HOCHSCHULE FULDA
UNIVERSITY OF APPLIED SCIENCES

Betriebssysteme

Stage 4 – Prozesse & Threads

Was bisher geschah ...

- Geschichte der Rechner und Betriebssysteme

- Von-Neumann-Architektur
  - Moderne Hardware und deren Zusammenspiel
  - Mehrprogrammbetrieb
  - Das Betriebssystem

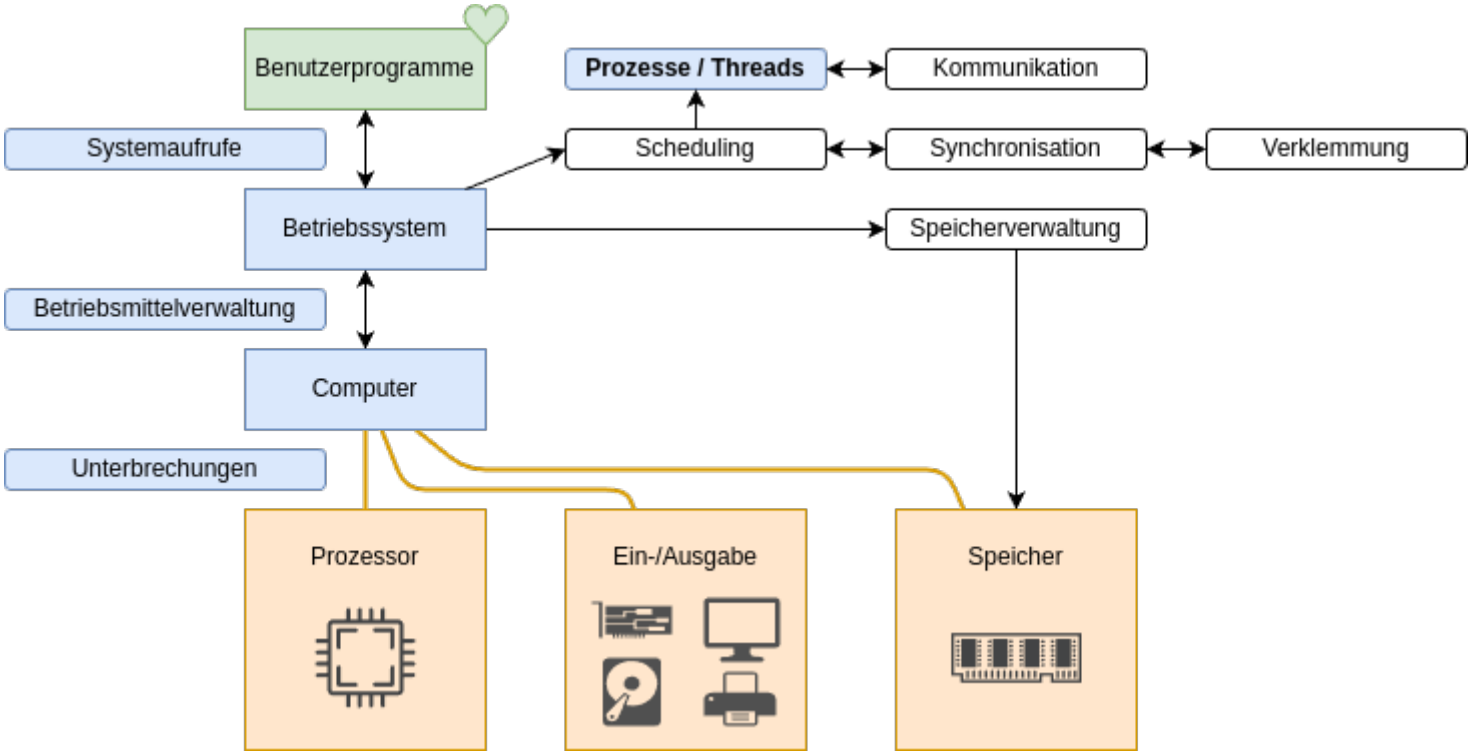
- Bootprozess und Sicherheitskonzept

- Interaktion mit dem Betriebssystem
  - Systemaufrufe (*system calls*)
  - Unterbrechungen (*interrupts*)

- Aufbau von Betriebssystemen

- Betriebssystemstrukturen
  - Monolithische Kerne (und Modulkonzept)
  - Mikrokerne (Spezialform Client-Server)
  - Mischform “Hybridkern” (Windows NT)

