | |
| 3:05 | Look in fridge, no milk | |
| 3:10 | Leave for grocery | |
| 3:15 | | Arrive home |
| 3:20 | Arrive at grocery | Look in fridge, no milk |
| 3:25 | Buy milk | Leave for grocery |
| 3:35 | Arrive home, put milk in fridge | |
| 3:45 | | Buy Milk |
| 3:50 | | Arrive home, put up milk |
| 3:55 |  | Oh no! |

source: Prof. Saman Amarasinghe, MIT

Gemeinsame Ressourcen

Nebenläufige (parallele) Prozesse können sich gegenseitig beeinflussen, z.B.: über gemeinsamen Adressbereich oder gemeinsame Ressourcen (Files, Geräte, etc.)

Dabei können **Dateninkonsistenzen** entstehen. Zugriffe auf gemeinsame Ressourcen und Daten müssen daher kontrolliert und konsistent erfolgen. Eine optimale Ressourcenzuteilung ist sehr schwierig und Programmierfehler können sich nur schwer reproduzieren/lokalisieren lassen.

Race-Condition

Race-Conditions entstehen, wenn zwei verschiedene Threads auf dieselben Daten zugreifen wollen – und das wirklich zur gleichen Zeit. In den Vorlesungsfolien wird dabei oft von **Erzeuger/Verbraucher-Problem** gesprochen.

Als Beispiel (von Wikipedia): Wir wollen zwei Threads laufen lassen:

| Thread 1 | Thread 2 | Integer value |
|----------------|----------------|---------------|
| | | 0 |
| read value | ← | 0 |
| increase value | | 0 |
| write back | → | 1 |
| | read value | 1 |
| | increase value | 1 |
| | write back | 2 |

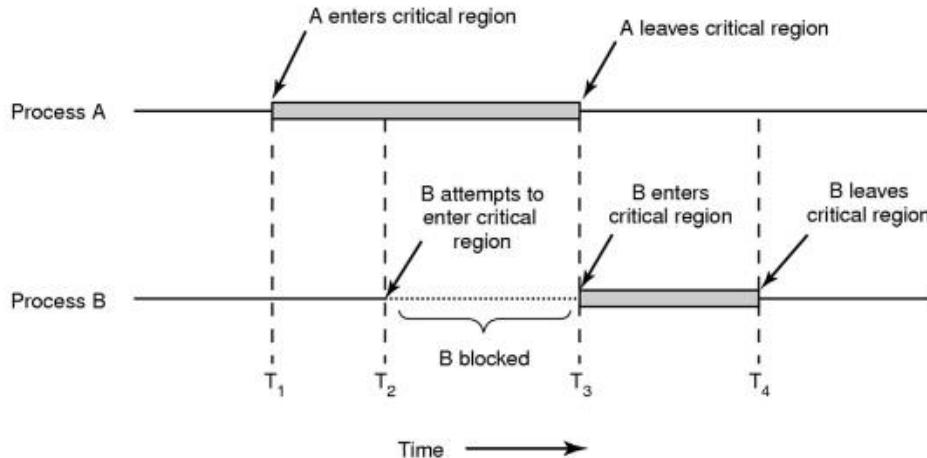
Aufgrund der Nebenläufigkeit entsteht eine Dateninkonsistenz --> Race-Condition. Der Grad der Dateninkonsistenz hängt dabei vom Scheduling ab.

| Thread 1 | Thread 2 | Integer value |
|----------------|----------------|---------------|
| | | 0 |
| read value | ← | 0 |
| | read value | 0 |
| increase value | | 0 |
| | increase value | 0 |
| write back | → | 1 |
| | write back | 1 |

Um Race-Conditions zu vermeiden, müssen nebenläufige Prozesse synchronisiert werden. Wir sprechen dabei von **Mutual Exclusion** (Wechselseitiger Ausschluss). Immer nur ein Prozess kann die gemeinsame Ressource zu einem bestimmten Zeitpunkt verwenden.

Kritische Region

Ist eine Programmregion, in der gemeinsame Daten geändert oder verwendet werden. Streben nun n Prozesse den Zugriff auf gemeinsame Daten an, dann hat jeder Prozess eine Code-Region (kritische Region) in der auf die gemeinsamen Daten zugegriffen wird. Das Problem ist nun aber, dass immer nur 1 Prozess in der kritischen Region sein darf. Dabei kann ein Prozess aber in der kritischen Region unterbrochen werden (Interrupt, Context-Switch, ...). Dann müssen die anderen Prozesse darauf warten, bis der ursprüngliche Prozess die Exekution fortsetzt und die kritische Region verlässt.



Die Realisierung von kritischen Regionen muss folgende Eigenschaften erfüllen:

- ∅ **Wechselseitiger Ausschluss:** Nur ein Prozess kann sich zu jedem Zeitpunkt in seiner kritischen Region befinden.
- ∅ **Fortschreiten des Programms:** Kein Prozess außerhalb der kritischen Region darf einen anderen Prozess beim Eintreten in seine kritische Region behindern.
- ∅ **Begrenzte Wartezeit:** Prozesse können sich nicht unbegrenzt lange in ihrer kritischen Region befinden, während andere Prozesse darauf warten, in die kritische Region einzutreten.

Die Software muss zur Realisierung der kritischen Region keine Annahmen für korrektes Funktionieren treffen, aber auch eine Lösung für alle möglichen Szenarien bereitstellen.

Die Hardware muss spezielle atomare Maschineninstruktionen bereitstellen.

Atomare Instruktionen können nur als Ganzes erfolgreich sein oder scheitern.

Das Betriebssystem stellt spezielle Funktionen und Datenstrukturen zur Verfügung, wie Semaphoren, Monitore, etc.

Softwarelösungen

Die Synchronisierung erfolgt durch gemeinsame Variablen (auch Semaphoren genannt). Jeder (nebenläufige) Prozess betreibt dann **Busy Waiting**:

- ∅ Warten auf das Eintreten einer Bedingung durch permanentes Testen
- ∅ benötigt aber Prozessorzeit, was zu Problemen führt, wenn mehrere Prozesse den Prozessor verwenden wollen

Folgende Softwarelösung kann zum Synchronisieren von 2 Prozessen verwendet werden.

Dekker Algorithmus 1

Wir verwenden eine gemeinsame Variable *turn* (bool). Ein Prozess i darf die kritische Region betreten, wenn *turn* = i. Dadurch entsteht eine wechselnde Nutzung des kritischen Abschnittes (dominiert durch langsameren Prozess). Allerdings blockiert der zweite Prozess dauerhaft, wenn Prozess 1 ausfällt.

Prozess 0:

```
repeat
    while turn != 0;
        // kritische Region
        turn = 1;
        // nicht-kritische Region
    until false
```

Prozess 1:

```
repeat
    while turn != 1;
        // kritische Region
        turn = 0;
        // nicht-kritische Region
    until false
```

Wechselseitiger Ausschluss ist gegeben. Dennoch funktioniert dieser Algorithmus nicht.

Weil im Dekker Algorithmus 1 der „Name“ des Prozesses gespeichert wird, um ihm einen Eintritt zu gewähren, entstehen eben Probleme. Besser wäre es, wenn jeder Prozess einen Schlüssel bekommt. Dazu führen wir einen boolschen Vektor ein: *flag[2]*. Jeder Prozess kann das Flag des anderen überprüfen, aber nicht ändern:

Dekker Algorithmus 2

Wir verwenden einen Prozess-Array, um festzulegen, welcher Prozess die kritische Region betreten möchte. *flag[i]* = true bedeutet, dass Pi bereit ist, in kritische Region einzutreten.

| | |
|---|--|
| Beide Prozesse können gleichzeitig hier sein. | <pre>bool flag[i]=false; repeat flag[i] = true; while (flag[j]); // kritische Region flag[i] = false; // nicht kritische Region until false;</pre> |
|---|--|

Auch hier ist ein wechselseitiger Ausschluss gegeben, aber wieder blockiert der andere Prozess, wenn einer in der kritischen Region ausfällt.

Würden „*flag[i]* = true“ und „*while(flag[j])*“ vertauscht, ist ein wechselseitiger Ausschluss nicht mehr gegeben (zur Erklärung was wechselseitiger Ausschluss bedeutet).

Wieder fehlt ein Algorithmus zur Bestimmung, wer in die kritische Region darf – Lösung: Jeder Prozess setzt Flag, um zu zeigen, dass er in die krit. Region will und die Reihenfolge für das Eintreten in die krit. Region wird festgelegt.

Wir kommen also zum Dekker Algorithmus 3:

Dekker Algorithmus 3

Ist eine Kombination aus Algorithmus 1 und 2. Wir initialisieren turn =1 (oder 0) und flag[0] = flag[1] = false.

```
// process i
bool flag [2];
int turn;
repeat
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    // kritische Region
    flag[i] = false;
    // nicht kritische Region
until false;
```

Diesmal sind Fortschreiten des Programms und begrenzte Wartezeiten gegeben – auch ein Wechselseitiger Ausschluss!

Bakery Algorithmus 1

Wollen mehr als 2 Prozesse in die kritische Region eintreten, müssen wir auf einen anderen Algorithmus zurückgreifen: Den Bakery Algorithmus.

Dabei bekommt jeder Prozess P_i eine Nummer $ticket[i]$ – aktuelle Nummer ist stets größer oder gleich der vorher vergebenen Nummer. Der Prozess mit der kleinsten Nummer $\neq 0$ kommt zuerst in die kritische Region. Allerdings kann nicht ausgeschlossen werden, dass mehrere Prozesse die gleiche Nummer bekommen: z.B.: 1, 2, 3, 3, 3, 4, 4, 5, 6, 6, 6, 7

Wenn nun die Prozesse P_i und P_j die gleiche Nummer bekommen und $i < j$, dann kommt P_i vor P_j in seine kritische Region. Prozesse betreten ihre kritische Region basierend auf dem FCFS-Prinzip.

Es wird folgende Notation (lexikographische Reihenfolge) für Prozess i mit Tupel $(ticket[i], i)$ verwendet:

$(a,b) < (c,d) \iff (a < c) \text{ OR } (a == c \text{ AND } b < d)$

Außerdem werden gemeinsame Variablen „bool choosing[n]“ (=false) und „int ticket[n]“ (= 0) verwendet/initialisiert

```
repeat          // aus der Sicht von Prozess i
    choosing[i]= true; // choosing mit false und ticket mit 0 initialisiert
    ticket[i]= max(ticket[0],...,ticket[N-1])+1;
    choosing[i] = false;
    for j = 0 to N-1
        do begin
            while choosing[j] do no-op;
            while ticket[j] != 0 and (ticket[j],j) < (ticket[i],i) do no-op;
        do end
        for end
    kritische Region
    ticket[i]= 0;
    ...
until false;
```

Ohne choosing sieht das so aus:

```
repeat           // aus der Sicht von Prozess i

    ticket[i] = max(ticket[0],...,ticket[N-1])+1;

    for j = 0 to N-1
        do begin

            while ticket[j] != 0 and (ticket[j],j) < (ticket[i],i) do no-op;
        do end
    for end

    kritische Region
    ticket[i] = 0;
    ...
until false;
```

Hardwarelösungen

Interrupt Disabling

Wird für Einprozessorsysteme verwendet, wobei mehrere Prozesse die CPU im Time-Sharing-Modus verwenden. Ein Prozess behält die CPU dann solange bis ein Systemaufruf getätigkt wird oder ein Interrupt erfolgt. Durch Verhindern von Interrupts können kritische Regionen realisiert werden. Das OS bietet die Möglichkeit, Interrupts ein- bzw. auszuschalten.

```
repeat
    disable interrupts
    critical section
    enable interrupts
    remainder
forever
```

Die Nachteile liegen aber auf der Hand:

Verlust von Performance durch Ausschalten von Time-Sharing-Modus, und funktioniert nicht für alle Multiprozessorsysteme, da sich pro Prozessor ein Prozess im kritischen Bereich befinden könnte.

