System-Programmierung

Klaus Kusche

Inhalt (1)

Eigentlich:

- *Zu 15 %:*
  Allgemeine ***Grundlagen***
  der ***Parallel-Programmierung***
- *Zu 65 %:*
  Systemnahe Parallel-Programmierung
  in C unter Unix / Linux
- *Zu 20 %:*
  Andere ***systemnahe Funktionen***
  in C unter Unix / Linux

Inhalt (2)

- Grundlagen & Überblick
- `system` & `popen`
- Signale
- C File-I/O vs. Unix File-I/O, `select`, ...
- `fork`, `wait`, `exec` (Daemonisierung, ...)
- Pipes
- Shared Memory, `mmap`
- Locks usw.

Ziel

Verständnis der Begriffe und Konzepte
(auf Betriebssystem-Ebene):

- *Prozesse* & Threads
- *Shared Memory*, Pipes, ...
- *Locks*, *Critical Regions*, Deadlocks, ...
- usw.

... und Beherrschung von deren praktischer Anwendung in C unter Linux

Voraussetzungen

- Beherrschung von ***C***
- *Bedienung* von ***Linux***
  (auch auf der Commandline)
- *C-Entwicklung* unter ***Linux***
  (auf der Commandline ist empfohlen,
  aber nicht zwingend)

Warum parallele Programmierung?

- Höhere ***Performance***
- Bessere ***Skalierbarkeit***
- Ausnutzung moderner Hardware (***Multicore-Prozessoren***)
- Höhere ***Robustheit und Sicherheit***:
  - ***Isolation*** von Abstürzen auf einzelne Programmteile
  - Betrieb exponierter Programmteile mit ***reduzierten Rechten*** oder in einer ***Sandbox***
  - ***Auslagerung*** sensibler Daten in separate Mini-Prozesse

Und warum nicht?

Das *menschliche Hirn* ist dafür nicht gebaut:

Paralleler Code
ist schwer vorzustellen
bzw. nachzuvollziehen!

==> Um Größenordnungen
höhere Fehleranfälligkeit

==> Viel ***schwerer zu testen*** und zu debuggen

(spezielle Tools, z.B. Thread Sanitizer)

Parallelisierungs-Strategien (1)

- **Pipeline:**
  Daten fließen *nacheinander* durch
  mehrere *unterschiedliche Bearbeitungs-Prozesse*
- **Master / Slave** (bzw. Worker):
  Master *verteilt* die Arbeit *"portionsweise"*
  (z.B. pro Request)
  auf *mehrere gleichartige Arbeits-Prozesse*
  (*"wer gerade frei ist"*)
- Sonderfall bei Netzwerk-Servern: **"accept + fork"**
  1 Worker pro einlangendem Client-Connect

Parallelisierungs-Strategien (2)

- **Partitionierung der Daten,**
  jeder Prozess macht *dieselbe Arbeit*
  auf einem *Teil der Daten*:
  - Supercomputing, phys. Simulationen
  - Große Datenbanken
- **Asynchrones I/O**, async. Event-Handling,
  GUI-Programmierung
- Sonderfall **Browser**: *1 Prozess pro Tab*
  - *Schneller* (Tabs blockieren einander nicht)
  - *Sicherer* (Absturz killt nur betroffenes Tab)

Grundsätzliches

Die ***Konzepte***
(Prozess, Thread, Shared Memory, Mutex, ...)

sind ***überall ähnlich*** und
relativ portabel bzw. universell!

(zwischen *verschiedenen Betriebssystemen* und
zwischen *verschiedenen Programmiersprachen*)

Die ***konkreten Implementierungen***
unterscheiden sich primär im API:

"Dasselbe sehr unterschiedlich verpackt!"

Abgrenzung: “Parallel” (1)

Hier:

Auf <u>einem</u> Rechner

(alle Teile des Programms greifen
auf <u>dasselbe Betriebssystem</u>
und <u>dasselbe RAM</u> zu,
<u>kein Netzwerk</u> dazwischen)

Gegenteil (hier *<u>nicht</u>*!):

“<u>Verteilte</u>” Programmierung,
z.B. RPC (Remote Procedure Call)

(auf <u>mehreren</u> Rechnern via <u>Netzwerk</u>)

Abgrenzung: “Parallel” (2)

Ebenen der Parallel-Programmierung:

- In der ***Shell*** (hier nicht):
  Pipelines
  Hintergrund-Programme, ...
- In ***höheren Programmiersprachen*** (hier nicht):
  Java, C#, C++, ...

  Plattform-unabhängige Konstrukte,
  komfortabel, *hohes Abstraktionsniveau*
- In ***C*** (siehe nächste Folien)

Abgrenzung: “Parallel” (3)

Parallelität in C:

- Microsoft API & andere ***proprietäre API's***
- Klassische ***Unix/Linux Syscalls*** (seit ~1970),
  SysV IPC (“Unix System V Inter Process Communication”, 1983),
  POSIX IPC (“Portable Operating System Interface”, 1993):

  Parallelität ***auf <u>Prozess</u>-Ebene***

  Eher *<u>low-Level</u>*, *<u>unkomfortabel</u>* zu programmieren:

  Näher bei *“was das System intern macht”*
  als bei *“was der Programmierer denkt”*

Abgrenzung: “Parallel” (4)

Parallelität in C (Fortsetzung):

- ***POSIX pthreads*** (seit 1995):

  Parallelität ***auf Thread-Ebene***
  (“leichtgewichtig” und effizient)

  Etwas *abstrahierter und komfortabler*,
  Richtung Konzepte von Java, ...

- ***C-Library***-Features: **`system`** und **`popen`**

  Sehr *einfach* zu verwenden,
  aber nur für *ganz einfache Aufgaben* anwendbar

Abgrenzung: “Parallel” (5)

Unix Syscalls/IPC und/oder **pthreads** sind Grundlage

- des ***gesamten “klassischen” parallelen C-Codes*** (Netzwerk-Server-Dienste, ...)
- von beträchtlichen Teilen ***alten, parallelen C++-Codes*** (vor C++ 11)
- der *internen Parallelitäts-Implementierung* ***aller Sprachen und Libraries auf Unix/Linux***

Wir machen kurz **`system`** & **popen**,
vor allem Unix/SysV/POSIX Syscalls & IPC,
ev. kurz pthreads.

Abgrenzung: “Unter Unix/Linux” (1)

Warum?

- Seit teilweise über 40, <u>*zumindest über 20 Jahren*</u> ***stabiles***, fast unverändertes API, ***etabliert*** und ***bewährt***
- Offizieller, ***Hersteller-unabhängiger Standard:***

POSIX = IEEE Std 1003.1abc

- Gut ***dokumentiert***, Beispiele vorhanden, ...