Übersicht

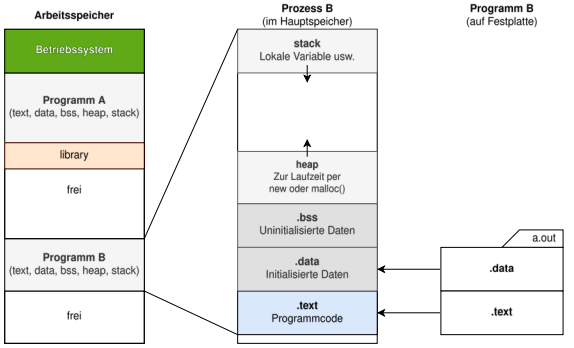

Benutzerprogramme
Systemaufrufe
Betriebssystem
Betriebsmittelverwaltung
Computer
Unterbrechungen
Prozesse / Threads
Kommunikation
Scheduling
Synchronisation
Verklemmung
Speicherverwaltung
Prozessor
Ein-/Ausgabe
Speicher

Themen und Lernziele

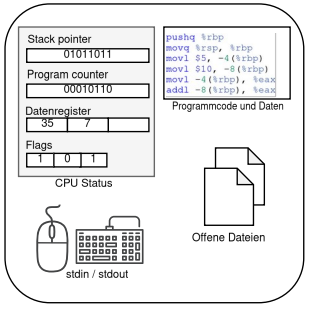
- Prozesse
  - Definition
  - Prozessmodell
  - Prozesszustände
  - Kontrollstrukturen
  - Prozesserzeugung
  - Beispiel
- Threads
  - “Threadmodell”
  - Implementierungen
  - Beispiel
- Vergleich

Definition: Sequentieller Prozess (Tanenbaum Kapitel 2.1)

- Programm
  - Statische Beschreibung eines sequentiellen Algorithmus (als Datei auf einer Festplatte, z.B. im ELF-Format)
  - Das selbe Programm kann mehrmals (auch gleichzeitig) ausgeführt werden
- Ein **Prozess** ist ein in Ausführung befindliches Programm
  - Einschließlich seiner aktuellen Werte des Programmzählers, CPU-Register, Speichervariablen, Stack, Referenzen auf geöffnete Dateien, usw.)
- Ein Prozess besitzt einen privaten Adressraum
  - Menge von Speicheradressen die vom Prozess nutzbar sind
  - Programmcode und Daten sind im Adressraum sichtbar



Programm und Prozess im Speicher



Prozess

Prozess und Prozessor

- Verhältnis Prozess – Prozessor
  - Prozess besitzt/benötigt konzeptionell einen eigenen virtuellen Prozessor
    (reale(r) Prozessor(en) werden zwischen den virtuellen Prozessoren umgeschaltet (Mehrprogrammbetrieb))
  - Umschalteinheit heißt Scheduler (ein Scheduling-Algorithmus legt Regeln fest)
  - Umschaltungsvorgang heißt Prozesswechsel oder Kontextwechsel (*context switch*)

- Abgrenzung Parallelität
  - Die Vorstellung des sequentiellen Prozesses ist unabhängig vom Konzept des Mehrprozessor- bzw. Multicore-System und der dort möglichen echt parallelen Ausführung von Prozessen

- Konsequenzen
  - Ausführungsgeschwindigkeit eines Prozesses ist nicht gleichmäßig und nicht reproduzierbar
  - Bei der Programmierung sind keine a-priori-Annahmen über den zeitlichen Verlauf zulässig
  - Bei zeitkritischen Anforderungen sind besondere Vorkehrungen im Scheduling-Algorithmus notwendig
    - z.B. Echtzeitsysteme

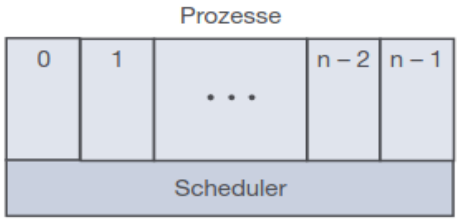
Prozessmodell

- Das Prozessmodell vereinfacht die Beschreibung der Aktivität des Rechensystems
  - Die ineinander verwobene Aktivität des Systems ist durch eine Menge von sequentiellen Prozessen beschreibbar