Test and Set

Ist eine mehrprozessorfähige HW-Lösung für eine kritische Region, wobei ein atomarer Hardwarebefehl existiert, der zwei Aktionen (Lesen und Schreiben/Prüfen) auf einer einzelnen Speicherstelle mit nur 1 Befehlszyklus ausführt. Atomar = Ohne Unterbrechung, bzw. Interrupt.

```
function testset (var i: integer): boolean;
begin
    if i = 0 then
        begin
            i := 1;
            testset := true;
        end;
    else testset := false;
end.
```

Die Realisierung erfolgt durch eine gemeinsame Variable (im gemeinsamen Speicher), die von allen Prozessen verwendet wird. Maximal 1 Prozess kann sich in der kritischen Region befinden und alle anderen Prozesse, die in die kritische Region wollen, befinden sich im Busy-Waiting-Modus.

Nachteil: Prozesse können verhungern, die Auswahl eines bestimmten Prozesses bei Eingang der kritischen Region ist beliebig.

```
var shared: integer;
shared := 0;    // globale Initialisierung

while not testset (shared) do nothing;
    // kritische Region
shared := 0;
    // remainder
```

Die Verwendung von speziellen, atomaren Maschinenbefehlen für die Realisierung von kritischen Regionen hat Vor- und Nachteile

Vorteile

- ∅ Kann für beliebige Anzahl von Prozessen für Ein- und Mehrprozessorsystemen (mit gemeinsamen Speicher) verwendet werden.
- ∅ Einfach Lösung
- ∅ Mehrere kritische Regionen (durch eindeutige Variablen definiert) können unterstützt werden.

Nachteile

- ∅ Busy-Waiting ist Verschwendug des Prozessors
- ∅ Starvation (Verhungern) möglich.
- ∅ Deadlock möglich

Semaphore

Weil Software- und Hardwarelösungen jeweils viele (situationsbedingt gravierende) Probleme mit sich ziehen, hat man sich nach einer Lösung umgesehen, die vom Betriebssystem realisiert wird: **Semaphoren**.

Dabei wird eine Integer Variable s (Wert ist immer ≥ 0) initialisiert und der Zugriff durch zwei atomare Operationen definiert:

```
wait:
    while s <= 0;
        s=s-1;
signal:
    s=s+1;
```

Noch bewirkt die Semaphore-Operation wait Busy-Waiting. Um nun auch noch Busy-Waiting zu vermeiden, wird eine eigene Datenstruktur realisiert:

```
type semaphore = record
    value: integer;
    L: list of processes;
end;
```

Ein Prozess, der auf die Semaphor S wartet, wird in die Liste der Prozesse (blocked queue) von S abgelegt. Die signal Operation entfernt einen Prozess aus dieser Liste und stellt ihn in die Ready-Queue.

```

init (S, val):      S.value := val; S.queue := leere Liste;

wait (S): S.value := S.value - 1;
           if S.value < 0
             then   add this process to S.queue
                     and block it;

signal (S):       S.value := S.value + 1;
                  if S.value <= 0
                    then   remove a process P from S.queue
                            and place it on the ready queue;

```

$S.value \geq 0$ bestimmt die Anzahl der Prozesse, die hintereinander ein wait ausführen können, ohne warten zu müssen.

Der Absolutbetrag eines negativen $S.value$ gibt die Anzahl der Prozesse an, die in der Warteschlange von S blockierend warten.

Mit der neuen Definition von Semaphore-Operationen blockieren wartende Prozesse in blocked Queues anstatt sich im Busy-Waiting-Modus zu befinden. Das heißt, der Status des Prozesses wird auf Waiting gesetzt und die Kontrolle geht an den CPU-Scheduler, der einem anderen Prozess die CPU zuteilen kann.

Beispiele

Kritische Region betreten

```

Repeat
  wait(mutex);
  // kritische Region
  signal(mutex)
  // nicht kritische Region
until false

```

Prozess 2 soll mit Ausführung von S2 erst beginnen, wenn Prozess 1 die Anweisung S1 ausgeführt hat. $synch=0$

Prozess 1

```

S1
signal(synch)

```

Prozess 2

```

wait(synch)
S2

```

Mehr zu Semaphoren (und deren Verwendung) im Kapitel *Prozesse/Interprozesskommunikation/Semaphoren*.

Monitor

Programmiersprachen/OS-Konstrukt zur Koordinierung von Prozessen bestehend aus:

- ∅ lokalen Variablen, die den Zustand des Monitors spezifizieren
- ∅ Prozeduren, die lokale Variablen abfragen oder verändern können
 - Prozesse kommen nur über diese Prozeduren in den Monitor
 - Maximal 1 Prozess im Monitor zu jedem beliebigen Zeitpunkt
- ∅ Initialisierungscode

Ein Monitor bietet äquivalente Funktionalität wie Semaphoren, jedoch mit besserem Überblick.

```
type monitor-name = monitor
    variable declarations
    procedure entry P1(...);
        begin ... end;
    ...
    procedure entry Pn(...);
        begin ... end;
    begin
        initialization code
    end.
```

Monitor mit Bedingungsvariablen

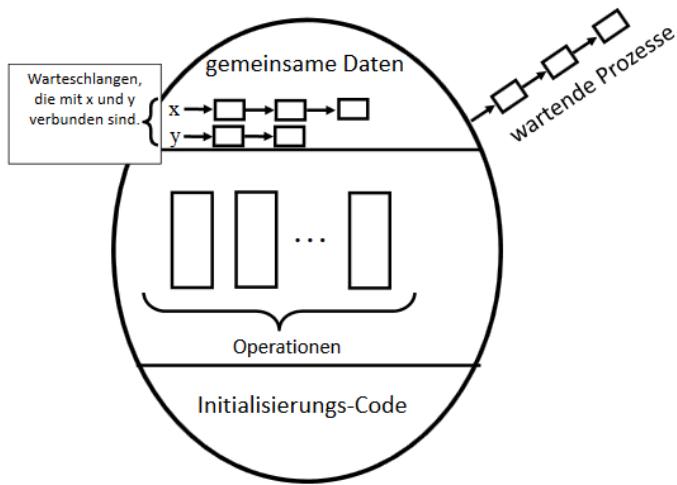
Monitor ermöglicht wechselseitigen Ausschluss (**Mutual Exclusion**). Bisheriges Monitorkonzept kann nicht alle Synchronisationsarten umsetzen. Daher benötigen wir neuen Synchronisationsmechanismus: **Condition Variables**

Die Syntax von Condition Variables:

```
var x,y: condition;
x.wait;           // Prozess ist blockiert, bis ein anderer Prozess
x.signal;         // x.signal aufruft. Condition Variable x wird true.
```

Condition Variables können nur im Zusammenhang mit 2 Operationen verwendet werden: signal und wait.

- ∅ Dabei setzt x.signal einen Prozess frei, der nach Ausführung von x.wait blockiert ist.
- ∅ Falls kein Prozess durch x.wait wartet, dann bewirkt x.signal keinerlei Veränderung (keine speichernde Wirkung im Gegensatz zu Semaphoren)
- ∅ Der Prozess, der x.signal ausführt, wartet bis der freigesetzte Prozess den Monitor wieder verlässt.



Erzeuger/Verbraucherproblem

```
type ErzeugerVerbraucher = monitor
  b: array[0..K-1] of items;
  In := 0, Out := 0, cnt := 0 : integer;
  notfull, notempty: condition

  append (v):
    if (cnt == K) notfull.wait;
    b[In] := v;
    In := (In + 1) mod K;
    cnt := cnt + 1;
    notempty.signal;

  take (v):
    if (cnt == 0) notempty.wait;
    v := b[Out];
    Out := (Out + 1) mod K;
    cnt := cnt - 1;
    notfull.signal;
```

Klassische Synchronisationsprobleme

Leser/Schreiber Problem

Ist ein klassisches Synchronisationsproblem, bei dem parallel exekutierende Prozesse sich gemeinsame Daten teilen (Shared Data). Einige Prozesse lesen Daten, andere lesen oder schreiben die Daten. Dabei kann

- ∅ eine beliebige Zahl von Leser Daten simultan lesen
- ∅ nur ein Schreiber kann Daten zu bestimmten Zeitpunkt schreiben
- ∅ kein Leser kann Daten lesen, wenn Schreiber Daten schreibt

Leser und Schreiber können verschiedene Prioritäten haben. Beispiel:

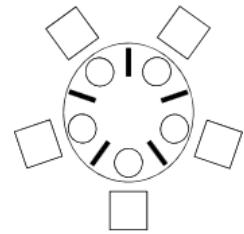
| | | |
|--------------------------------------|-----------------------|--|
| <code>reader()</code> | <code>writer()</code> | readcount: Wieviele Leser gerade lesen |
| { readcount=0; x =1; wsem = 1; | { | |
| repeat | repeat | semaphore x für exklusiven Zugriff auf readcount |
| wait(x); | wait(wsem); | semaphore wsem für exklusiven Schreibzugriff |
| readcount=readcount+1; | WRITEUNIT; | |
| if readcount == 1 then wait(wsem); | signal(wsem); | Leser hat Priorität |
| signal(x); | forever; | Schreiber hat nur Zugang, wenn es kein Leser gibt |
| READUNIT; | }; | Schreiber können eventuell nie auf Daten zugreifen |
| wait(x); | | |
| readcount=readcount-1; | | |
| if readcount == 0 then signal(wsem); | | |
| signal(x); | | |
| forever; } | | |

Dining Philosophers Problem

5 Philosophen denken und essen.

Jeder braucht zum Essen zwei Essstäbchen (ES).

Jeder Philosoph kann nur 1 ES zu einem bestimmten Zeitpunkt aufheben.



Gesucht: Lösung ohne Deadlock (Verklemmung) und Verhungern

Lösungsversuch 1

Jeder Philosoph wird als Prozess realisiert und jedes ES wird durch ein Semaphor kontrolliert.

Prozess Pi:

```

repeat
    wait(chopstick[i]);
    wait(chopstick[i+1 mod 5]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[i+1 mod 5]);
    think
until false
  
```

Dabei kann eine Verklemmung entstehen, wenn jeder Philosoph das ES zu seiner rechten aufhebt/die erste wait Operation von synchron ausgeführt wird (z.B. wait(chopstick[2]) und wait(chopstick[3], und so weiter...)).

Diese Verklemmung könnte vermieden werden, indem ein zusätzliches Semaphor eingeführt wird, um maximal 4 Philosophen gleichzeitig das Aufheben zu erlauben.

Auch könnte man einführen, dass ein Philosoph nur dann ein ES aufheben darf, wenn beide verfügbar sind.

Ein etwas „interessanter“ Ansatz wäre eine asymmetrische Lösung: Philosoph mit gerader ID nimmt zuerst linkes und dann rechtes Essstäbchen auf. Philosoph mit ungerader ID geht in umgekehrter Reihenfolge vor.

Lösungsversuch 1

```

type dining-philosophers = monitor
  var state : array [0..4] of (thinking, hungry, eating);
  var self : array [0..4] of condition;

  procedure pickup (i:0..4);
  begin
    state[i] := hungry;
    test(i);
    if state[i] != eating then self[i].wait;
  end;

  procedure putdown (i:0..4);
  begin
    state[i]:=thinking;
    test(i+4 mod 5);
    test(i+1 mod 5);
  end;

  procedure test (k:0..4);
  begin
    if state[k+4 mod 5] != eating
      and state[k] = hungry
      and state[k+1 mod 5] != eating
    then begin
      state[k] := eating;
      self[k].signal;
    end;
  end;

  begin
    for i:= 0 to 4
      do state[i] := thinking;
  end.
  
```

Hier wird einem Philosophen nur dann erlaubt, ein ES aufzunehmen, wenn sein linkes und rechtes ES gerade nicht benutzt werden. Auch wird eine Condition Variable self verwendet, um das Essen für einen Philosophen zu verzögern, wenn ihm nicht 2 ES zur Verfügung stehen.

Philosoph i muss die Operationen pickup und putdown in der folgenden Reihenfolge durchführen:

```
dp.pickup(i);
    eat
dp.putdown(i);
```

Nun werden auch Deadlocks verhindert.

Deadlocks

Ausschnitt aus testset:

1. P1 kommt in kritische Region über testset Operation
2. P1 wird durch Interrupt in kritischer Region unterbrochen.
3. P2 mit höherer Priorität als P1 bekommt die CPU und möchte in kritische Region.
4. P2 kann aber erst in die kritische Region, wenn P1 diese verlassen hat.
5. P2 geht daher in Busy-Waiting Modus.
6. P1 bekommt die CPU nicht wegen geringerer Priorität als P2
7. P1 und P2 befinden sich in einem Deadlock.