==> Oft ***gelehrt***, weitum ***bekannt***

Abgrenzung: “Unter Unix/Linux” (2)

Unterstützt von:

- ***Allen Unix-Systemen***, Linux, macOS, ...
- Vielen ***Echtzeit-Systemen*** (vxWorks, QNX, ...)
- Im Prinzip ***Windows***:

  POSIX-API bzw. Linux-Syscall-API wird von MS *offiziell angeboten*
  (“WSL”, als Alternative zu Windows-eigenem Environment)

  Ist aber *nicht nativ* implementiert,
  sondern wird auf Windows umgesetzt
  (läuft betreffend Parallel-Features teilweise etwas unrund...)

  Ok für kleine Anwendungen, suboptimal für hochkritischen Code
  (besser mit Windows-spezifischen Calls arbeiten)

Abgrenzung: “Unter Unix/Linux” (3)

Warum nicht Windows?

- ***Windows Syscalls*** sind <u>*nicht direkt zugänglich*</u> (nicht dokumentiert, nicht offiziell unterstützt)
- ***Windows C-Library***-Parallel-Features
  - sind nicht Langzeit-stabil
  - sind nicht mehr praxisrelevant
  - haben auch bei MS geringen Stellenwert
- Windows Parallelität heute primär in ***C# / .net***: Sehr praxisrelevant, aber nicht systemnah: Abstraktionen sehr ähnlich zu Java und C++

Abgrenzung: “Systemnah”

Wir programmieren

<u>nicht im</u> Betriebssystem (Kernel)

(d.h. wir schreiben keine Device-Treiber o.ä.)

sondern schreiben

normale (parallele) <u>Anwendungsprogramme</u>

unter

möglichst <u>direkter Verwendung</u>
von <u>Betriebssystem-Aufrufen</u>

(d.h. <u>ohne</u> die “Luxus-Features” vordefinierter Libraries)

Abgrenzung: “In C” (1)

- **C** bietet den *direktesten und ballastfreisten* ***Zugriff auf Systemaufrufe***
- **C++** bietet systemnah die gleichen Möglichkeiten, aber *keinen Mehrwert*, eher Komplikationen (und C++-11 Parallelität ist *nicht* systemnah)
- **Assembler** mit direkten Syscalls wäre technisch eine lehrreiche Alternative, aber praktisch nicht relevant
- **Shell** ist praktisch sehr relevant, aber gehört in ein anderes Lehrfach

Abgrenzung: “In C” (2)

C enthält

keine speziellen Sprachkonstrukte
und ***keine spezielle Compiler-Unterstützung***
für die Parallel-Programmierung!

Realisierung nur mit

normalen Library-Funktionen,
Typen, Makros, ...
aus (zusätzlichen) Header-Files

(funktionieren mit jeder Sprache,
die die C-Library aufrufen kann)

Abgrenzung: “In C” (3)

Unter Linux *an der Dokumentation erkennbar*:

- Funktionen aus **man Section 2**:

  Reine Wrapper-Funktionen *um einen* *System Call*:
  Keine eigene Logik in der Library,
  Aufruf geht ***1:1 an das Betriebssystem*** durch

- Funktionen aus **man Section 3**:

  Library-Funktionen, eine Schicht höher:
  Mit ***eigener Logik in der Library*** implementiert,
  mehr Abstraktion / Komfort / Funktionalität
  als das entsprechende Betriebssystem-Feature

Abgrenzung: “In C” (4)

Wenn C keine vordefinierte Library-Funktionen
für einen bestimmten System Call bietet:

Universelle Funktion “**`syscall`**”

Kann *jeden beliebigen System Call direkt aufrufen*!

Technisch:

Das Betriebssystem hat nicht eine Einsprung-Adresse pro System Call,
sondern im Unterschied zu einer Library
für *alle* System Calls *einen einzigen, gemeinsamen* Aufruf:

Der gewünschte Call wird *durch eine Nummer* angegeben,
die als *erster Parameter* übergeben wird.
Die **`syscall`**-Funktion erlaubt die *freie Angabe* der Call-Nummer.

Parallelprogrammierung (1)

Das **Problem** der Parallelprogrammierung ist

***nicht* die Erzeugung**
paralleler Abläufe
(die ist relativ “einfach”),

sondern

deren ***Kommunikation***
(Datenaustausch)
und ***Synchronisation***.

Parallelprogrammierung (2)

Für die ***Parallelität*** gibt es
<u>*zwei grundsätzlich verschiedene Konzepte*</u>:

Threads oder ***Prozesse***

Grobe Einschätzung:

- **Threads** sind <u>*effizienter*</u>, universeller
  und meist <u>*einfacher*</u> zu programmieren
- **Prozesse** sind <u>*robuster*</u> bzw. sicherer

<u>*Gemischte*</u> Verwendung ist <u>*selten*</u>, aber <u>*möglich*</u>
(mehrere Prozesse mit jeweils mehreren Threads)

Parallelprogrammierung (3)

Auch für die ***Kommunikation*** gibt es
zwei grundsätzlich verschiedene Konzepte:

Shared Memory oder ***Message Passing***

Grobe Einschätzung:

- **Shared Memory** ist *viel universeller*,
  aber *schwierig zu programmieren* & fehleranfällig
- **Messages** eignen sich für *viel weniger Aufgaben*,
  aber man kann *weniger falsch machen*

Im Regelfall *entweder/oder*, *nicht beides gleichzeitig!*

Prozesse (1)

Aus *Programmierer*-Sicht:

1 Prozess =
ein in Ausführung befindliches Programm
(Code & Daten)

Aus *Betriebssystem*-Sicht:

Prozesse sind die Einheiten,

- denen ***Speicher*** zugewiesen wird
- denen bestimmte ***Rechte*** zugewiesen werden
- die ***Files*** offen haben und vieles mehr ...

Prozesse (2)

==> *Alles*, was *außerhalb des Kernels* läuft,
läuft *in Prozessen*
(auch Dienste, Server, ...)

==> Bei *jedem Starten eines Programms*
entsteht ein *neuer Prozess*
(bei sehr komplexen Programmen auch *mehrere*)

Programm *mehrmals gestartet*
==> *Mehrere* (unabhängige!) *Prozesse*!

Siehe Windows ***Task Manager***,
Linux "**ps**", "**top**", ...