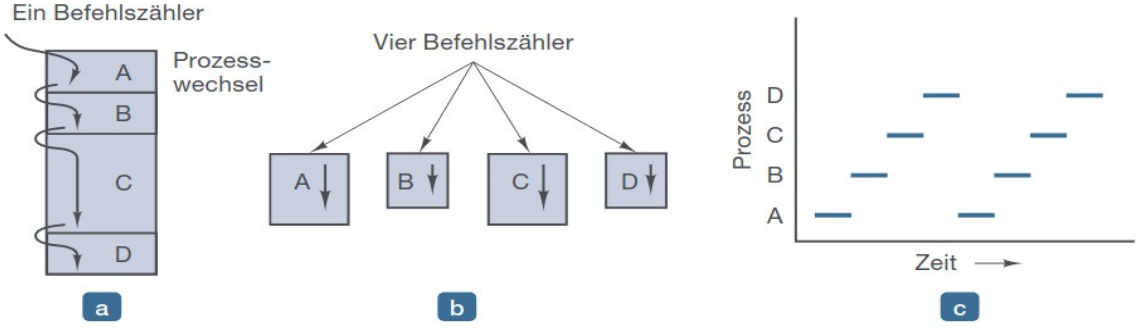
    → Unterste Schicht des Betriebssystems behandelt Unterbrechungen und ist für Scheduling verantwortlich

    → Der gesamte Rest des Systems besteht aus sequentiellen Prozessen

    (In der Realität ist diese strikte Trennung oft nicht ganz zutreffend)

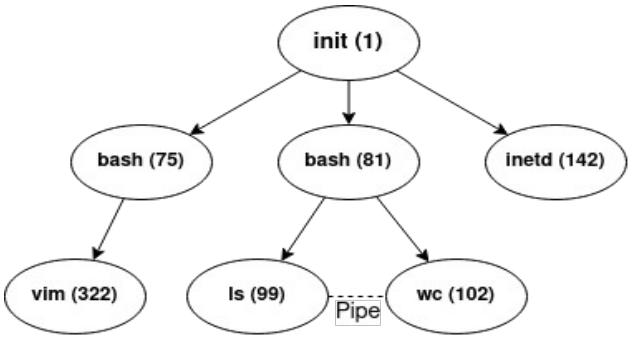


- Verschiedene Sichtweisen auf Prozesse
  - (a) Multiprogrammierung von vier Programmen im Speicher
  - (b) Konzeptionelles Modell von vier individuellen, sequenziellen Prozessen
  - (c) Zu jedem Zeitpunkt ist immer nur ein Programm aktiv



Prozesserzeugung

- Betriebssystem benötigt einen Mechanismus zur Erzeugung von Prozessen
  - In einfachen Systemen (z.B. Gerätesteuerung)
    - Menge der zur Laufzeit existierenden Prozesse ist häufig statisch (bei Systemstart erzeugt)
  - **Allgemeiner Fall: Prozesse werden dynamisch (zur Laufzeit) durch andere Prozesse erzeugt**
    - Erzeugender Prozess heißt Eltern-Prozess
    - Erzeugte Prozesse heißen Kind-Prozess(e)
- Wiederholte Prozesserzeugung durch Kindprozesse
  - es entsteht eine baumartig strukturierte Prozessmenge
- Prozessfamilie
  - Prozess mit all seinen direkten und indirekten Nachfahren



Prozesszustände

Prozesszustände

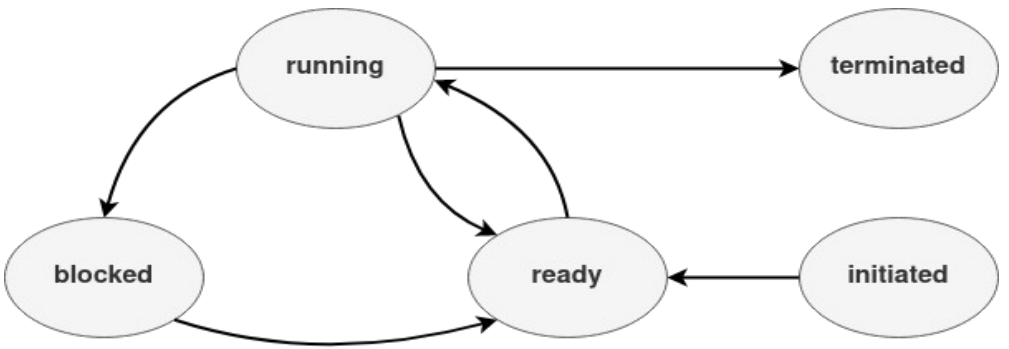
- Prozesse können sich in unterschiedlichen Zuständen befinden
  - Er kann z.B. aktiv auf der CPU ausgeführt werden, I/O durchführen, oder auch warten
  - Obwohl unabhängig, können diese logisch voneinander abhängig sein (Beispiel UNIX Pipes)
    → cat somefile | grep “bob”