Interprozesskommunikation

Prozesse können zum einen durch Nachrichten über den Kernel (message passing), zum anderen aber über einen gemeinsamen, physischen Speicher (shared memory) interagieren. Außerdem können Prozesse über asynchrone Signale interagieren.

Das müssen sie tun, weil die Prozesse kaum etwas voneinander wissen (Speicherschutz!). Manchmal müssen einfach spezielle Daten gemeinsam verwendet, an einen anderen Prozess weitergeleitet und folglich dann auch irgendwie koordiniert werden.

So könnte man beispielsweise über Dateien im Dateisystem kommunizieren – das wäre aber sehr langsam und umständlich (wer darf wann darauf schreiben/lesen), welche zwar teilweise gelöst werden können – aber es gibt wesentlich elegantere und schnellere IPC (Interprozesskommunikation) Techniken.

| | |
|--|--|
| <ul style="list-style-type: none"> ∅ Pipes (halbduplex) ∅ FIFOs (benannte Pipes) | <ul style="list-style-type: none"> ∅ Message-Queues ∅ Semaphore ∅ Shared-Memory |
| Streams | Sockets |
| <ul style="list-style-type: none"> ∅ Stream Pipes ∅ Benannte Stream Pipes | machma im 4. Semester |

Sehen wir uns zunächst aber Signale an, welche thematisch auch zu den IPCs gehören – aber aufgrund der Einfachheit nicht für „größere“ Daten gedacht sind:

System V IPC vs POSIX IPC

Srry das ich plötzlich Englisch angefangen hab – war Teil einer Aufgabe, die musste Englisch abgegeben werden :P

Sollt etwas hier noch nicht klar sein, die Erklärungen finden sich in den nachfolgenden Kapiteln, habs aber vorgeschoben, weils alle nachfolgenden betrifft.

Both are using the same concepts – semaphores, shared memory and message queues. Yet, they offer a different interface to those tools.

System V IPC has been around for a longer time, which has resulted in POSIX IPC to be spread less widely. You usually have a better chance having System V being implemented, than with POSIX IPC.

On the other hand, POSIX IPC has learned from the issues and strengths of the System V API – so for many people POSIX IPC is easier to use. Furthermore, POSIX IPC was created to standardize the interface – the same way any interface in the POSIX-Standard was meant to be a standardization.

According to “UNIX, Third Edition: The Textbook”, written by Syed Mansoor Sarwar and Robert M. Koretsky, POSIX IPC Interfaces are multithread safe! This is a huge advantage.

Also, POSIX IPC uses Ascii strings for the names, instead of integers like the System V Interface does. This is more usable IMO.

The System V IPC allows semaphores to be automatically released if a process dies (using SEM_UNDO Flag). So, we can ensure that the semaphore will be reset, in case our process terminates before manually releasing it. This Flag does not exist in POSIX IPC.

| System V | POSIX |
|--|---|
| <ul style="list-style-type: none"> ∅ Exists longer, more common ∅ SEM_UNDO Flag ∅ Uses integers as keys ∅ Not Threadsafe | <ul style="list-style-type: none"> ∅ Has learned from System V ∅ POSIX -> is a standard ∅ Uses strings as keys ∅ Multithreadsafe ∅ Easier to use for many |

Signale (Unix)

Allgemein ermöglichen Signale den Austausch von 1-Bit Informationen zwischen Prozessen (und/oder dem Betriebssystem). Sie benachrichtigen also lediglich über das Auftreten eines Ereignisses, wie zum Beispiel

- ∅ Systemänderung: Fenstergröße

- ∅ Zeitgrenze erreicht: Alarmnachrichten
- ∅ Fehlereignisse: Segmentation Fault
- ∅ I/O Ereignisse: Daten stehen zur Verfügung
- ∅ CTRL-C: SIGINT Signal terminiert Prozess
- ∅ CTRL-Z: SIGTSTP Signal suspendiert Prozess temporär

Signale ähneln SW-Interrupts, bei denen ein Prozess angehalten und eine spezielle Behandlungsfunktion aufgerufen wird.

Abhängig vom Zustand des Prozesses und dem Signal führt der Prozess spezielle Signal Handler aus. Es stehen vordefinierte Signale zur Verfügung.

Synchrone Signale

Sie entstehen bei der Abarbeitung des Instruction Streams eines Prozesses, wie etwa SIGILL (Illegal Instruction) oder SIGSEGV (Illegal Address). Dabei verursachen sie einen TRAP im OS Kernel Trap Handler.

Traps sind SW-Interrupts

Danach wird das Signal an den Prozess (bzw. Thread) weitergeleitet, der den Trap verursacht hat.

Asynchrone Signale

Quelle kommt von außerhalb des gerade exekutierenden Prozesses, wie etwa SGPROF (Profiling Clock), oder SIGINT (Terminal Interrupt).

Beispiel-Signale

| Signal Name | Number | Description |
|----------------|-----------|---|
| SIGHUP | 1 | Hangup (POSIX) |
| SIGINT | 2 | Terminal interrupt (ANSI) |
| SIGQUIT | 3 | Terminal quit (POSIX) |
| SIGILL | 4 | Illegal instruction (ANSI) |
| SIGTRAP | 5 | Trace trap (POSIX) |
| SIGFPE | 8 | Floating point exception (ANSI) |
| SIGKILL | 9 | Kill(can't be caught or ignored) (POSIX) |
| SIGSEGV | 11 | Invalid memory segment access (ANSI) |
| SIGTERM | 15 | Termination (ANSI) |
| SIGSTKFLT | 16 | Stack fault |
| SIGSTOP | 19 | Stop executing(can't be caught or ignored) (POSIX) |
| SIGPROF | 27 | Profiling alarm clock (4.2 BSD) |
| SIGWINCH | 28 | Window size change (4.3 BSD, Sun) |
| SIGPWR | 30 | Power failure restart (System V) |
| ... | ... | ... |

Pipes

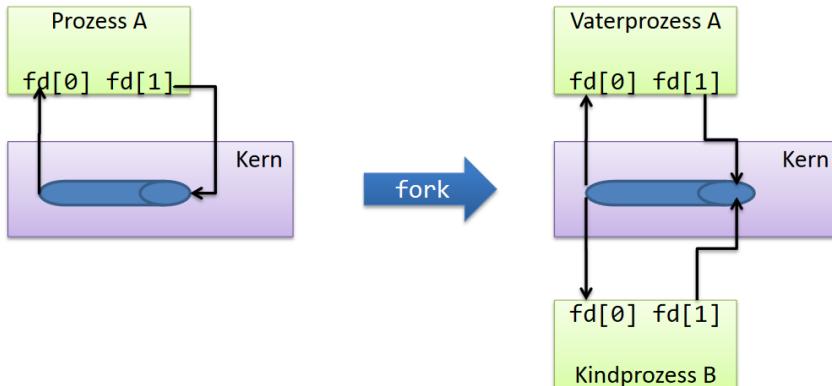
Sie waren die erste Form der IPC und werden heute von allen UNIX-Systemen angeboten. Dabei gilt es aber zu beachten, dass Pipes nur zwischen Prozessen eingerichtet werden können, die einen gemeinsamen Vorfahren haben. Weiters sind Pipes halbduplex.

Halbduplex = Daten können immer nur in eine Richtung fließen

Zum Einrichten einer Pipe können wir den Befehl pipe verwenden (uhuuh). Er richtet eine Pipe im Kern ein, die für den Prozess selbst zunächst absolut nutzlos ist. Danach wird mit fork ein Child-Prozess für die Kommunikation erzeugt.

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

nach **pipe**-Aufruf



Um eine Pipe zu einem anderen Programm einrichten und schließen zu können, müssen wir folgende Schritte durchführen:

1. Pipe einrichten
2. Child-Prozesses erzeugen
3. Standardeingabe des Child-Prozesses auf die Leseseite der Pipe einrichten

```
a. close(fd[1]);
b. dup2(fd[0], stdin);
```

4. Mithilfe von exec den Kindprozess mit einem entsprechenden Programm überlagern

Genau diese Schritte erledigt `popen` für uns – als Rückgabe bekommen wir einen Dateizeiger

```
fp = popen(cmdstring, "r");
```



```
fp = popen(cmdstring, "w");
```



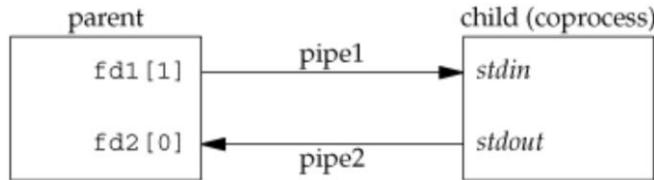
Um diese Pipe (bzw. den FilePointer) zu schließen, verwenden wir `pclose`

```
pclose(fp);
```

Coprozesse

Null Plan was der Professor uns damit sagen wollte:

Denke einfach, wenn Prozesse nebeneinander laufen und miteinander kommunizieren wollen, brauch ma halt zwei pipes...



Benannte Pipes (FIFOS)

Während Pipes nur zwischen verwandten Prozessen verwendet werden können, können FIFOs zwischen beliebigen Prozessen verwendet werden. Dabei ist FIFO einfach nur eine Pipe, die einen Namen hat und wie eine Datei behandelt wird. Daten können aber nur einmal gelesen werden und sind danach **nicht** mehr vorhanden! Sie fließen also wortwörtlich durch die Pipe wie Wasser.

Kommandozeile

Weil FIFOs wie Dateien behandelt werden, können wir sie auch in der Shell einfach verwenden:

In der ersten Shell schreiben wir

```
mkfifo /tmp/test
cat </tmp/test
```

Damit werden die Daten aus der FIFO „/tmp/test“ gelesen. Vergiss nicht – es handelt sich um eine Pipe, bei der die Daten wie Wasser fließen – wir haben sie nun geöffnet, also alles was jetzt in „/tmp/test“ geschrieben wird, fließt direkt in die erste Shell.

Das probieren wir gleich in der zweiten Shell aus:

```
cat >/tmp/test
```

die Shell wartet nun auf Tastatureingaben, die sie sofort (nach einem Return) in die FIFO schreibt.

Der Erzeuger der FIFO darf dabei Benutzungsrechte setzen. **p** zeigt an, dass es sich um eine FIFO handelt (z.B.: “prwxrw-r--“)

FIFOs in C

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

pathname ist der FIFO-Dateiname (wer hätte gedacht)

mode legt die Zugriffsrechte für die FIFO fest (symbolische Konstante, z.B. 0_NONBLOCK)

Tritt ein Fehler auf, gibt mkfifo den Wert 1 zurück, sonst immer 0. Die FIFO kann wie eine gewöhnliche Datei geöffnet und benutzt werden, also mit fopen, open, write, etc..

Regeln für FIFOs

open auf eine FIFO:

- ∅ Ohne **O_NONBLOCK**
 - Leser/Schreiber blockiert bis ein anderer Prozess die FIFO zum Schreiben/Lesen öffnet
- ∅ Mit **O_NONBLOCK**
 - Open für „nur lesen“ kehrt sofort zurück
 - Open für „nur schreiben“ führt zu einem Fehler, wenn kein Leser vorhanden

Schreiben in eine FIFO ohne einen Lese-Prozess generiert ein SIGPIPE-Signal.

Geschlossen wird die Pipe durch den letzten Schreiber, für den Leseprozess wird ein EOF generiert.

Problematisch wird es beim gleichzeitigen Schreiben durch mehrere Prozesse:
Wenn write nicht mehr als **PIPE_BUF** Bytes schreibt, ist alles in Ordnung – sonst werden die Daten zufällig vermischt!