Prozesse (3a)

Jeder Prozess hat:

- Eine *eindeutige Nummer*: ***Process-Id (Pid)***
- Seinen *eigenen* ***virtuellen Adressraum***:
  - **Logisch**:
    Eigene, *vor anderen Prozessen geschützte* ***Daten***
  - **Technisch**:
    Eine eigene ***Pagetable***
    Ihm (exklusiv) ***zugeordnetes RAM***

Prozesse (3b)

- Ein dazugehöriges ***Programm***
  (**.exe**-File und Shared Libraries)
- ***Offene Files*** und Netz-Verbindungen,
  zugeordnete IPC-Ressourcen
  (Shared Memory, Message Queres, Pipes), ...
- Ein ***aktuelles Arbeitsverzeichnis***
- Sein eigenes ***Environment***
  (seine eigene Kopie der Environment-Variablen)

Prozesse (3c)

- ***Rechte:***
  Benutzer & Gruppe
  (real & effective, vom Starter oder vom **.exe**-File)
  Umask, *Gruppen-Liste*, ...
- Eine ***Vater-Process-Id***:
  "Wer hat diesen Prozess gestartet?" &
  "Wer wird von seinem Ende informiert?"
  Prozess-Hierarchie = Baum mit "Urvater" Pid 1
- ***Session-Id*** & ***"Controlling Terminal"***
  z.B. für Ctrl/C und
  für automatisches Beenden beim Abmelden der Sitzung

Prozesse (3d)

- **Limits** ("**`ulimit`**": max. reales & virtuelles *RAM*, max. offene *Files*, max. CPU-Verbrauch, ...)
- Eine ***Scheduling-Priorität*** und ***-Strategie***
- ***Signal***-Einstellungen,
  Timer
- ***Ressourcen-Statistiken***
  (User & System CPU Time,
  Paging, I/O, Startzeitpunkt, ...)
- ***Namespaces*** in Container-Environments

Prozesse (4a)

Unter Unix / Linux ist

- *Parallelen Prozess* im *selben Programm* starten
- *Neues Programm* starten (z.B. in der Shell, am Desktop)

dasselbe Ding:

System Call "**fork**"

Erzeugt einen *neuen Prozess*
als *exakte Kopie* des Aufrufers

==> Zwei *unabhängige Prozesse* (alt + neu),
beide führen *dasselbe Programm* aus!

Prozesse (4b)

Zum *Starten eines neuen Programms*:

2 Schritte nötig!

- Zuerst **`fork`**
- Dann im neu erzeugten Prozess:

***System Call* "exec"**

Ersetzt das laufende Programm
im aktuellen Prozess
durch *ein anderes Programm*

Prozess bleibt dabei *gleich*, kein neuer Prozess!

Threads (1)

Aus *Programmierer*-Sicht:

1 Thread =
ein paralleler Ablauf
innerhalb eines laufenden Programms

Jeder Thread (nicht jeder Prozess!)
befindet sich *zu jedem Zeitpunkt*
an *genau einer Stelle* im Programmcode

Threads (2)

Aus *Betriebssystem*-Sicht:

Threads sind die Einheiten,
denen vom Scheduler Rechenzeit
(= ein CPU-Core) zugewiesen wird

Mehrere Threads eines Programms
können *gleichzeitig* ausgeführt werden
(auf Multicore-CPU's)

(neben Anwendungs-Threads gibt es auch
Kernel-interne *Betriebssystem-Threads*,
mit denen befassen wir uns *hier nicht*!)

Threads (3)

Jeder Thread enthält nur

- seinen eigenen ***Instruction Pointer***:
  "An welcher Stelle im Programm
  steht oder rechnet der Thread gerade?"
- seinen eigenen ***Speicherbereich für den <u>Stack</u>***,
  <u>innerhalb</u> des Adressraumes des Vater-Prozesses
  ==> jeder Thread hat seine eigenen
  lokalen Variablen / Parameter / Return-Adressen
- und seinen eigenen ***Stackpointer***
  auf die *<u>Daten der gerade laufenden Funktion</u>*

Threads (4)

Alles andere "erbt" der Thread von seinem Prozess!

==> Jeder Thread gehört zu *genau einem* Prozess

==> Ein Thread kann *nicht ohne Prozess* existieren

==> Aber *ein Prozess* kann *viele Threads* enthalten!

Unter Linux aus Kernel-Sicht:

Getrennte ***Prozess-Id's*** und ***Thread-Id's***

Jeder Thread hat seine eigene *eindeutige Thread-Id*,
aber alle Threads eines Prozesses haben *dieselbe Prozess-Id*

Threads (5)

Alle Threads eines Prozesses ...

- führen ***dasselbe Programm*** aus
- sehen ***denselben Speicher***
  unter ***denselben Adressen***:

  - Selbe *globale / statische* Variablen
  - Selber *Heap* (**`malloc`** / **`free`** bzw. **`new`** / **`delete`**)
  - *Pointer gelten* in allen Threads gleich!

  - Jeder Thread hat seinen *eigenen Stack*, aber kann rein speicherverwaltungs-technisch (mittels Pointer) *auch auf die Stacks der anderen* zugreifen

Threads (6)

- haben ***dieselben offenen Files***
  und ***IPC-Ressourcen***
- haben ***dieselben Rechte***
- haben ***dasselbe Environment***, ***Working Dir***, ...

Je nach System
entweder pro Prozess oder pro Thread:

- ***Scheduling-Priorität*** und ***-Strategie***
- ***Core Affinity***

Threads (7)

Neue *Speicherklasse*
(bei Bedarf *explizit* anlegen / deklarieren):

TLS = "Thread Local Storage"

- Verhält sich *innerhalb* eines Threads
  wie *global* bzw. statisch
- Aber es existiert eine *separate Kopie pro Thread*
  (lokal zum jeweiligen Thread)

Beispiel:
C-Fehlercode-Variable **`errno`**

Threads (8)

Ein “normaler” Prozess
(ohne explizite Thread-Programmierung)
enthält konzeptionell bei seinem Start
automatisch einen Thread,

der aber nicht separat
angelegt / verwaltet / angezeigt wird
(d.h. nicht ganz so explizit manipulierbar ist
wie händisch erzeugte Threads)

Unter Linux ist das der Thread mit

Kernel-Thread-Id = Process-Id

Prozesse vs. Threads (1)

Eine ***Thread-<u>Umschaltung</u>*** ist

viel <u>schneller</u> und <u>einfacher</u>
(betr. HW & SW)

als ein Prozesswechsel:

Der ***Adressraum*** (MMU-Kontext, Pagetable)
wird ***<u>nicht</u> umgeschaltet***

==> Der *<u>TLB</u>* (MMU Cache) wird *<u>nicht geleert</u>*!