- Prozess muss ggf. auf eine Eingabe warten
  - Prozess blockiert und wartet auf ein (für ihn externes) Ereignis
  - Abhängig von den relativen Ausführungsgeschwindigkeiten
  - Konsequenz auf Einprozessorsystem?
    - Prozessor wird unmittelbar einem anderen Prozess zugeordnet
    - Entzug des Prozessors (Suspendierung) in diesem Fall problembegründet

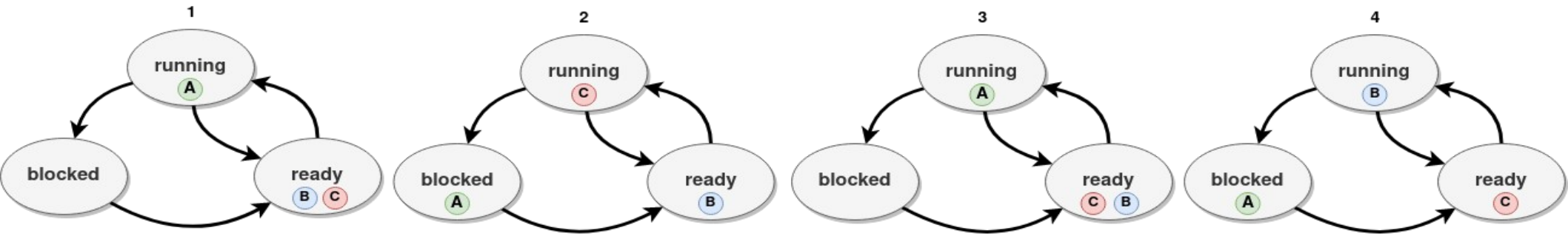
- Preemption
  - Scheduler entscheidet auf Prozesswechsel
  - Kann geschehen, obwohl der Prozess selbst weiter ausgeführt werden könnte/möchte

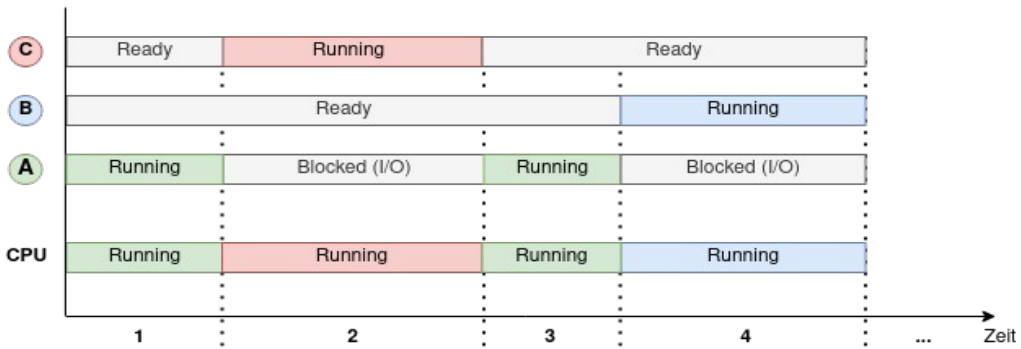
Prozessplanung und -zustände

- **Kurzfristig**
  - **running (aktiv, rechnend)**
    Dem Prozess ist ein Prozessor zugeordnet, der das Programm vorantreibt
  - **ready (bereit, rechenwillig):**
    Prozess ist ausführbar, aber Prozessor ist anderem Prozess zugeordnet (bzw. alle verfügbaren Prozessoren sind anderen Prozessen zugeordnet)
  - **blocked (blockiert, schlafend)**
    Der Prozess wartet auf ein Ereignis. Er kann solange nicht ausgeführt werden, bis dieses Ereignis eintritt
- **Mittelfristig**
  - **Ausgelagert bereit / -blockiert**
    (nicht Thema der Veranstaltung)
- **Langfristig**
  - **initiated (neu, initiiert)**
    In Vorbereitung (Anfangszustand)
  - **terminated (beendet, terminiert)**
    Prozess ist beendet (Endzustand)