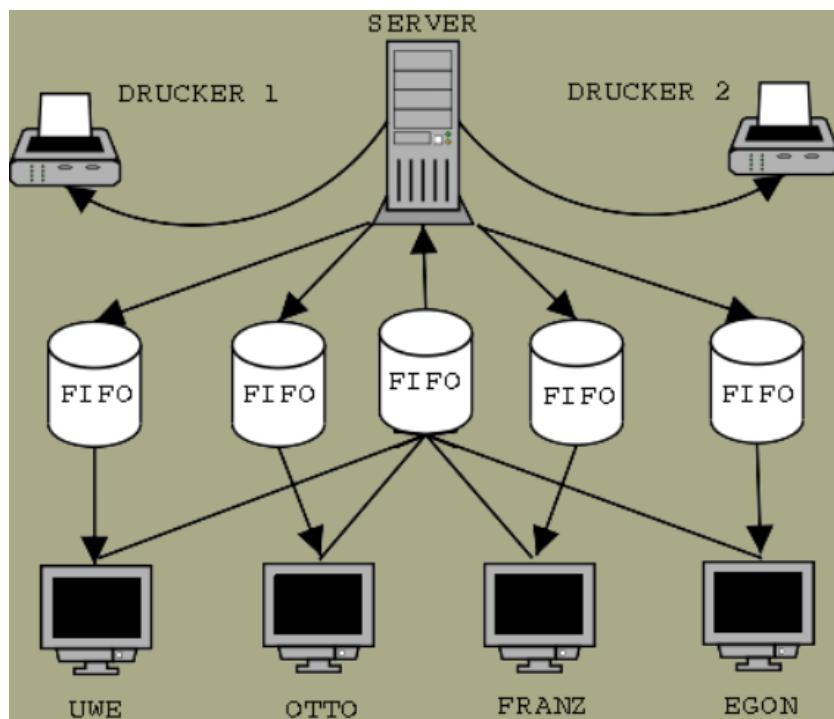
Beispiele

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    FILE *fp_fifo;
    int count;
    char buf[4096];
    if ( mkfifo("FIFO_test", 0777) == -1) {
        perror("mkfifo");
        exit(EXIT_FAILURE);
    }
    if ( (fp_fifo = fopen("FIFO_test", "r")) == NULL){
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    printf("You told my child: ");
    while( (count = fscanf(fp_fifo, "%s", buf)) > 0) {
        printf("%s", buf);
    }
    printf("\n");
    if ( fclose(fp_fifo) == EOF ) { perror("fclose");
        exit(EXIT_FAILURE); }
    if ( unlink("FIFO_test") == -1) { perror("unlink");
        exit(EXIT_FAILURE); }
    exit(EXIT_SUCCESS);
}
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    FILE *fp_fifo;
    char buf[4096];
    if ( (fp_fifo = fopen("FIFO_test", "w")) == NULL ) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    printf("I am child %d, tell me something: ", getpid());
    scanf("%s", buf);
    fprintf(fp_fifo, "%s", buf);
    if ( fclose(fp_fifo) == EOF ) {
        perror("fclose");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```



XSI IPC

X/Open System Interface

Grundsätzlich sind drei Methoden definiert:

- ∅ Zum Austausch von Nachrichten zwischen Prozessen (Message-Queues)
- ∅ Zur Synchronisation über Semaphoren, welche Semaphoren verwendet.
- ∅ Zum Austausch von Daten über gemeinsame Speicherbereiche (Shared-Memory)

Dabei werden unterschiedliche **IPC-Objekte** verwendet:

- ∅ Nachrichtenwarteschlangen
- ∅ Semaphoren
- ∅ Gemeinsamer Hauptspeicherbereiche

Diese Objekte werden einheitlich verwaltet, dh.: alle drei Methoden verwenden dieselben systemweiten Ressourcen, welche von mehreren Prozessen gemeinsam genutzt werden können. Eine Kommunikation zwischen nicht verwandten Prozessen ist möglich und es werden keine File-Deskriptoren verwendet.

Einrichten neuer IPC-Objekte

Jedem Objekt wird beim Einrichten intern vom Kern eine eindeutige **Kennung** (nichtnegative Zahl) zugeteilt. Diese Kennzahlen können sehr groß werden, weil Kennungszahlen von gelöschten Objekten nicht unmittelbar freigegeben werden und grundsätzlich immer bis zu einem maximalen Wert hochgezählt wird und dann wieder bei 0 begonnen wird.

Neben der **Kennung (Identifier)** wird auch ein Schlüssel vergeben, der den Typ key_t (meist in <sys/types.h> definiert) hat und mit der Kennung verbunden wird. Dadurch können nicht verwandte Prozesse ein Objekt gemeinsam nutzen, weil die Prozesse nur den Schlüssel kennen – der sollte halt dann eindeutig sein.

Zum Einrichten haben wir drei Funktionen:

```
msgget  
semget  
shmget
```

Alle Funktionen geben die Objektkennung (oder -1 bei Fehler) zurück und haben neben dem Schlüssel noch ein **flag**. Ein Objekt kann auf zwei verschiedene Arten kreiert werden:

- ∅ **IPC_PRIVATE**
 - Damit wird ein privates Objekt eingerichtet (also kein Schlüssel zugeordnet -> damit ist es anderen Prozessen nicht bekannt)
- ∅ **IPC_CREAT**
 - Es muss ein noch nicht existierender Schlüssel angegeben werden
 - Um sicherzustellen, dass neues Objekt erzeugt wird, muss auch IPC_EXCL gesetzt sein

Verbindung herstellen

Client-Prozesse können zu bereits existierenden Objekten eine Verbindung herstellen – müssen dazu einfach eine entsprechende Funktion (msgget, semget, shmget) aufrufen und den gleichen Schlüssel angeben, der bei der Erzeugung verwendet wurde. Das Flag IPC_CREAT darf **nicht** gesetzt sein – sonst würd ma ja versuchen ein neues Objekt zu erzeugen...

Existierende IPC-Objekte können sogar auf der Kommandozeile angezeigt werden – dazu verwenden wir einfach **ipcs**, welche alle aktuellen Objekte auflistet

```
ipcs
```

Löschen von Objekten

Ein Objekt existiert grundsätzlich solange, bis es explizit gelöscht wird oder aber durch einen Shutdown des Systems. Dabei kann jeder Benutzer/jedes Programm mit den entsprechenden Zugriffsrechten (Eigentümer immer) Objekte löschen. Dazu verwendet man

```
msgctl
```

`semctl``shmctl`

Das Argument/Flag **IPC_RMID** muss angegeben werden.

Auf der Kommandozeile steht uns **ipcrm** zur Verfügung, welches Objekte über den Schlüssel löschen kann.

`ipcrm`

Zugriffsrechte und Limits

Zu jedem Objekt existiert eine Struktur vom Typ ipc_perm, welche die Zugriffsrechte für das Objekt und dessen Eigentumsverhältnisse festlegt. Einige Komponenten dieser Struktur können mit den Befehlen msgctl, semctl und shmctl gesetzt werden (**IPC_SET**). Grundsätzlich ähneln die Zugriffsrechte denen von Dateien – es gibt aber keine Execute-Rechte.

Es gibt gewisse Limits, die für alle drei Objektarten individuell festgelegt werden und nur durch eine Neukonfiguration des Kernels geändert werden können.

Kommunikationsmöglichkeiten

Mit **IPC_PRIVATE**

Vaterprozess erzeugt das Objekt mit **IPC_PRIVATE** und speichert die Kennung in einer Variable. Der Child-Prozess bekommt diese Kennung bei fork durch Vererbung (Kopie).

Sind die Prozesse allerdings nicht verwandt, muss Prozess A das Objekt erzeugen (auch mit **IPC_PRIVATE**) und die Kennung in einer Datei speichern. Prozesse B und C können nun die Kennung aus dieser Datei lesen und dadurch dann auf das Objekt zugreifen.

Ohne **IPC_PRIVATE**

Mehrere Prozesse können einen gemeinsamen Schlüssel per Definition vereinbaren (z.B.: Headerdatei). Ein Prozess (Server) erzeugt dann das Objekt und die anderen Prozesse (Clients) greifen über den vereinbarten Schlüssel darauf zu.

Dann muss aber der Server auch eine Fehlerbehandlung durchführen, für den Fall, dass der vereinbarte Schlüssel bereits existiert.

Nachrichtenwarteschlangen

Standardmäßig werden Message-Queues in Form einer verketteten Liste (nach dem FIFO-Prinzip) verwaltet, wobei Prioritäten folglich möglich sind.

`msgget``msgsnd``msgrcv`

Die Messages oben sollten eigentlich eh selbsterklärend sein, es sei aber angemerkt, dass msgrcv die Nachrichten nicht in der Reihenfolge erhalten muss, in der sie in die Queue eingetragen wurden.

Jede Message besteht aus einem Message-Typen (meist Long, benutzerdefiniert), der Länge des Message (size_t) und der eigentlichen Nachricht (string).

Zu jeder Message-Queue existiert auch eine **msqid_ds**-Struktur, die den aktuellen Status der Queue festlegt.

| Limit | Beschreibung |
|---------------|--|
| MSGMAX | Maximale Anzahl der Bytes einer Nachricht |
| MSGMNB | Maximale Anzahl von Bytes in einer Message-Queue |
| MSGMNI | Maximale Anzahl von Message-Queues im System |
| MSGTQL | Maximale Anzahl von Messages im System |

Message-Get

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

Key stellt den eindeutigen Schlüssel (Name) der Message-Queue dar. Wird der Wert **IPC_PRIVATE** übergeben, wird eine neue Message-Queue erzeugt und das Flag ignoriert. Dabei wird die Kennung der Message-Queue zurückgegeben.

Alternativ lässt sich eine Message-Queue über die msgflag erzeugen. Dazu wird die Flag **IPC_CREAT** übergeben (und übergebener key darf noch nicht existieren). Zudem kann auch das Flag **IPC_EXCL** gesetzt werden, welches in Kombination mit **IPC_CREAT** den Aufruf abbricht, sofern die Message-Queue mit dem übergebenen key schon existiert.

Es wird -1 zurückgegeben, wenn die Erzeugung fehlschlägt.

Message-Send

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

msqid ist die Kennung der Message-Queue, welche wir oben erzeugt haben.

msgp ist ein Zeiger auf den Nachrichtenpuffer. Dieser Puffer muss vom struct msgbuf sein:

```
struct msgbuf{
    long mtype; //Typ
    char mtext[MSGSZ]; //Daten
};
```

msgsz gibt die Größe von mtext in Bytes an.

msgflg:

- 0: blockiert, wenn die Warteschlange voll ist
- **IPC_NOWAIT**: sofort zurückkehre, wenn die Queue voll ist und **errno** setzen.

Rückgabewert ist 0 bei Erfolg, sonst -1 und errno wird gesetzt.

Message-Send

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

Im Gegensatz zu msgsnd gibt msgsz hier die maximale Länge von mtext in Bytes an. msqid und msgp sind ident zu msgsnd.

msgtype

- 0: erste Nachricht aus der Warteschlange (FIFO)
- >0: erste Nachricht mit Typ msgtype
 - mit MSG_EXCEPT kann man die Abfrage umkehren, d.h. es wird die erste Nachricht empfangen, die nicht von diesem Typ ist.
- <0: erste Nachricht (mit kleinstem Typ), welche kleiner als der Absolutwert von msgtyp ist

msgflg kann gesetzt werden, wobei 0 blockiert wenn die queue leer ist und IPC_NOWAIT sofort zurückkehrt (und errno setzt).

Rückgabe ist wieder 0 bei Erfolg und -1 bei Fehler (errno wird gesetzt)

Message-Control

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

cmd legt eine auszuführende Aktion fest:

- ∅ **IPC_STAT**: Abfrage des Status der Message-Queue
- ∅ **IPC_SET**: ermöglicht das Setzen des Eigentümers, der Zugriffsrechte und der maximalen Größe.
 - Eigentümer: msg_perm.uid, msg_perm.gid
 - Zugriffsrechte: msg_perm.mode
 - maximale Größe: msg_qbytes
- ∅ **IPC_RMID**: Entfernt die Message-Queue
 - Prozesse, welche danach noch auf die Queue zugreifen, bekommen einen Fehler (errno = **EIDRM**)

buf dient zur Übergabe von Statusinformationen für IPC_SET oder IPC_STAT Operationen. Das struct msqid_ds ist definiert in <sys/msg.h>

Siehe Folie 7 – Interprozesskommunikation für ein Beispiel.

Shared Memory (Gemeinsamer Speicher)

Der Vorteil von shared memory sollte auf der Hand liegen: Prozesse teilen sich den Speicher, daher müssen sie nicht „kommunizieren“, um die Daten zu erhalten – sie können direkt darauf zugreifen. Daher ist es der schnellste IPC-Mechanismus.