Prozesse vs. Threads (2)

Auch das ***Erzeugen und <u>Starten</u>***
(und das Beenden) eines Threads ist

viel schneller und einfacher

als das eines neuer Prozess (aus demselben Grund)

==> Threads heißen daher auch

"<u>Lightweight</u> processes" (LWP)

Prozesse sind im Gegensatz dazu organisatorisch

sehr "<u>fett</u>"!

Prozesse vs. Threads (3)

Der ***Datenaustausch*** zwischen Threads ist auch

einfacher zu programmieren und ***effizienter***,

weil alle Threads eines Prozesses
direkt auf dieselben Daten zugreifen können...

... aber diese auch zerstören können!

Daher aus Betriebssystem-Sicht:

Absturz eines Threads

==> Betriebssystem ***beendet*** meist
den ***ganzen Prozess*** (*alle* Threads)

Prozesse vs. Threads (4)

==> Threads eignen sich ***<u>nicht</u>*** als

"Sicherheits-Container"

Für

Robustheit & Sicherheit

(Isolierung, Sandboxing, spezielle Rechte, ...)

müssen

<u>separate</u> Prozesse

verwendet werden:
Threads reichen nicht!

Thread-Erzeugung in Linux (1)

Aus *Benutzersicht*:

Prozesse & Threads sind *separat* wie besprochen

In *anderen Systemen*:

Zwei *verschiedene Syscalls* zur Erzeugung

In *Linux intern* (nicht auf Benutzer-Ebene):

Für beides *gemeinsamer Syscall* "**`clone`**"

Über Parameter fein *einstellbar*:
Was wird geerbt / geshared, was ist separat, was wird kopiert, ...

==> **`clone`** kann auch klassisches **`fork`**

Thread-Erzeugung in Linux (2)

Faustregel:

Zuerst parallele Prozesse starten
(wenn gewünscht)

Dann erst in jedem Prozess für sich
Threads starten

Nicht umgekehrt!

Ein **fork** *ohne* **exec** in einem *Multithreaded-Prozess*
ist meist eine *schlechte Idee!*
(der neue Prozess hat im Kernel nur *einen* Thread,
aber auf Library-Ebene den *gesamten internen pthread-Status* geerbt)

Shared Memory (1)

Mehrere parallele Abläufe

"sehen" <u>denselben</u> Speicher (im RAM)

und können dort *<u>unabhängig voneinander dieselben</u>* Daten *<u>lesen und schreiben</u>*.

Der Zugriff erfolgt meist genauso
wie normales Auslesen und Zuweisen
von Variablen
(bzw. Arrays oder dyn. Datenstrukturen):

Es gibt ***keine speziellen Konstrukte***
zum Datenaustausch!

Shared Memory (2)

Alle ***Threads*** eines Programms laufen
im selben Adressraum

==> Alle Threads "sehen" automatisch

- dieselben ***globalen & statischen Variablen***
- denselben ***Heap*** (dynamische Daten)

unter *denselben Adressen*:

Pointer sind über alle Threads hinweg gültig!

(im Prinzip kann ein Thread sogar auf die *lokalen Variablen* eines anderen Threads zugreifen, wenn er einen Pointer darauf bekommt, aber das sollte man *nur in Ausnahmefällen* machen!)

Shared Memory (3)

Bei ***parallelen <u>Prozessen</u>:***

Jeder hat seinen ***<u>eigenen</u> Adressraum!***

==> ***<u>Kein</u> gegenseitiger Zugriff***

- auf *<u>globale Variablen</u>*
- auf den *<u>Heap</u>* (jeder hat seinen eigenen!)

==> Pointer "gelten"

nur <u>innerhalb</u> ihres eigenen Prozesses!

==> Von sich aus

<u>kein</u> gemeinsamer Speicher!

Shared Memory (4)

Zwischen Prozessen daher:

Gemeinsame Speicherbereiche

- müssen *explizit angelegt* werden
- und *explizit* in den Adressraum einzelner Prozesse *eingeblendet* werden

==> Der *Zugriff* erfolgt über

den ***Pointer auf den Anfang des Bereiches***, den das System beim Einblenden liefert

Die Bereiche existieren unabhängig von den Prozessen!

Shared Memory (5)

Achtung!!!

- In expliziten Shared-Memory-Bereichen
  gibt es ***keine* dynamische Speicherverwaltung**
  (**`malloc`** / **`free`**, **`new`** / **`delete`**)!

  ==> Wenn nötig *selbst programmieren*!

- *Derselbe Shared-Memory-Bereich*
  wird in verschiedenen Prozessen meist
  an verschiedenen Adressen eingeblendet!

  ==> ***Pointer*** im Shared Mem ***funktionieren nicht***!

  ==> Dyn. Datenstrukturen usw. statt mit Pointern z.B.
  mit *Index ab Shared-Mem-Anfang* programmieren!

Shared Memory (6)

Mögliche Realisierungen:

1) ***SysV IPC Shm***

   (meist in älteren Programmen)

2) **`mmap`** = ***Memory mapped Files***

   (oft in neueren Programmen)

3) **`mmap`** in Kombination mit ***POSIX Shm***

   (aktueller Standard, aber noch eher selten)

Message Passing (1)

Beim ***Message Passing***

- werden Daten *meist blockweise*
- durch *explizite Aufrufe* (*"Senden" / "Empfangen"*)
- von einem parallelen Prozess / Thread
  zu einem anderen *geschickt*

Die Daten werden dabei ***kopiert***
(d.h. sind nicht "gemeinsam",
sondern jede Seite hat ihre *eigene, private Kopie*).

Message Passing (2)

Mögliche Realisierungen:

1) ***SysV IPC Message Queue***
   bzw. ***POSIX Message Queue*** (blockweise)
   (selten, mühsam, aber meist effizient)

2) Normales ***File-I/O*** (**char**-Stream)
   über normale ***Pipes*** oder ***Named Pipes*** (*FIFO's*)
   (verwenden wenn geeignet,
   geringster Lern- und Schreib-Aufwand!)

3) ***Unix Sockets*** oder ***Netz-Verbindungen***
   (häufig, aber hier nicht)

Synchronisation (1)

Vor allem *in Verbindung mit Shared Memory*,
so gut wie nie bei Message Queues.

Hauptziel:

Verhinderung kollidierender Zugriffe
auf das Shared Memory,

wohldefinierte Reihenfolge der Zugriffe

Siehe separater Vortrag!

Synchronisation (2)

Wichtigste Konzepte:

Lock / Mutex / Semaphore
(das ist alles ungefähr dasselbe),

vor allem zur Implementierung

Kritischer Regionen

Achtung: *Vordefinierte* Konstrukte verwenden!!!

- *Nicht selber* basteln!
- *Keine Spinlocks (= Warteschleifen)* programmieren!