Beispiel: Prozesszustände





Zustandsübergänge

rechnend → blockiert
(Versetzung in den Wartezustand, Warten auf Ereignis)

rechnend → rechenwillig
(Scheduler entzieht den Prozessor)

rechenwillig → rechnend
(Scheduler teilt Prozessor zu)

blockiert → rechenwillig
(Ereignis tritt ein)

Prozesse: Spezielle Prozessezustände

Verwaiste Prozesse

- Ein UNIX-Prozess wird zum Waisenkind, wenn sein Elternprozess terminiert
- Der Prozess an der Wurzel adoptiert alle verwaisten Prozesse. So bleibt die Prozesshierarchie intakt
  - (System V: init, GNU/Linux heute oft systemd)

Zombie Prozesse

- Ein UNIX-Prozess wird zum Zombie, wenn er bereits beendet wurde, sein PCB aber noch existiert
- Situation
  - Kernel sendet SIGCHLD an den Elternprozess, sobald ein Kind endet
  - Elternprozess kann Art der Beendigung erfragren: erfolgreich, mit Fehler, usw.
  - Eintrag des Prozesses verbleibt in der Prozesstabelle bis Status abgefragt wurde
  - In dieser Zeit hat der Prozess den Status Zombie
  - Spezialfall Linux-Kernel:

    Falls der Elternprozess explizit SIGCHLD ignoriert, werden die Einträge sofort gelöscht

Kontrollstrukturen

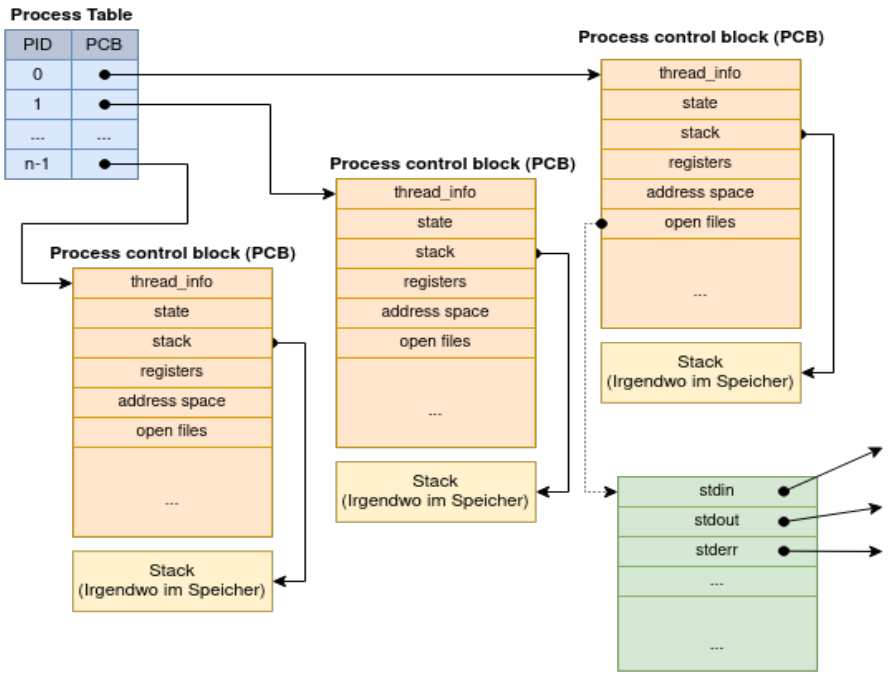
Prozesstabelle und Prozesskontrollblock

- Für jeden Prozess pflegt das Betriebssystem einen Process Control Block (PCB)
  - Enthält Sicherung der Register und Referenz auf den Stack
  - Enthält Referenzen auf Dateideskriptoren