Gleichzeitig ist es aber auch der größte Nachteil von Shared Memory: Der Zugriff muss synchronisiert werden. Sonst weiß man ja nicht ob die Daten aktuell sind, ob nicht gerade der andere Prozess auf die Daten zugreift etc. Diese Problematik lösen wir durch Semaphoren.

```
shmget //Shared-Memory-Segment anfordern
shmat //Shared-Memory-Segment in den Adressraum eines Prozessors
       einbinden
memcpy //schreibt in ein Shared-Memory-Segment
shmdt //Abkoppeln eines angebundenen Shared-Memory-Segments
shmctl //Optionen setzen, Informationen abfragen
```

Zu jedem Shared-Memory-Segment existiert eine shmid_ds-Datenstruktur im Kern. Keinen Plan was folgende Grafik zu bedeuten hat, aber wahrscheinlich sind die Werte in der Datenstruktur enthalten:

| Limit | Beschreibung | Typischer Wert |
|--------|--|----------------|
| SHMMAX | Maximale Größe (in Bytes) eines Shared-Memory-Segments | 0x2000000 |
| SHMMIN | Minimale Größe (in Bytes) eines Shared-Memory-Segments | 1 |
| SHMMNI | Minimale Anzahl von Shared-Memory-Segmenten | 4096 |
| SHMSEG | Maximale Anzahl von Shared-Memory-Segmenten | 4096 |

System V-Interface

SharedMemory anfordern

```
int shmget(key_t key, int size, int shmflag);
```

key und shmflag sind wie bisher (Message-Queues) besprochen zu verwenden.

size gibt die **minimale** Größe an, wobei size=0 ein bereits existierendes Segment öffnet.

Rückgabewert ist die ID des Segments, sonst -1.

Beispiel

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

if ((shmid = shmget( 5L, 200, IPC_CREAT | 0666)) < 0) {
    perror("shmget failed");
    exit(EXIT_FAILURE);
}
```

ShM in Adressraum einbinden

```
void *shmat(int shmid, const void *shmaddr, int shmflag);
```

SharedMemory-Attach

shmaddr gibt die Adresse an, an die das Segment angebunden wird. Hierbei wird empfohlen, NULL zu übergeben, dann wird das Segment nämlich an die erste verfügbare Adresse angebunden.

shmflag kann den Wert SHM_RDONLY enthalten, dann wird nur zum Lesen angebunden. Standardmäßig wird zum Lesen und Schreiben angebunden.

Rückgabewert ist ein Zeiger auf das Segment, -1 beim Scheitern.

Beispiel

```
#include <sys/types.h>
#include <sys/shm.h>
. . .
if (( shm_p = shmat( shmid, (char *) 0, 0 )) < (char *) 0 ) {
    perror("shmat failed");
    exit(EXIT_FAILURE);
}
```

ShM schreiben

```
void *memcpy(void *dest, void *src, size_t n);
```

Ähnlich strcpy kopiert memcpy von der Quelle (src) in das Ziel (dest). Zurückgegeben wird dann ein Zeiger auf den Anfang von dest.

```
memcpy( shm_p, p_str, strlen(p_str) );
```

Da ein Shared-Memory-Segment aber im Grunde nur ein Void-Pointer ist, können wir auch direkt reinschreiben – gegeben dem Wissen über die Größe:

```
(* (int*)shmPointer) = 200; //wenn shmPointer ein integer-feld hält
```

ShM vom Adressraum abkoppeln

SharedMemory-Detach

```
int shmdt(const void *shmaddr);
```

shmaddr ist die Adresse an die das Segment angebunden ist

Der Rückgabewert gibt an, ob die Operation erfolgreich war.

Beispiele

```
#include <sys/shm.h>
. . .
if (shmdt( shm_p ) < 0 ) {
    perror("shmdt failed");
    exit(EXIT_FAILURE);
}
```

ShM-Control

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

cmd gibt eine auszuführende Aktion an

- ∅ **IPC_STAT**: Status abfragen
- ∅ **IPC_SET**: Eigentümer/Zugriffsrechte setzen (über buf)
- ∅ **IPC_RMID**: Löschen des Segments

buf ist eine Struktur, welche die entsprechenden Werte angibt.

Rückgabewert zeigt an, ob die Operation erfolgreich war.

Beispiele

```
#include <sys/ipc.h>
#include <sys/shm.h>
. . .
if( shmctl( shmid, IPC_RMID, 0 ) < 0 ) {
    perror("shmctl failed");
    exit(EXIT_FAILURE);
}
```

Beispiel

```
#define SHMSZ 27

char c, *shm, *s;
int shmid;
key_t key;

key = 5678;

if ((shmid = shmget(key, SHMSZ,
                     IPC_CREAT | 0666)) < 0) {
    perror("shmget"); exit(EXIT_FAILURE);
}

if ((shm = shmat(shmid, NULL, 0)) ==
    (char *) -1) {
    perror("shmat"); exit(EXIT_FAILURE);
}

s = shm;
for (c = 'a'; c <= 'z'; c++) *s++ = c;
while (*shm != '*') sleep(1);
```

```
#define SHMSZ 27

char *shm, *s;
int shmid;
key_t key;

key = 5678;

if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget"); exit(EXIT_FAILURE);
}

if ((shm = shmat(shmid, NULL, 0)) ==
    (char *) -1) {
    perror("shmat"); exit(EXIT_FAILURE);
}

for (s = shm; *s != NULL; s++) putchar(*s);
putchar('\n');

*shm = '*';
```

SharedMemory & fork/exec/exit

Durch einen fork erbt der Kindprozess **alle** angebundenen Shared-Memory-Segmente. Diese werden bei exec allerdings wieder getrennt (nicht zerstört!) – genauso bei exit (auch hier werden sie nicht zerstört!).

Ein Shared-Memory-Segment wird erst dann gelöscht, wenn der letzte Prozess, der es benutzt, terminiert oder die Anbindung aufhebt.

Nach Aufruf von shmdt führt ein Zugriff in dem Prozess zu einem Fehler. Logisch.

POSIX-Interface

Not much to say here, it works the same way System V Shared Memory does, but the Methods differ:

Create a Shared-Memory-Segment

```
#include <sys/mman.h>
#include <sys/stat.h> /* For mode constants */
#include <fcntl.h> /* For O_* constants */
int shm_open(const char *name, int oflag, mode_t mode);
```

Creates and opens a new POSIX-Shared Memory Object. Those Memory Objects are in fact a handle which can be used by unrelated processes to mmap the same region of shared memory.

oflag ist a bit mask and defines, how the object will be opened:

- ∅ O_RDONLY: Read-Only
- ∅ O_RDWR: Read-Write access
- ∅ O_CREAT: Create Memory Object if it doesn't exist
- ∅ O_EXCL: Returns an error, if O_CREAT specified and it already exists
- ∅ O_TRUNC: Truncates the memory object to zero bytes if it exists

Close a Shared-Memory-Segment

```
int shm_unlink(const char *name);
```

Is the converse operation to `shm_open`, which will remove an POSIX-ShM-Object.

Semaphore

Semaphoren kannst dir vorstellen als „Flags“, die angeben, ob die (gemeinsame) Ressource verwendet werden darf. Semaphoren stellen also eine klassische Methode, um den Zugriff durch mehrere Prozesse zu beschränken (man spricht dabei von **synchronisieren**).

Eine Semaphore ist eine geschützte Variable, die von bestimmten Prozessen gelesen/geschrieben werden kann.

Will nun ein Prozess auf einen gemeinsamen Datenbereich zugreifen, überprüft er zunächst den aktuellen Wert der Semaphore.

- ∅ Ist der **Wert > 0**, dann darf der Prozess die Ressource verwenden. Dabei muss der Prozess die Semaphore vor dem Zugriff dekrementieren.
- ∅ Ist der **Wert = 0**, dann wird die Ressource benutzt/ist ausgelastet. Der Prozess muss warten, bis die Semaphore positiv wird.

Danach (also sobald der Zugriff beendet wurde) muss der Prozess die Semaphore wieder inkrementieren, um den anderen Prozessen zu sagen: Die Ressource wird von mir nicht mehr verwendet.

Wertüberprüfung und De-/Inkrementieren müssen atomare Operationen sein und daher im Kern implementiert sein!

Atomare Operationen: Verbund von Einzeloperationen zu einer logischen Einheit, die nur als Ganzes erfolgreich oder fehlschlagen kann.

Synchronisierung mit mehreren Semaphoren

P(s) dekrementiert, V(s) inkrementiert die Semaphore s

| Prozess 1 (Speicher schreiben) | Prozess 2 (Speicher lesen) |
|---|--|
| <pre>Anweisung; ... P(write); /* kritische Region */ /* Schreiben in den Speicherbereich */ V(read); ... end;</pre> | <pre>Anweisung; ... P(read); /* kritische Region */ /* Auslesen des Speicherbereichs */ V(write); ... end;</pre> |

- **Vorsicht bei Initialisierung**
 - write = 1 und read = 0: strikte Ordnung
 - write = read = 0: Deadlock
- **Hinweis**
 - Binäre Semaphore nehmen nur die Werte 0 oder 1 an
 - Allgemeine Semaphore können Werte von 0 bis N annehmen, wobei der aktuelle Wert angeibt, wie viele Prozesse den kritischen Bereich noch benutzen dürfen

Angemerkt sei hier, dass Prozess 1 erst wieder auf den Speicherbereich schreiben darf, wenn Prozess 2 gelesen hat.

Arten von Semaphoren

Dies ist nur ein Überblick, da die tatsächlichen Implementierungen vom UNIX-System abhängen.

So gibt es zum Beispiel **namenlose Semaphoren** (analog zu Pipes), welche für die einfache Synchronisierung von Threads oder Parent-/Childprozessen verwendet wird.

```
#include <semaphore.h>
sem_t sem;
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_post(sem_t *sem);                                // V(sem)
int sem_wait(sem_t *sem);                               // P(sem)
int sem_destroy(sem_t *sem);
```

Logischerweise gibt's dann halt ein Gegenstück, die benannten Semaphoren (analog zu FIFOs), welche Pseudodateien mit Zugriffsrechten sind. Damit lassen sich Prozesse synchronisieren, die eben keinen gemeinsamen Speicherbereich haben.

```
sem_t sem_open(const char *name, int oflag, ...);
int sem_post(sem_t *sem);                                // V(sem)
int sem_wait(sem_t *sem);                               // P(sem)
int sem_close(sem_t sem);
```

SVR 4 Semaphore

Ein einzelnes Semaphor stellt eine Menge von einem oder mehreren Semaphorenwerten dar. Die Anzahl der Werte dieser Menge muss beim Erzeugen festgelegt werden und erlaubt eine detaillierte Aufteilung in kritischen Bereichen.

Das Erzeugen von SVR 4 Semaphoren besteht aus 2 Befehlen (Erzeugen und Initialisieren) und muss immer explizit gelöscht werden!

| |
|--|
| <pre>semget //Semaphore anlegen, bzw. Zugriff auf bestehenden besorgen semctl //Statuskontrolle, Initialisierung, Löschen</pre> |
|--|

```
semop, semtimedop //Operationen P(s) und V(s) (In/Dekrementieren)
```

Status und Limits

Zu jedem Semaphor existiert eine semid_ds – Datenstruktur:

| Limit | Beschreibung | Typischer Wert |
|--------|--|----------------|
| SEMVMX | Maximaler Wert eines Semaphors | 32767 |
| SEMMNI | Maximale Anzahl von Semaphormengen | 128 |
| SEMMNS | Maximale Anzahl von Semaphoren | 3000 |
| SEMMSL | Maximale Anzahl von Semaphoren pro Semaphormenge | 250 |
| SEMOPN | Maximale Anzahl von Operationen pro semop -Aufruf | 32 |

System V-Interface

Semaphore anlegen

```
int semget (key_t key, int nsems, int semflag);
```

nsems gibt die Anzahl der Semaphore in der Menge an, wird auf 0 gesetzt, wenn eine existierende Semaphorenmenge geöffnet wird.

Rückgabewert ist die ID des Semaphors oder -1 bei Fehler.

Beispiel

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

if (( semid = semget(5L, 1, IPC_CREAT | 0666) ) < 0 )
{
    perror("semget failed");
    exit(EXIT_FAILURE);
}
```

Semaphore Initialisieren/Löschen

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

semnum ist ein bestimmter Wert aus der Semaphormenge [0, nsems-1]

cmd legt die auszuführende Aktion fest

- ∅ **IPC_STAT**: Abfragen der Struktur semid_ds
- ∅ **IPC_SET**: Eigentümer/Zugriffsrechte setzen
- ∅ **IPC_RMID**: Löschen der gesamten Menge
- ∅ **GETVAL, SETVAL**: Abfragen/Setzen eines Wertes
- ∅ **GETALL, SETALL**: Abfragen/Setzen aller Werte

```
union semun {
    int val;           /* value of SETVAL */
    struct semid_ds *buf; /* buffer for IPC_STAT,IPC_SET */
    ushort *array;     /* array for GETALL, SETALL */
};
```

Rückgabewert musst dir in der ManPage anschauen, weils von der Rückgabe abhängig ist. (man semctl).