Synchronisation (3)

Mögliche Realisierungen von Locks:

1) ***SysV IPC Semaphore*** bzw. ***POSIX Semaphore***

   (in Verbindung mit *Prozessen*)

2) ***pthreads Mutex***

   (in Verbindung mit *Threads*)

Wenn es nur um's *"Warten und Aufwecken"* geht:

Ev. Signale (**`pause`** und **`kill`**),
einige andere Konstrukte...

Synchronisation (4)

Konzepte "darunter":

<u>Atomare</u> Operationen
(bzw. atomare Variablen)

Idee:

Mach einen ***<u>zusammengehörenden</u>***
Lese- und Schreibzugriff auf eine Variable
(z.B. Wert um 1 erhöhen, Wert austauschen,
auf 1 setzen wenn 0, ...)

<u>ohne</u> *dass ein anderer* ***dazwischen zugreifen*** *kann*
(erfordert *<u>Hardware-Unterstützung</u>*!)

Synchronisation (5)

Konzepte “darüber”:

- ***Condition Variables***
- ***Barriers***
- ***Call-once***
- ...

(z.B. in **`pthreads`** enthalten)

Synchronisation (6)

Alternative Konzepte:

- ***Lockfreie*** Algorithmen
  (basierend auf *atomaren* Operationen)
- ***Transactional Memory***
- ...

SysV IPC API versus POSIX API

Das POSIX API für Shm / Sem / MsgQueue
ist eine *später* erfolgte ***"Verschönerung"***
des ursprünglichen SysV IPC Shm / Sem / MsgQueue API:

- Die *Aufrufe* sind ***inkompatibel*** (andere Namen, ...),
  POSIX ist deutlich ***komfortabler / lesbarer***.
- Die *Funktionalität* dahinter ist sehr ***ähnlich***,
  meist steckt hinter beidem
  dieselbe Implementierung im Betriebssystem.
- SysV IPC ist wesentlich *älter*,
  deshalb in bestehendem Code
  viel *häufiger* anzutreffen als POSIX.

`system` & **`popen`** (1)

- *C-Library*-Funktionen, keine Syscalls:
  Verwenden intern **`fork`**, **`exec`**, **`pipe`**, **`wait`**, ...
- Starten das Programm *durch einen Shell-Aufruf*

  ==> Erlaubt sind nicht nur *exe-Files*,
  sondern auch *Shellscripts*, *Builtin-Befehle*, ...

  ==> Befehl kann *Pipes*, *Redirects*, ... enthalten
- *Suchen* den Befehl im **`PATH`**

  ==> Potentielles *Sicherheitsproblem*

  ==> *Nicht* in Code mit **`suid`** / **`sgid`** verwenden!

`system` & **`popen`** (2)

Ausgeführtes Programm läuft in einem

eigenen Prozess

Dieser ist ein *Sohn*, d.h. *erbt* viel, u.a.

- *Rechte*, *Environment* (vorher putzen?), *Working Dir*, ...
- Aktuelles **`stdin`**, **`stdout`**, **`stderr`**

  (außer dem, der durch **`popen`** geändert wird)
- *Ignorierte* Signale

`system` & **`popen`** (3)

Returnwert von **`system`** und **`pclose`**:

Exitstatus des Programms / der Shell,
codiert wie bei **`wait`**-Funktion

==> *Decodierung* mit **`WIF`**...-Makros

Liefert u.a.,

- ob das Programm an einem *Signal* starb
  (und an welchem)
- ob die Shell das Programm *nicht starten konnte*
  (**`exit`** mit **`126`** oder **`127`**)

`system`

... ist ***synchron***:

Der Aufruf kehrt *erst zurück*,
wenn das gestartete Programm
geendet hat!

... erlaubt

keinen Datenaustausch
mit dem gestarteten Programm!

popen (1)

... ist ***asynchron***:

Kehrt *"sofort"* zurück,
Aufrufer & gestartetes Programm
arbeiten *parallel* weiter!

... ***verbindet*** Aufrufer & gestartetes Programm

mit einer ***Pipe***
(wie **|** auf der Shell)

... liefert *ein Ende* der Pipe als **FILE***-*Returnwert*

popen (2)

Die Pipe ist *unidirektional*:

- **Entweder:**
  Der *Aufrufer liest* aus dem **popen `FILE*`**,
  was *gestartetes Programm* auf **`stdout`** schreibt
- **Oder:**
  Gestartetes Programm bekommt auf **`stdin`**,
  was der *Aufrufer* in den **popen `FILE*`** *schreibt*.

Eine Pipe ist *gepuffert* (meist mit 4 KB).

popen (3)

- *Schreiber endet* vor dem Leser:

  ==> Leser bekommt beim Lesen ***"end of File"***.

- *Leser endet* vor dem Schreiber:

  ==> Schreiber bekommt beim Schreiben
  Signal **`SIGPIPE`**
  (default: *Beendet* das Programm)

- **`pclose`** *wartet,*
  bis der gestartete Befehl *geendet* hat!

Signale (1)

Signale sind ***ähnlich Interrupts***:

<u>Unterbrechen</u> ein Programm
spontan & an <u>irgendeiner</u> Stelle

4 mögliche Reaktionen (pro Signal *<u>festlegbar</u>*):

- Programm wird ***sofort beendet*** (ev. mit *<u>Core</u>*)
- ***Signal Handler*** wird ausgeführt,
  dann macht das Programm weiter, wo es war
- Signal wird völlig ***ignoriert***
- Signal ist ***blockiert*** / ***maskiert*** (wirkt erst *<u>später</u>*)

Signale (2)

Übersicht, Haupt-Doku unter Linux,
Liste aller Signale und ihres Auslösers:

`man 7 signal`

Signale sind uralt und *vor den Threads* entstanden ...

==> Das meiste gilt *pro Prozess*

==> Bei *Multithreaded*-Programmen heikel,
Doku *genau* lesen!

... und *"historisch gewachsen"*

==> mehrere in Details *inkompatible Versionen*

Signal Handler einrichten (1)

... bzw. Signal *ignorieren*, *Default*-Aktion wiederherstellen, bisher definierte Aktion auslesen, ...