- Die Prozesstabelle enthält Referenzen auf die PCB aller Prozesse im System

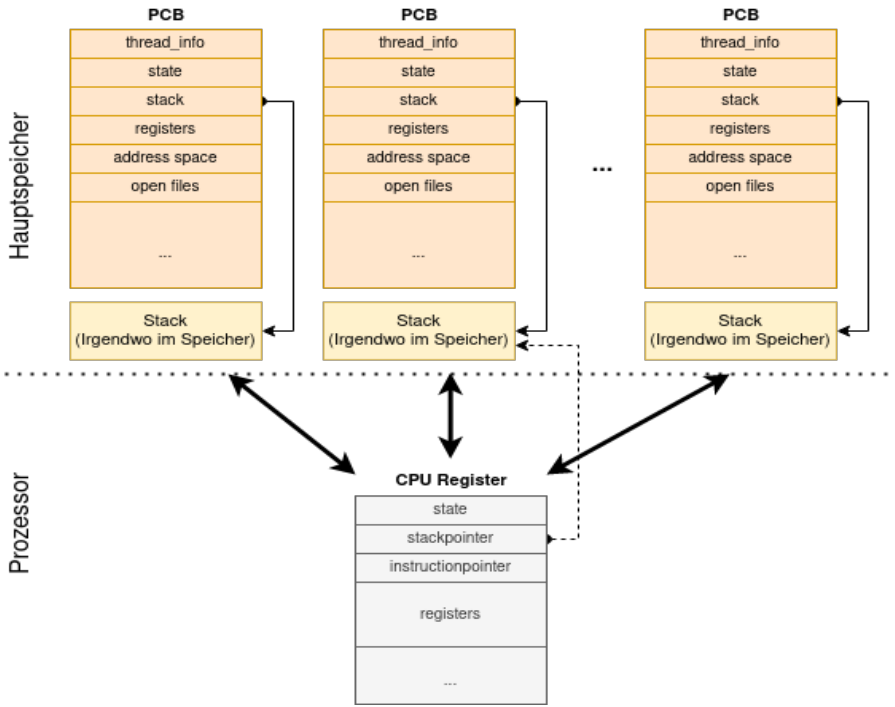
- Typische Felder im PCB

Prozessverwaltung	Speicherverwaltung	Dateisystem
Register	Zeiger auf Textsegment	Wurzelverzeichnis
Programmzähler	Zeiger auf Datensegment	aktuelles Verzeichnis
Programmstatuswort	Zeiger auf BSS-Segment	UMASK-Maske
Stack-Zeiger	Prozessgruppe	offene Dateideskriptoren
Prozesszustand	reale UID	effektive UID
Prozessnummer	effektive UID	effektive GID
Vaterprozessnummer	reale GID	Aufrufparameter
Prozesserzeugungszeitpunkt	effektive GID	verschiedene Flags
Terminierungsstatus	verschiedene Flags	
verbrauchte Prozessorzeit		
Prozessorzeit der Kinder		
Alarm-Zeitpunkt		
Signalstatus		
Signalmaske		
unbearbeitete Signale		
Zeiger auf Nachrichten		
verschiedene Flags		