Beispiel

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

. . .

/* initialize */
union semun arg;
arg.val = 1;
if( semctl( semid, 0, SETVAL, arg ) < 0 ) {
    perror("semctl SETVAL failed");
    exit(EXIT_FAILURE);
}

. . .

/* remove semaphore 0 */
if( semctl( semid, 0, IPC_RMID, arg ) < 0 ) {
    perror("semctl IPC_RMID failed");
    exit(EXIT_FAILURE);
}
```

Semaphore setzen/freigeben

```
int semop(int semid, struct sembuf *sops, size_t nsops);

struct sembuf {
    ushort sem_num;           /* semaphore no.: [0..nsems-1] */
    short sem_op;             /* operation */
    short sem_flg;            /* flags (IPC_NOWAIT, SEM_UNDO) */
};
```

sops ist die Zeiger zu einem Array an Semaphoroperationen (**struct sembuf**).

sem_op (in der sembuf struct) ist die auszuführende Operation:

- ∅ > 0: Ressource freigeben (Semaphore um eins erhöhen)
- ∅ < 0: Ressource anfordern (Semaphore um eins verringern)
- ∅ = 0: Warten bis Semaphor gleich 0 ist

nsops ist die Anzahl der Operationen (Elemente) im Array *sops*.

die sem_flg können folgende 3 Werte annehmen (wahrscheinlich auch mehr, siehe Manpage, aber die 3 sind die meistgenützten):

IPC_NOWAIT: Abbruch wenn der Semaphorwert

- ✓ **sem_op < 0**: Kleiner als der absolute Wert von **sem_op** ist
- ✓ **sem_op = 0**: nicht gleich 0 ist

0

- ✓ **sem_op < 0**: wartet bis der Semaphorwert größer oder gleich dem absoluten Wert von **sem_op** ist, oder der Semaphor gelöscht wird oder ein Signal eintrifft
- ✓ **sem_op = 0**: wartet bis der Semaphorwert gleich 0 ist, oder der Semaphor gelöscht wird oder ein Signal eintrifft

SEM_UNDO

- ✓ Wenn man bei Prozessen, die sich freiwillig oder unfreiwillig vorzeitig beenden, sicherstellen will, dass die gesetzten Semaphore wieder zurückgesetzt werden
- ✓ Wird **SEM_UNDO** gesetzt, dann merkt sich der Kern in einem sogenannten undo-Zähler, wie viele Ressourcen durch diese spezielle Semaphorvariable belegt werden
- ✓ Beim Beenden des Prozesses setzt der Kern den entsprechenden Semaphor wieder zurück

Rückgabewert gibt an, ob Operation erfolgreich war.

Beispiel

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

...
/* reset a semaphore: V-operation */
struct sembuf ops[1];

...
ops[0].sem_num = 0;
ops[0].sem_flg = 0;
ops[0].sem_op = +1;

...
if( semop( semid, ops, 1 ) < 0 ) {
    perror("semop V() failed");
    exit(EXIT_FAILURE);
}
```

POSIX-Interface

Named Semaphores

Create new semaphore

```
#include <semaphore.h> //for semaphore...
#include <fcntl.h> //for O_*
#include <sys/stat.h> //for mode constants

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
               unsinged int value);
```

This will create a new named semaphore or open an existing named semaphore.

Lock a Semaphore

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem); //non-blocking, errno set to EAGAIN
```

This will lock a semaphore, meaning it decrements the semaphore. If the value is already on 0, it will block (or return error if trywait).

Unlock a Semaphore

```
int sem_post(sem_t *sem);
```

Yeah, this unlocks the semaphore. It will return 0 on success.

Close Semaphore

```
int sem_close(sem_t *sem);
```

This will close the named semaphore. You want to use this, if you your process has finished using the semaphore. The semaphore still remains in the system!

Destroy Semaphore

```
int sem_unlink(sem_t *sem);
```

This will remove the Semaphore completely from the system – but only when the reference count reaches 0. Meaning, all process having the semaphore opened must have either closed the semaphore or been terminated.

Unnamed Semaphores

Create new semaphore

```
#include <semaphore.h> //for semaphore...  
  
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

This will initialize the unnamed semaphore at the address pointed to by sem. sem just has to be a declared variable accessible. So you could do:

```
sem_t *sem; //pass this variable as sem to sem_init
```

The pshared argument indicates whether this semaphore is to be shared between the threads of a process or between processes. If it is 0, then the semaphore is shared between the threads and should be located at some address that is visible to all threads (globl variable, malloc'ed). Otherwise it is shared between processes and should be located in a region of shared memory.

The value is just the initial value of the semaphore.

Lock a Semaphore

Works identically to named semaphores

Unlock a Semaphore

Too.

Destroy the Semaphore

```
int sem_destroy(sem_t *sem);
```

When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using sem_destroy.

Note, that destroying a semaphore that other processes are currently blocked on produces undefined behaviour! But you could theoretically reinitialize the semaphore using sem_init and will have a defined behaviour from this moment on again :^}

Threads

Prozesse sind zwar sehr einfach zu erzeugen und zu verwenden, sowie auch zu verwalten, allerdings sind sie für gewisse Aufgabenstellungen einfach ineffizient. Prozesse verbrauchen viele Ressourcen und benötigen aufwändige Context-Switches. Auch ist die Rückgabe von Daten des Kindprozesses nicht sehr flexibel und benötigt Kodierungsaufwand (IPC). Daher hat man Threads entwickelt.

Der Thread

Es gibt nur einen Prozess, dafür aber mehrere Ausführungsstränge (Threads). Der Prozess entspricht damit einer Einheit für die Ressourcenverwaltung, während der Thread einer Einheit für die Ausführung entspricht.

Eigenschaften

Grundsätzlich müssen wir zwischen **Benutzer-Threads** (*Green Threads*), **Kernel-Threads** (*Native Threads*) und **Hybride Threads** unterscheiden.

Anwendung finden Threads vor allem in GUIs (zur Interaktion und Berechnung im Hintergrund), Server (z.B. Thread pro Aufruf), um Multiprozessormaschinen auszunutzen und für Ereignisbasierte Systeme.

Es gibt **gemeinsame Ressourcen für alle Threads**, wie

- ∅ Adressraum
- ∅ Offene Dateien (Filedeskriptoren)
- ∅ Signal-Handler und -Einstellungen
- ∅ Benutzer- und Gruppenkennung
- ∅ Arbeitsverzeichnis

Aber auch **Thread-eigene Elemente**

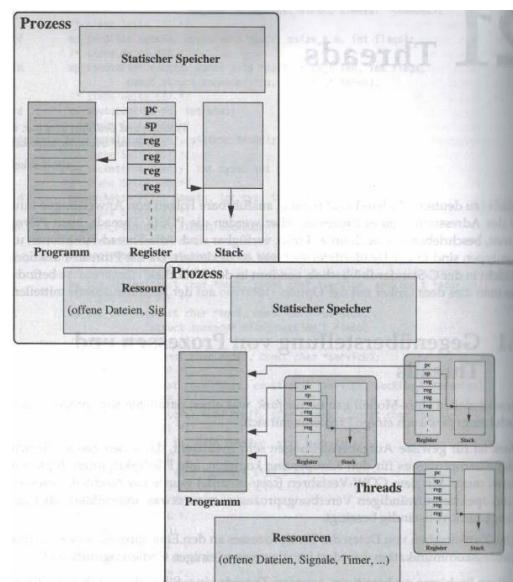
- ∅ Eigene Thread-Kennung (Thread-ID)
- ∅ Eigene Kopie der Prozessorregister (einschließlich PC und SP)
- ∅ Eigener Stack (für Parameter, lokale Variablen und Rückgabeadressen)
- ∅ Eigene Signalmaske
- ∅ Eigene Priorität
- ∅ Eigene errno-Variable

Vorteile / Nachteile

Der größte Vorteil ist natürlich die schnelle Erzeugung, daher auch für fein-körnige Parallelisierung von kleinen Codeteilen geeignet.

Aufgrund der gemeinsamen Ressourcen ist auch ein Context-Switch wesentlich schneller. Zudem resultiert daraus eine schnelle Kommunikation, die allerdings synchronisiert werden muss.

Sollte ein Thread blockieren, können andere Threads trotzdem weiterarbeiten (aber nur Kernel-Threads). Asynchrone Kommunikation kann durch Threads ermöglicht werden.



Nachteile sind allerdings ein fehlender Speicherschutz und die Tatsache, dass Threads eines Prozesses voneinander nicht geschützt sind.

POSIX-Threads

Es wird bei der POSIX-API für Threads zwischen Kernel- & User-Threads unterschieden.

Bibliotheken, welche die POSIX-Threads implementieren, werden oft als – *tam tam tam* – **Pthreads** bezeichnet. Obwohl diese Pthreads überwiegend in UNIX-Systemen verbreitet sind (hence, Posix..) finden mittlerweile auch Windows-Implementierungen ihre Wege.

Wichtige Informationen

Alle Thread-Funktionen und Datentypen sind einheitlich in pthread.h deklariert.

Thread-Funktionen befinden sich nicht in der C-Standardbibliothek, sondern in libpthread.so. Daher muss man beim Linken auch **-lpthread** angeben.

POSIX-Interface

Thread erzeugen

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```

- ∅ **thread** zeigt auf die ID des neuen Threads
- ∅ **attr** spezifiziert bestimmte Attribute
- ∅ **start_routine** ist die Funktion, die der Thread aufruft wenn er gestartet wird
 - nur ein void * Argument **arg** wird übergeben

Erzeugt und startet einen neuen Thread. Dieser beginnt direkt nach erfolgreicher Erzeugung asynchron mit seiner Ausführung.

Rückgabewert ist 0 wenn ok, sonst Fehlercode (nicht errno gesetzt).

Wollen wir einen Thread als eigenständigen Thread erzeugen, können wir das Thread-Attribut **PTHREAD_CREATE_DETACHED** verwenden.

Zugriff auf Thread-ID

```
pthread_t pthread_self(void);  
int pthread_equal(pthread_t t1, pthread_t t2);
```

`pthread_self` returniert die ID des aufrufenden Threads. Hier sind keine Fehler definiert, ähnlich `getpid()` bei Prozessen.

`pthread_equal` gibt einen Wert ungleich 0 zurück, wenn die Threads t1 und t2 gleich sind, sonst 0. Auch hier sind keine Fehler definiert.

Terminieren von Threads

```
void pthread_exit(void *value_ptr);
```

- ∅ **value_ptr** ist der Rückgabewert des Threads
 - die von `pthread_create` aufgerufene Funktion gibt void * zurück

Terminiert die Ausführung des aktuellen Threads, ansonsten terminiert ein Thread immer, wenn er aus seiner Startroutine zurückkehrt.

value_ptr kann von einem Thread, der pthread_join aufruft, verwendet werden.

Abbruchfunktionen von Threads

```
int pthread_cancel(pthread_t id);
```

Schickt eine Abbruchaufforderung an einen Thread. Kann also verwendet werden, um einen anderen Thread zu terminieren.

```
void pthread_testcancel(void);
```

Prüft, ob eine Abbruchaufforderung vorliegt. Wenn ja, dann wird abgebrochen, ansonsten eben nicht. Realisiert eigene Abbruchpunkte (man 7 pthreads, Cancellation Points).