Aktuelle, standardisierte Funktion:

`sigaction`

Nicht verwenden (alt oder nicht portabel):

`signal` (ursprüngliche Funktion)

`sysv_signal`, **`bsd_signal`**

`sigset`, **`sigvec`**

Signal Handler einrichten (2)

1.) *Signal-Handler-Funktion schreiben:*

Signal Handler =

Normale *Funktion im Programm*
die *aufgerufen* werden soll
wenn ein *bestimmtes Signal* eintrifft

Prototyp immer:

```
void mysighandler(int sig) {
   ...
}
```

Parameter **sig** (= Nummer des auslösenden Signals):
- Nützlich wenn gleiche Funktion für mehrere Signale eingerichtet
- Wird sonst meist ignoriert

Signal-Handler-Funktionen

... können auf *globale Daten* & den *Heap* zugreifen

... können mit **`sigsetjmp`** / **`siglongjmp`**
an eine *fixe Stelle im normalen Code springen*

(häufig, aber ***nicht schön!***)

... Dürften *"eigentlich"* (hält sich kaum jemand dran!)
nur "signal-sichere" Funktionen enthalten
(z.B. *kein* I/O !)

Grund:
Verhalten wäre *undefiniert*, wenn das Signal gerade
eine *"unsichere" Funktion unterbrochen* hat! *(z.B. I/O im I/O)*

Signal Handler einrichten (3)

2.) *Einrichten des Signal Handlers* z.B. am Anfang von **`main`**
(*vor* dem ersten Auftreten des Signals!):

```
struct sigaction action;
action.sa_handler = mysighandler;   // Handler
sigemptyset(&(action.sa_mask));     // oder ...fill...
action.sa_flags = SA_RESTART;

sigaction(SIG..., &action, NULL);
```

Dieser Aufruf des System Calls **`sigaction`**
sagt dem Betriebssystem

"Wenn in Zukunft einmal das Signal **`SIG...`** *auftritt,*
mach das, was in **`action`** *festgelegt wurde!"*
(hier: **`mysighandler`** *aufrufen)*

sigaction

Sonderfälle für **sa_handler**:

- **SIG_IGN** … *ignoriere* dieses Signal
- **SIG_DFL** … richte wieder die *Default-Reaktion* ein

Signalmaske **sa_mask** (Typ **sigset_t** = *Menge von Signalen*):

Die in **sa_mask** angegebenen Signale werden
während der Ausführung des Signal Handlers
vom Betriebssystem ***blockiert***

Sonst könnte die *Ausführung eines Signal Handlers*
durch einen *anderen* Signal Handler *unterbrochen* werden!

sigemptyset … keine, **sigfillset** … alle, siehe **man sigsetops**

`sa_flags: sigaction` Flags

`SA_RESTART` (siehe nächste Folie)

`SA_NOCLDSTOP` bei **`SIGCHLD`**:
`SIGCHLD` kommt nur bei Sohn-Prozess-*Ende*,
nicht wenn Sohn-Prozess nur *gestoppt* wird

`SA_RESETHAND`:
Signal Handler wird nur für *"one shot"* eingerichtet,
dann automatisch wieder Default-Verhalten

`SA_NODEFER`:
Eigenes Signal im Signal-Handler rekursiv erlauben
(default: Wird automatisch blockiert)

... und einige mehr (mit bitweisem **|** kombinierbar)

Signale und Syscalls

Signal mit Signal Handler kommt
während eines "langdauernden" Syscalls: *(z.B I/O)*

Default-Verhalten: ***Syscall endet sofort danach***
mit *Fehler* und **`errno`** = **`EINTR`**

Sehr störend / mühsam! ***(==> Schleife um alle Syscalls)***

SA_RESTART: Unterbrochene Syscalls
laufen nach dem Signal Handler *normal weiter*

Achtung: Wirkt *nicht für alle* Syscalls!
Sockets, **`select`** & **`poll`**, **`semop`**, ...
liefern *immer* **`EINTR`**, wenn sie unterbrochen werden

Auf Signal warten usw.

Auf *irgendein* Signal warten: **`pause`**

Warten *mit Maske* (auf *bestimmte* Signale):
`sigsuspend`

Warten *ohne Ausführen des Handlers*:
`sigwait`, **`sigwaitinfo`**, **`sigtimedwait`**

Signalmaske setzen: **`sigprocmask`**

Anstehende Signale abfragen: **`sigpending`**

Signalmengen **`sigset_t`** manipulieren / prüfen:
`sigaddset`, **`sigdelset`**, **`sigemptyset`**, **`sigfillset`**, **`sigismember`**

Signal auslösen

Außer der "eigentlichen Ursache" (z.B. Ctrl/C):

- **`kill`** (Befehl auf der Shell-*Befehlszeile*) oder
  `kill` (*Syscall* in einem Programm):

  Beliebiges Signal an fast beliebigen Prozess schicken
  (Sonderfälle wenn Pid = 0 / -1 oder Signal = 0)

- **`raise`**:
  Beliebiges Signal an den *eigenen* Prozess schicken

- **`abort`** und **`assert`**:
  "kill yourself with core dump" (**`SIGABRT`**)

Der Alarm-Timer

- … zählt im *Sekundentakt abwärts*
- … löst **`SIGALRM`** aus wenn er **`0`** wird
- … merkt sich *nur eine Zeit* pro Prozess,
  nicht mehrere
- … ist *"One-Shot"*, nicht periodisch
  (sonst im Signal Handler wieder starten)

Setzen / löschen / Restzeit abfragen:

`alarm`

Wenn *kein* Handler: **`SIGALRM`** *beendet* Programm

`alarm` & `sleep`

- In vielen (vor allem alten) Unixes (nicht Linux) verwendet **`sleep`** intern **`alarm`**

  Der Standard erlaubt das explizit!

  ==> **`sleep`** und **`alarm`** ***stören einander!***
  ==> **`sleep`** verursacht *Signale*

  ==> Nicht mischen, ist nicht portabel!

  (selbes gilt für **`usleep`**)

- ***Garantiert signalfrei*** und **`alarm`**-sicher:

  `nanosleep`, **`clock_nanosleep`**

Allg. Grundlagen

- Saubere C Fehlerbehandlung, ***Fehlermeldungen***: **`stderr`**, **`errno`**, **`strerror(errno)`**, ...

- Unterschied & Zusammenhang ***File-I/O***
  - ... auf ***C-Library-Ebene*** (mit **`FILE *`**)
  - ... auf Kernel- bzw. ***Syscall-Ebene*** (mit *Filedeskriptoren*, d.h. **`int`**)

`fork` (1)

Fork startet einen ***neuen Prozess*** als

<u>unabhängige Kopie</u> des Aufrufers

==> *<u>beide</u> kehren aus* `fork` *zurück!*

Unterscheidung *"Bin ich der Vater / der Sohn?"*
durch *<u>Returnwert</u>*:

`0` … im **Sohn**

`>0` (= *<u>Pid</u> des neuen <u>Sohnes</u>*) … im **Vater**

`-1` … im Vater, **Fehler** (*<u>kein</u>* Sohn erzeugt)

`fork` (2)

Unter Linux:

- **`fork`** ist ein Spezialfall von **`clone`**
- Implementierung mit *"Copy-on-Write"*

Typ `pid_t` = *"Process Id"* (eine *positive, ganze Zahl*)

- *Von* **`fork`**, **`getpid`**, **`getppid`**

  (im Vater beim **`fork`** Sohn-Pid *merken*,
  kann nicht anders ermittelt werden!)