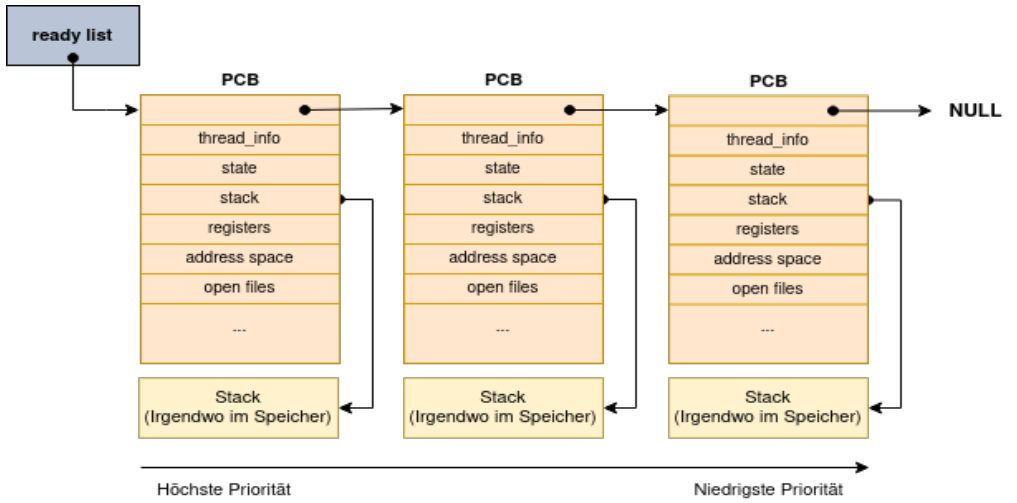
Prozesswechsel

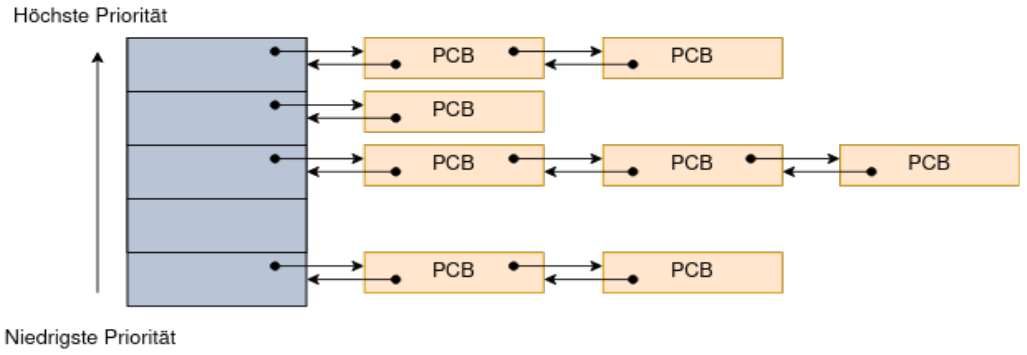
- Wechsel des Prozess-Kontextes
  - z.B. während der Interruptbehandlung
- Typische Ausführungsschritte
  - Programmzähler (und weitere Register) werden durch die Hardware auf dem Stack abgelegt
  - Neuer Programmzählerinhalt aus wird aus dem Unterbrechungsvektor ermittelt und geladen
  - Eine Assembler-Routine rettet die Registerinhalte des alten Prozesses
  - Eine Assembler-Routine bereitet den neuen Stack vor
  - Markierung des unterbrochenen Prozesses als rechenwillig
  - Der Scheduler bestimmt den Prozess, der als nächster ausgeführt werden soll
  - Eine Assembler-Routine läd den Kontext (Registerinhalte) des ausgewählten Prozesses und startet diesen



Prozess-Warteschlange

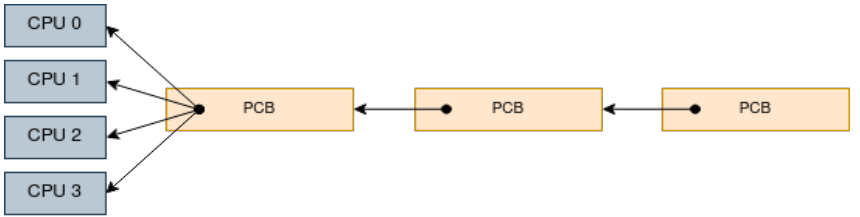
- Ready-List
  → Enthält alle laufbereiten Prozesse
- Implementierung
  - Zum Beispiel als einfache FIFO-List (ggf. sogar mit Prioritäten)
  - Mehrere Listen, die Prozesse gleicher Priorität enthalten
- Pflege weitere Listen
  - z.B. Alle Prozesse die auf ein bestimmtes Ereignis warten



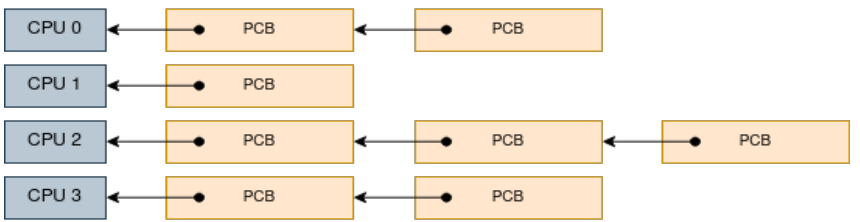


Prozess-Warteschlange

- Gemeinsame Warteschlange für alle CPUs



- Eigene Warteschlange pro CPU



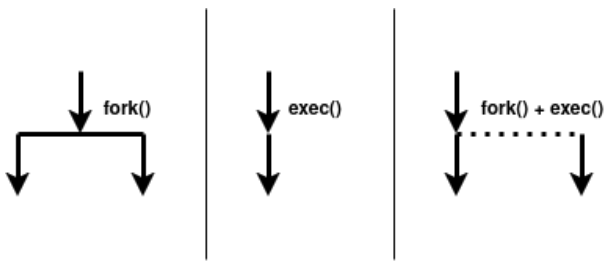
Stichwort NUMA (Non-Uniform Memory Access)

- Besitzt ein Rechner mehrere physische CPUs, so ist i.d.R. jeweils ein Teil seines Hauptspeichers an jeweils eine der CPUs angebunden. Durch die andere CPU zwar auch adressierbar, jedoch mit Geschwindigkeitseinbußen!