Zur Erklärung: Ein Thread/Prozess(?) kann mit pthread_cancel eine Abbruchaufforderung an einen Thread schicken. Dieser definiert für sich selbst, wie er damit umgeht (siehe Abbruchaufforderung unten). Sollte er sich selbst den Type *PTHREAD_CANCEL_DEFERRED* markieren, bekommt er zwar die Abbruchaufforderung zugestellt (und merkt sich diese), wird aber erst beim Aufruf von pthread_testcancel (den er selbst aufruft) abgebrochen.

Abbruchaufforderungen von Threads

Dies setzt die Abbruchaufforderungen, welche bestimmen, wie mit Abbruchaufforderungen (pthread_cancel) umgegangen wird.

In den folgenden beiden Fällen wird der alte State/Type in den übergebenen Parameter zurückgeliefert.

Cancel-State

```
int pthread_setcancelstate(int state, int *oldstate);
```

Grundsätzlich muss definiert werden, ob ein Thread eine Abbruchbedingung überhaupt annimmt.

- ∅ PTHREAD_CANCEL_ENABLE
 - Thread ist **cancelable**. Dies ist die Default-Einstellung, dabei werden Cancel-Request entsprechend dem Cancel-Type bearbeitet.
- ∅ PTHREAD_CANCEL_DISABLE
 - Thread ist **nicht cancelable**. Ein Cancel-Request wird aber nicht ignoriert, sondern vorgemerkt. Sobald sich der State auf Enabled ändert, wird der Request abgearbeitet.

Cancel-Type

```
int pthread_setcanceltype (int type, int *oldtype);
```

Gibt den Cancel-Type an.

- ∅ PTHREAD_CANCEL_DEFERRED
 - Thread läuft weiter und wird erst später (bei einem Abbruchpunkt, definiert durch pthread_testcancel) abgebrochen
- ∅ PTHREAD_CANCEL_ASYNCHRONOUS
 - Thread beendet sich unmittelbar nach dem Empfang der Abbruchaufforderung

Exit-Handler

Ein Thread kann mehrere Exit-Handler Funktionen installieren oder deinstallieren. Sie werden in umgekehrter Reihenfolge zu ihrer Installation ausgeführt. Sie können z.B. für Aufräumarbeiten verwendet werden.

```
pthread_cleanup_push(void (*routine) (void *), void *arg);
```

Installiert Funktion **routine** mit dem Argument **arg**.

```
pthread_cleanup_pop(int execute);
```

Entfernt den zuletzt installierten Exit-Handler.

Ist **execute == 0**, dann wird der Handler nur entfernt, ansonsten wird er auch noch ausgeführt.

Detached / Joined

Ein Thread ist entweder verknüpfbar (joinable) oder eigenständig (detached).

Wenn ein Thread beendet wird, gibt er seine Ressourcen normalerweise noch nicht frei, dazu muss **detach** aufgerufen werden

```
pthread_detach
```

```
int pthread_detach(pthread_t thread);
```

Thread-ID und Exit-Status wird sofort freigegeben.

Aufruf von **join** ist nicht mehr möglich. Umgekehrt, wird auf einen Thread bereits gewartet (mit join), dann hat dieser Aufruf keine Auswirkung.

```
pthread_join
```

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Mit pthread_join warten wir auf einen Thread. Der Aufruf führt dazu, dass der Aufrufer wartet und den Rückgabewert des beendeten Threads in die Adresse value_ptr (Zeiger-Zeiger!) geschrieben bekommt.

Ist daher vergleichbar mit waitpid bei Prozessen.

Kann nur einmal von höchstens einem Thread für eine bestimmte Thread-ID aufgerufen werden. Nachfolgende Aufrufe resultieren in einem Fehler.

Wird nicht auf einen Thread gewartet, dann bleiben Thread-ID und Exit-Status erhalten, bis pthread_join (oder pthread_detach?) aufgerufen wird. (Vergleiche Zombie-Prozess).

Beispiel

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

struct data {
    int nr;
    int zeit;
};

void *rennen(void *arg) {
    int meter = 0, sek;
    struct data *data_zgr = (struct data *) arg;

    while (meter < 50) {
        sleep(sek = rand()%3+1);
        data_zgr->zeit += sek;
        meter += 5;
        fprintf(stderr, "%*s%3d\n", data_zgr->nr * 15-7, " ", meter);
    }
    fprintf(stderr, "%*s\n", data_zgr->nr * 15, "-----");

    return arg; /* auch möglich: pthread_exit( arg ); */
}

int main(void) {
    int i, j;
    pthread_t id[3];
    struct data *renner_daten[3];
    srand(time(NULL));
    for (i=0; i<3; i++)
        renner_daten[i] = calloc( 1, sizeof(struct data) );
    printf("%15s%15s%15s\n", "Laeufer 1", "Laeufer 2", "Laeufer 3");
    printf("-----\n");
    for (i=0; i<3; i++) {
        renner_daten[i]->nr = i+1;
        pthread_create(&id[i], NULL, &rennen, renner_daten[i]);
    }
    for (i=0; i<3; i++)
        pthread_join(id[i], (void **) &renner_daten[i]);

    for (i=0; i<2; i++)
        for (j=i+1; j<3; j++)
            if (renner_daten[i]->zeit > renner_daten[j]->zeit) {
                struct data h = *renner_daten[i];
                *renner_daten[i] = *renner_daten[j];
                *renner_daten[j] = h;
            }
    fprintf(stderr, "Zieleinlauf:\n");
    for (i=0; i<3; i++)
        fprintf(stderr, "%5d: Laeufer %d (%2d Sek.)\n", i+1, renner_daten[i]->nr, renner_daten[i]->zeit);
    return 0;
}
```

Übersetzung:

Sei der Programmname **thread1.c**.

```
gcc -lpthread -o thread1 thread1.c
```

Nebenläufigkeitsprobleme

Zugriff auf gemeinsame Ressourcen (z.B. Dateien oder globale Variablen) muss synchronisiert werden. Ansonsten kommt es zu **Race-Conditions**. Dafür gibt es Mutexe, für die wiederrum eine eigene Bibliothek existiert.

Wenn mehrere Threads mit der gleichen Funktion aufgerufen werden, dann führen sie den gleichen Code aus.

Sollten verschiedene Threads dieselben Funktionen aufrufen, müssen wir überprüfen, dass diese **thread-safe** sind.

Beispiel

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

#define ANZAHL 100000          /* eventuell höher setzen */
int zaehler;                  /* wird durch die Threads hochgezählt */
void *zaehl_thread(void *);

int main(void) {
    pthread_t id1, id2;
    pthread_create(&id1, NULL, &zaehl_thread, NULL);
    pthread_create(&id2, NULL, &zaehl_thread, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    fprintf(stderr, "zaehler = %d\n", zaehler);
    return 0;
}

void *zaehl_thread(void *unbenutzt) {
    int i, zahl;
    for (i=0; i < ANZAHL; i++) {
        zahl = zaehler;
        fprintf(stderr, "\r %7d", zahl+1);
        zaehler = zahl + 1;
    }
    fprintf(stderr, "\n..... Thread %d fertig (zaehler = %d)\n", (int) pthread_self(), zaehler);
    return NULL;
}
```

Beispiel für Ausgabe:

48756..... Thread 1074792800 fertig (zaehler = 48755)
 100708..... Thread 1075845472 fertig (zaehler = 100708)
 zaehler = 100708

Mutual Exclusion

Mutex erlauben es, Codeteile für andere Threads zu sperren, wenn ein Thread diesen Codeteil gerade ausführt. Dabei wird zwischen statischen und dynamischen Mutexen unterschieden.

Statische Mutexe

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Mutex haben den Datentyp **pthread_mutex_t**. Statische Mutex-Variablen müssen mit der Konstante **PTHREAD_MUTEX_INITIALIZER** initialisiert werden.

pthread_mutex_lock sperrt einen Mutex oder blockiert solange, bis der Mutex freigegeben wird.

pthread_mutex_unlock gibt den Mutex wieder frei.

pthread_mutex_trylock sperrt einen Mutex, wenn er frei ist. Kehrt aber sofort wieder mit einem entsprechenden Fehlercode (EBUSY) zurück, wenn der Mutex schon gesperrt ist.

Beispiel

```
#include <pthread.h>
#include <stdio.h>
#define ANZAHL 100000           /* eventuell höher setzen */
int zaehler;                  /* wird durch die Threads hochgezählt */
pthread_mutex_t zaehler_mutex = PTHREAD_MUTEX_INITIALIZER;
void *zaehl_thread(void *);

int main(void) {
    pthread_t id1, id2;
    pthread_create(&id1, NULL, &zaehl_thread, NULL);
    pthread_create(&id2, NULL, &zaehl_thread, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    fprintf(stderr, "zaehler = %d\n", zaehler);
    return 0;
}
void *zaehl_thread(void *unbenutzt) {
    int i, zahl;
    for (i=0; i<ANZAHL; i++) {
        pthread_mutex_lock(&zaehler_mutex);
        zahl = zaehler;
        fprintf(stderr, "\r %d", zahl+1);
        zaehler = zahl + 1;
        pthread_mutex_unlock(&zaehler_mutex);
    }
    fprintf(stderr, "\n..... Thread %d fertig (zaehler = %d)\n", (int)pthread_self(), zaehler);
    return NULL;
}
```

Beispiel für Ausgabe:

165106..... Thread 1074792800 fertig (zaehler = 165106)
 200000..... Thread 1075845472 fertig (zaehler = 200000)
 zaehler = 200000

Dynamische Mutexe

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Dynamische Mutexe werden zum Beispiel benötigt, wenn sich die Mutex-Variable in einer Struktur befindet, zu der man sich mittels malloc einen Speicherbereich allozieren lassen möchte.

pthread_mutex_init initialisiert den Mutex **mutex** mit den in **mutexattr** festgelegten Attributen. Mit NULL werden die default-Attribute verwendet.

pthread_mutex_destroy löscht einen mit init angelegten Mutex und gibt die belegten Ressourcen wieder frei.

Einsatz von Mutexe

Codeteile, die durch einen Mutex geschützt werden, sollen nur so groß sein, wie notwendig. Mutexe machen das Programm effektiv langsamer (sperren/freigeben kostet, warten auf Freigabe kostet).

Mutexe serialisieren die Programmausführung. Wenn Threads häufig auf Mutexe zugreifen, werden Sie meist auch viel Zeit beim Zugreifen und Freigeben verbringen. Daher sollten nur notwendige Teile geschützt werden, sowie mehrere kleinere Codeteile durch mehrere verschiedene Mutexe schützen.

In der Praxis wird man zuerst größere Codeteile durch Mutexe schützen. Schrittweise wird dann verkleinert, sollte die Performance Probleme bereiten.

Deadlocks und Backoff-Algorithmus

Wenn zwei Threads gegenseitige Abhängigkeiten haben, dann können Deadlocks auftreten. Threads versuchen nacheinander die gleichen Mutexe zu sperren. Die Sperrreihenfolge unterscheidet sich, es gibt aber zyklische Abhängigkeiten.

Um solche Deadlocks zu vermeiden, können wir den sogenannten Backoff-Algorithmus verwenden:

- ∅ Dabei wird mit `pthread_mutex_lock` der erste Mutex gesperrt
- ∅ Alle nachfolgenden Mutexe versucht man mit `pthread_mutex_trylock` zu sperren
- ∅ Ist ein Mutex gesperrt, dann werden rückwärts alle zuvor gesperrten Mutexe wieder freigegeben

Bedingungsvariablen

Mutexe dienen nur zur Synchronisation des Zugriffs auf gemeinsame Ressourcen. Manchmal möchte man beim Warten darüber informiert werden, dass eine bestimmte Bedingung eingetreten ist, durch die der Thread fortfahren kann.

Mit Bedingungsvariablen kann der Thread auf das Eintreten einer Bedingung warten. Diese Variablen haben immer den Typ `pthread_cond_t`. Dennoch gibt es zwei Arten: Statische und Dynamische.

Statische Bedingungsvariable

Müssen mit `PTHREAD_COND_INITIALIZER` initialisiert werden:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Aufrufende Methode muss den Mutex **mutex** gesperrt haben. Dann wird der Mutex freigegeben und gewartet, bis die Bedingungsvariable **cond** erfüllt ist. Beim Zurückkehren wird wieder automatisch die Sperre eingerichtet.