- *Für* **`wait`**, **`waitpid`**, **`kill`**, ...

Ende von Prozessen (1)

2 Möglichkeiten:

- *"geordnet" mit Returncode* (**`exit`**, Ende von **`main`**)
- Durch ein *Signal*

Folgen:

- Vater bekommt **`SIGCHLD`**
  (wird aber defaultmäßig ***ignoriert***!)
- Prozess wird ***"Zombie"***
  (= nur *"defunct"*-Eintrag in Process Table, sonst *nichts* mehr!),
  bis der *Vater* seinen *Exitstatus* mit **`wait`** *abholt*
  (abschaltbar, dann verschwindet der Sohn spurlos)

Ende von Prozessen (2)

==> Nur der <u>Vater</u>

- … *<u>erfährt</u>* vom Tod eines Prozesses,
  kann mit **`wait`** auf ihn *<u>warten</u>*
- … kann den *<u>Exitstatus</u>* des Prozesses mit **`wait`**
  abrufen und dekodieren (mit **`WIF`**…-Makros):

  "Wie gestorben?" (*<u>Returncode</u>* oder *<u>Signalnummer</u>*)

Ein Sohn bekommt vom Tod des Vaters <u>nichts</u> mit!

Wenn der Vater *<u>vor</u>* dem Sohn stirbt: *<u>Sohn läuft weiter</u>!*
Prozess **`init`** (Pid **`1`**) *<u>wird neuer Vater</u>* für den verwaisten Sohn,
`init` räumt periodisch alle verwaisten Zombies weg

Warten auf Prozesse (1)

- **`wait`**:

  Auf *irgendeinen* Sohn (wer als nächstes stirbt)

  Immer blockierend (*hängt* bis einer stirbt)

- **`waitpid`**:

  Auf einen *bestimmten* Sohn / bestimmte Söhne

  Wahlweise blockierend
  oder *nichtblockierend* (zum Pollen)

Man kann für jeden Sohn nur *einmal*
den Exitstatus abrufen, nicht mehrmals!

Warten auf Prozesse (2)

Programmtechnisch 3 typische **`wait`**-Ansätze:

- Wenn der Vater *sonst nichts* (mehr) zu tun hat:

  Blockierendes **`wait`** / **`waitpid`**, bis Sohn stirbt

- **`wait`** / **`waitpid`** *"on demand"*
  (jedesmal genau wenn ein Sohn stirbt):

  In *Signal Handler* für **`SIGCHLD`**

- Periodisches *nichtblockierendes* **`waitpid`**:

  In einer Schleife: *"Ist in letzter Zeit wer gestorben?"*

exec*xxx* (1)

Startet ***im <u>aktuellen</u> Prozess***

ein ***neues Programm***

(Ausführung beginnt bei **`main`**)

==> Prozess bleibt *<u>gleich</u>*, es wird einiges *<u>vererbt</u>*!

Das neue Programm ***ersetzt*** das bisherige

==> **exec** kehrt *<u>nie zurück</u>* (außer bei Fehler)

==> *<u>Daten</u>* des bisherigen Programms sind *<u>weg</u>*

==> *<u>Alle Threads</u>* des bisherigen Programms sind *<u>weg</u>*

exec*xxx* (2)

exec startet das Binary *direkt* (ohne Shell)

==> *Keine* Shell-Scripts, keine Shell-Features,
kein Redirect, keine Wildcard-Auflösung, ...

==> Unter Linux werden *Skripts mit Shebang* **#!...**
richtig gestartet (portabel???)

==> Wenn nötig:
Man startet mit **exec** *explizit eine Shell*
und übergibt dieser Shell
das *auszuführende Programm als Argument*

exec*xxx* (3)

6 Aufrufs-Varianten:

- ***Argumente*** als *einzelne Parameter* (**l** wie "Liste") oder als *Array* (**v** wie "Vektor")

  (bei beidem: **NULL** am Ende!)

- Mit explizitem ***Environment*** (**e**)

  (sonst: Environment vererbt)

- Mit **PATH**-***Suche*** (**p**)

  (sonst: Programmname *mit vollem Pfad* angeben!)

exec*xxx* (4)

<u>*Erhalten*</u> bleiben u.a.:

- ***Offene Files*** (hängt von *"Close-on-Exec"*-Flag ab), aber nur Kernel-Fd's, keine **`FILE *`**

  Insbesondere **`stdin`** / **`stdout`** / **`stderr`**

- ***Signal***-Einstellung *"ignore"*, Signal-Maske, ... (aber keine Signal-Handler!)

- Der ***Alarm***-Timer!

pipe (1)

Eine ***Pipe*** ist eine

unidirektionale Datenverbindung
zwischen *gleichzeitig laufenden Prozessen*

Sie hat ***2 Enden*** (Schreib-Ende, Lese-Ende),
an *jedem* sitzt *ein Prozess* (selten mehrere)

Lesen & Schreiben funktioniert ***wie auf Files***,
aber die Daten werden *vom Kernel direkt kopiert*
(*ohne* Umweg über die Platte)

Achtung: Pipes sind ***gepuffert***! (meist 4 KB)

pipe (2)

Erzeugen mit System Call **`pipe`**:

Füllt ein Array mit ***2 File-Nummern***:
Lese-Ende & Schreib-Ende der neuen Pipe

Achtung: **`pipe`** liefert *Filedeskriptor*-**`int`**'s (OS-Ebene),
keine **`FILE*`** (nicht C-Ebene)

Daher danach meist:

- Entweder Zugriff über **`FILE *`** mit **`fdopen`**
- Oder umlegen auf **`stdin`** / **`stdout`** mit **`dup2`**

pipe (3)

`pipe` macht man fast nur *in Kombination mit* **`fork`** (meist *unmittelbar davor*):

Beide Prozesse sehen danach dieselbe Pipe
(*keine* Kopie!)

Schließen eines Endes normal mit **`close`** (wie File).

Ende-Verhalten siehe **popen**:

- **EOF** für Leser wenn *kein Schreiber* mehr
- **SIGPIPE** für Schreiber wenn *kein Leser* mehr

pipe (4)

Achtung:

`close` *der* *<u>unbenutzen Enden</u>* *nicht vergessen!*

(Leser schließt <u>sofort</u> sein Schreib-Ende,
Schreiber schließt <u>sofort</u> sein Lese-Ende)

Programm <u>hängt</u> sonst am Ende <u>ewig</u>!!!

(Leser bekommt <u>nie</u> **EOF***,*
solange noch jemand (auch er selbst!) das Schreibende offen hat!)

==> Siehe Grafik im alten HTL-Skript!

Eine Pipe *<u>verschwindet</u>* nach Benutzung *<u>von selbst</u>*
(sobald *<u>beide Enden geschlossen</u>* sind),
kein explizites “Wegräumen” nötig.

Shared Memory, Mutex, CAS ...

... siehe separater Vortrag!

Shared Mem & Sem unter Linux

3 Möglichkeiten:

- Shared Memory zwischen **Threads**
  - Speicher ist *implizit* geshared
    (normale globale Variablen & Heap)
  - Synchronisation siehe ***Pthreads-Library***
    (mehrere Synchronisations-Mechanismen)
- Shared Memory zwischen **Prozessen**:
  - Wahlweise alte Aufrufe: ***SysV IPC Shm & Sem***
  - … oder neuer Standard: ***POSIX Shm & Sem***

POSIX Shared Mem & Sem (1)

Unter POSIX:

2 verschiedene Varianten

betreffend **Anlegen / Verbinden / Löschen**
von Shared-Memory-Bereichen und Semaphoren:

- ***Anonym:***

  Nur *innerhalb eines Programms*
  (= zwischen Prozessen mit gemeinsamem Vater)

- ***Benannt:***

  Zwischen *unabhängigen Prozessen & Programmen*

POSIX Shared Mem & Sem (2)

Egal ob anonym oder benannt:

Der eigentliche Zugriff ist <u>gleich</u>

- **<u>Shared Memory</u>** … ist technisch ein *<u>Array von Bytes</u>*

  `mmap` liefert einen *<u>Pointer auf den Anfang des Arrays</u>*
  ==> auf den gewünschten *<u>Pointer-Typ casten</u>*
  und *<u>über diesen Pointer</u>* auf die Daten zugreifen

- **<u>Semaphore</u>** … ist eine *<u>Datenstruktur</u>* vom Typ **`sem_t`**
  (Innenleben unbekannt)

  "Lock" durch Aufruf von **`sem_wait`** usw.,
  "Unlock" durch Aufruf von **`sem_post`**

Anonyme Shared Mem & Sem (1)

Nur zwischen *Prozessen mit **demselben** Vater*:

- Vater legt *vor* dem **`fork`** Shared Mem & Sem an
- Alle Söhne *erben dasselbe Shared Mem* & Sem (brauchen nichts mehr tun)

Anonyme Shared Mem & Sem

- … haben *keinen Namen* o.ä.,
  sind *von außen nicht sichtbar* / zugreifbar
- … *verschwinden automatisch*,
  wenn das Programm endet

Anonyme Shared Mem & Sem (2)

Für Shared Mem:

Nur **mmap** (ohne Filedescriptor), *kein* **shm_open**

mmap legt an, legt die Größe fest, blendet ein

Für Sem:

Nur **sem_init**, *kein* **sem_open**
(und am Ende **sem_destroy**)

sem_init braucht einen *Pointer auf Speicher*
- der bereits *in einem Shared Mem* liegt
- der *groß genug* für eine **sem_t**-Variable ist

Named Shared Mem & Sem (1)

Für Shared Mem & Sem zwischen
unabhängig voneinander gestarteten Programmen

- *Explizites Anlegen* bzw. Zugreifen
  mit einem *Namen*
  (wie ein Filename, mit / am Anfang)
- Jeder, der den Namen kennt,
  kann darauf zugreifen
  (*Zugriffsrechte* mit User & Group wie bei Files)

Named Shared Mem & Sem (2)

Benannte Shared Mem & Sem *bleiben erhalten*
und können wiederverwendet werden

- bis das System rebootet wird
- oder bis sie *explizit gelöscht* werden

Auch *nach dem Ende* des anlegenden Programmes!
Auch wenn sie *kein Programm mehr* verwendet!

==> ***Anlegen & einblenden*** sind *2 getrennte Schritte*

==> ***Ausblenden & wirklich löschen***
sind auch *zwei getrennte Funktionen*

Named Shared Mem & Sem (3)

Anlegen: **shm_open** & **sem_open**

- **Entweder:**
  Neu anlegen
  Namen & Rechte festlegen
  Initialisieren
- **Oder:**
  Bestehendes mit diesem Namen *suchen*

Festlegen ob
"nur neu" / "nur bestehend" / "neu oder bestehend"
mit **O_CREAT** und **O_EXCL**

Named Shared Mem & Sem (4)

Bei Sem: Nur **`sem_open`**

Legt an / sucht & verbindet Programm damit

Bei Shared Mem: ***3 Schritte***

- **`shm_open`** *legt an* oder sucht bestehendes
- *Falls neu* angelegt: **`ftruncate`** legt *Größe* fest
- **`mmap`** *blendet* es im Programm *ein*

Nach **`mmap`** kann der Deskriptor von **`shm_open`**
sofort wieder mit **`close`** geschlossen werden

Named Shared Mem & Sem (5)

Ausblenden: **`munmap`** & **`sem_close`**

trennen die Verbindung (ohne etwas zu löschen!)
zwischen Shared Mem & Sem
und dem aufrufenden Programm

Wirklich löschen: **`shm_unlink`** & **`sem_unlink`**

entfernen den Namen (sofort)

vernichten das Shared Mem / die Sem
(erst *nachdem* sich das letzte Programm
davon getrennt hat)

POSIX Semaphore Problem (1)

Die alten *SysV IPC Semaphore*
hatten einen ***"Undo-Mechanismus"***:
Bei Programmende hat das Betriebssystem
ev. noch fehlende ***"unlock"*** *automatsich nachgeholt*.

POSIX Semaphore haben ***kein "Undo"!***

==> Bei einem *Absturz*, *externen Abbruch*, ...
während einer Kritischen Region
bleibt die Semaphore "ewig" gesperrt!!!

==> Defensiv programmieren (mit *Signal Handler*, ...)
oder *anderes Konstrukt* wählen (*z.B. File Locking*)

POSIX Semaphore Problem (2)

Die alten *SysV IPC Semaphore*
konnten *mehrere* Semaphore
in einem *einzigen* Aufruf
atomar belegen oder freigeben.

POSIX Semaphore können pro Aufruf
nur *eine* Operation auf *einer* Semaphore.

Wenn mit POSIX mehrere Belegungen nötig:

Gefahr von Deadlocks!

(systematisch belegen: In immer gleicher Reihenfolge!)