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
                           *mutex, const struct timespec *abstime);
```

Wartet nur eine bestimmte Zeitspanne **abstime**, danach wird wieder die Sperre eingerichtet.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Weckt einen wartenden Thread (für **cond**) auf (nach Prioritäten sortiert)

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Weckt alle Threads auf, die auf **cond** warten

Dynamische Bedingungsvariable

Wie bei Mutexen wird es z.B. in einer Struktur verwendet.

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
```

Initialisierung der Bedingungsvariable **cond**.

attr legt die Attribute fest (NULL für Standard)

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Löscht Bedingungsvariable, gibt belegten Ressourcen wieder frei.

POSIX-Auswirkung auf UNIX

Die Verwendung von Pthreads hat auch Auswirkung auf das traditionelle UNIX-Prozesskonzept.

Fork

Ruft ein Thread fork auf, dann existiert im Kindprozess nur der Thread, der fork aufgerufen hat. Der Thread des Kindprozesses hat denselben Zustand wie der Thread im Elternprozess (z.B. gleichen Mutexe gesperrt).

Damit resultiert eine Deadlock-Problematik bei Mutexen, wenn z.B. ein anderer Thread (im Eltern-Prozess) den Mutex sperrt. Dieser existiert ja schließlich im Kindprozess nichtmehr :)

Die Lösung dazu sind spezielle Funktionen für das Registrieren von Handler-Funktionen. Hier gibt man dann meist die in **prepare** geschlossenen Mutexe wieder frei.

pthread_atfork

```
pthread_atfork(void(*prepare) (void), void(*parent) (void),
                void (*child) (void));
```

prepare wird von Elternprozess vor der Erzeugung des neuen Prozesses aufgerufen. Hier werden meist betreffende Mutexe gesperrt.

parent wird vom Elternprozess aufgerufen, unmittelbar bevor fork zurückkehrt.

child wird vom Kindprozess aufgerufen, unmittelbar bevor fork zurückkehrt.

Sollten mehrere Aufrufe stattfinden, erfolgt prepare im LIFO-Ordnung und parent&child in FIFO-Ordnung.

Exec und Exit

Exec ersetzt den ganzen Prozess, Threads werden automatisch beendet. Es werden **keine** Exit-Handler ausgeführt und Mutexe, sowie Bedingungsvariablen verschwinden (kann mit pshared-Attribut verhindert werden).

Exit beendet alle Threads im entsprechenden Prozess. Sobald der Main-Thread beendet wird, werden auch die anderen Threads beendet.

Thread-Safety & Eintrittsinvariante Funktionen

Eintrittsinvariant, wiedereintrittsfähig oder **reentrant**.

Die meisten Standard-Funktionen sind heutzutage schon Thread-sicher und reentrant gemacht. Bei manchen Funktionen war eine thread-sichere Re-Implementierung bei Beibehaltung der externen Schnittstelle nicht möglich.

Thread-Sichere Varianten der Methoden haben das Suffix „_r“ bekommen, z.B. strtok_r, rand_r, etc. Diese Funktionen sind **Thread-Safe und Reentrant**, da sie keine internen statischen Puffer verwenden. Dafür muss der Aufrufer eine Pufferadresse mitgeben, an die dann die Informationen geschrieben werden.

Signale

Jeder Thread kann eine eigene Signalmaske haben, welche mit `pthread_sigmask` gesetzt wird. Beim Erzeugen erbt ein Thread die Signalmaske des Threads, der ihn kreiert. Vom Main-Thread können alle Threads erben.

Signal-Handler gelten aber prozessweit für alle Threads.

Signale, welche von der Hardware geschickt werden, werden immer dem Thread zugestellt, der sie auslöst (SIGFPE, SIGTRAP, etc.).

Andere Signale werden einem beliebigen Thread geschickt. So können Signale mit `pthread_kill` an Threads geschickt werden. **Wichtig** ist hier, dass Signale, die sich auf das Prozessverhalten auswirken, immer auf den ganzen Prozess auswirken (z.B. SIGKILL beendet den ganzen Prozess).

Um auf das Eintreffen eines Signals zu warten, stehen die `sigwait`, `sigwaitinfo` (setzt `errno`) und `sigtimedwait` (setzt `errno`) zur Verfügung.

Weitere Synchronisierungsmöglichkeiten

Semaphoren

Siehe Interprozesskommunikation. Hier verwenden wir Semaphoren aus `semaphore.h`

Read/Write-Sperren

Mehrere gleichzeitige Leser erlaubt, aber nur ein Schreiber erlaubt.

Barrier

Ist ein Synchronisationspunkt, der erst dann freigegeben wird, wenn eine bestimmte Anzahl von Threads an dieser Barriere angekommen sind.

Spinlocks

Zur Synchronisation auf Multiprozessorsystemen.

Zusammenfassungen

Dieses Kapitel gibt eine Übersicht aller Zusammenfassungen der Folien. Vermutlich sind diese Themen in der Klausur relevant.

Einführung

- Einordnung, Zweck und Qualitäten eines Betriebssystems

Historische Entwicklung

- 4 Generationen
- Primitive BS, Mehrprogramm- und Mehrbenutzersysteme

Betriebssystemstrukturen

- Kernel-typen, Arten von BS

Systemaufrufe

- Fehlerbehandlung, Unix file I/O

Systemkontexte

- Übergänge, Preemption, Schutzmechanismen

Grundsätzliche Prozesskonzepte

- Was (Programminstanz)
- Prozesssicht aufs System (Adressraum, Datei-Handles, virtuelle CPU, etc.)
- Optimierung: Durchsatz, Latenz, etc.

Prozesse aus Anwender/Anwendungssicht

- Erzeugen & ausführen (fork, exec)
- Signale
- Beenden (exit, kill)

Prozesse aus Kernelsicht

- Prozesszustände
- PCBs und deren Verwaltung
- Scheduling

Context Switch

- Unterbrechend oder nicht unterbrechend

Multi-Threading

- Prozesse vs. Threads
- Warum (Nebenläufigkeit, Overhead)
- Scheduling vs. Ressourcenverwaltung

Thread API

Arten von Threads

- Kernel Threads
- User Threads
- Hybrid Threads
- Vor- und Nachteile der Threadtypen

Scheduling-Algorithmen

Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):

- Exekutiere den Job mit der geringsten (Rest)laufzeit
- Vorteile: Optimale durchschnittliche Laufzeit für Jobs mit variierender Länge
- Nachteile: Vorhersage, unfair

Multi-Level Feedback Scheduling:

- Mehrere Warteschlangen mit verschiedenen Prioritäten
- Automatische Anpassung von Prioritäten um SJF/SRTF zu erreichen.

Lotterie Scheduling:

- Jeder Thread bekommt eine prioritätenabhängige Zahl von Losen (kurze Jobs bekommen mehr Lose)
- Reserviere eine Mindestzahl von Losen für jeden Job um Fortschritt und Fairness zu gewährleisten.

• Anforderungen an nebenläufige Prozesse

- konsistenter Zugriff auf gemeinsame Daten
- vorgegebene Reihenfolge von Aktionen

• Konsistenz durch wechselseitigen Ausschluss

• Reihenfolge durch Synchronisierung

• Kritischer Abschnitt

- Aktionen, die gemeinsame Daten manipulieren
- mit Konstrukten zur Synchronisierung gesichert

- Sicherung des kritischen Abschnitts
 - Softwarelösungen (z.B. Dekker)
 - Test and Set
 - Unterstützung durch Betriebssystem
 - Semaphore
 - init, wait und signal
 - Monitor
 - Zusammenfassung von gemeinsamen Objekten, Zugriffsfunktionen und Bedingungsvariablen
 - 4 Deadlock Bedingungen
 - Wechselseitiger Ausschluss
 - Hold and Wait
 - Kein unfreiwilliges Abgeben von Ressourcen
 - Circular Wait
 - Deadlock Prevention
 - Deadlock Avoidance
 - Erkennen eines Deadlocks
 - Recovery
- In dieser Vorlesungseinheit wurden die klassischen Formen der Interprozesskommunikation besprochen
- Pipes
 - FIFOs
 - Message-Queues
 - Semaphore
 - Shared-Memory
- Daneben gibt es neuere Formen
- Stream Pipes und benannte Stream Pipes
 - Erlauben zum Beispiel den Austausch von Filedeskriptoren

Client-Server-Eigenschaften klassischer IPC

- Pipes
 - Client-Prozess richtet zwei Pipes ein
 - Mit **fork** und **exec** wird ein eigener Server-Prozess gestartet
 - Kommunikation über Pipes
- FIFOs
 - Server und Clients sind unabhängige Prozesse
 - Für die Anforderungen der Clients reicht eine FIFO aus
 - ✓ Diese ist allen Clients bekannt
 - ✓ Jeder Client schickt bei seiner Anforderung seine Prozess-ID mit
 - Für die Server-Antworten je Client muss eine eigene FIFO eingerichtet werden
 - ✓ Eindeutiger Name durch Verwendung der Prozess-ID im FIFO-Namen
 - ✓ Nur der entsprechende Client sollte aus der FIFO lesen
 - Entsprechende Fehlerbehandlung notwendig

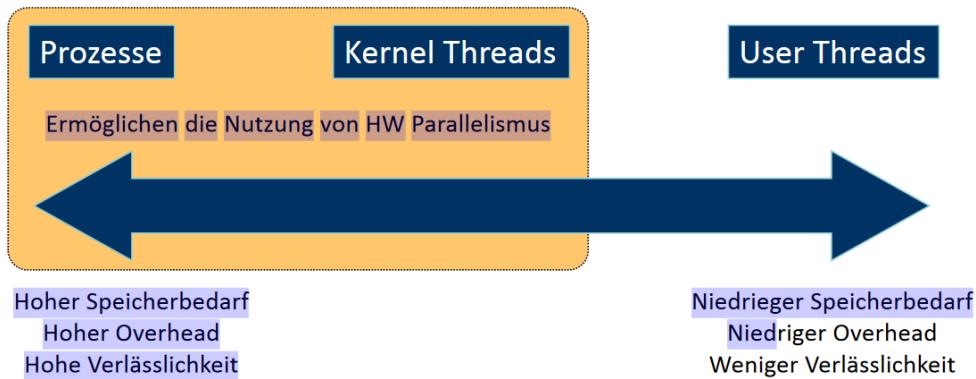
■ Message-Queue

- Message-Typ wird verwendet, um den Empfänger der Nachricht festzulegen
 - ✓ Alle Client-Anfragen haben den Typ 1 (Prozess-ID wird in der Anfrage mitgeschickt) und benutzen eine Message-Queue (Schlüssel bekannt)
 - ✓ Bei den Server-Antworten wird dann die Prozess-ID als Message-Typ verwendet
- Client-spezifische Message-Queue
 - ✓ Es existiert eine Message-Queue für alle Client-Anforderungen (über vereinbarten Schlüssel allen Clients bekannt)
 - ✓ Jeder Client richtet seine private Message-Queue (IPC_PRIVATE) ein und schickt die Kennung bei seiner ersten Anfrage mit

■ Shared-Memory und Semaphore

- Ähnlich wie Message-Queue
- Zusätzlich Semaphore oder auch Signale benutzen

Übersichten

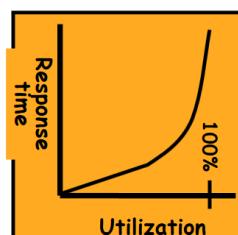


Wann ist Scheduling relevant?

Zu wenig Ressourcen

Wann sollte man einen besseren Rechner kaufen?

- (oder mehr Speicher, besseren Bus, mehr Cores, ...)
- Kaufe, wenn man dadurch bessere Performance erzielt
 - Wenn schlechtere Performance teuer ist.
 - Kaufe erst wenn Nutzung bei 100 % liegt?
Nein!
 - Laufzeiten gegen unendlich bei hoher Nutzung



Interpretation des Diagramms

- Viele Scheduling Algorithmen funktionieren im linearen Bereich sehr gut, versagen sonst.
- Kaufe mehr/bessere Ressourcen im Bereich des Knies der Kurve