Prozesserzeugung

UNIX-Prozesssteuerung

- Für die Erzeugung eines neuen Prozesses steht der Systemaufruf fork zur Verfügung
  - Erzeugung einer exakten Kopie des aufrufenden Prozesses
  - Standardisiert in POSIX.1-2024 (Historisch: POSIX.1-2001, SystemV R4, 4.3BSD)
- Für die Ausführung eines anderen Programms steht der Systemaufruf execve zur Verfügung
  - Ersetzen des aktuellen Programmcode, sowie dessen Stack, Heap und Datensegmente
  - Standardisiert in POSIX.1-2024 (Historisch: POSIX.1-2001, SystemV R4, 4.3BSD)



UNIX-Prozesssteuerung

- **pid_t fork (void)**
  - Erzeugen einer Kopie des Prozesses.
    Elternprozess erhält pid des Kindprozesses oder -1 bei Fehler, Kind erhält 0 als Ergebnis
- **exit (status)**
  - Beenden des laufenden Prozesses und zur Verfügung stellen des Exit-Status an den Elternprozess
- **pid_t wait (int* status)**
  - Warten auf die Beendigung eines Kindprozesses.
    Dessen ID wird über den Rückgabewert, sein Status über status zurückgegeben
- **pid_t waitpid (pid_t pid, int* status, int opts)**
  - Warten auf das Ende eines bestimmten Kindprozesses
    Dessen ID über den Parameter pid übergeben wird
- **int execve (char* name, char* argv[], char* envp[])**
  - Überlagern des in einem Prozess ausgeführten Programms (Code, Daten, Stack) durch ein neues Programm
  - Andere Varianten: execl, execle, execlp, execv, execvp
- **pid_t getpid (void)**
  - Rückgabe der eigenen Prozess-ID
- **pid_t getppid (void)**
  - Rückgabe der Prozess-ID des Elternprozesses

Beispiel: Prozesserzeugung

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

int main (int argc, char **argv) {
    int pid;

    printf ("Aktuelle Daten: SID: %d, PID: %d, PPID: %d\n",
            getsid (0), getpid (), getppid ());

    pid = fork ();

    if (pid > 0) {
        printf ("Hallo vom Elternprozess, Kind-PID: %d\n", pid);

    } else if (pid == 0) {
        printf ("Hallo vom Kindprozess, PID: %d, PPID: %d\n",
                getpid (), getppid ());

    } else {
        printf ("Es ist ein Fehler aufgetreten!\n");
        exit (EXIT_FAILURE);
    }

    exit (EXIT_SUCCESS);
}
```

stdlib.h → exit() und EXIT_SUCCESS
stdio.h → printf()
unistd.h → getpid(), getppid(), getsid()

Erzeugung einer Kopie des aktuellen Prozesses

Return liefert die ID des Kindprozesses
an den Elternprozess und 0 an den Kindprozess

Ab hier laufen 2 identische Prozesse!

Dieser Teil wird vom **Elternprozess** ausgeführt
(return von `fork() > 0`)

Dieser Teil wird vom **Kindprozess** ausgeführt
(return von `fork() == 0`)

Dieser Teil wird im **Fehlerfall** ausgeführt
(return von `fork() == -1`)

```
fd3430@pc01:~$ ./fork
Hallo vom aktuellen Prozess: PID: 114750, PPID: 44943
Hallo vom Elternprozess, Kind-PID: 114751
Hallo vom Kindprozess, PID: 114751, PPID: 114750
```

Prozesserzeugung: Implementierung

- Der Aufruf fork() ist *kostspielig*
  - Der gesamte Adressraums des Prozesses muss kopiert werden
  - Verschwendung, falls unmittelbar danach z.B. exec..() aufgerufen wird

- Historische Lösung: **vfork**
  - Der Elternprozess wird suspendiert, bis der Kindprozess exec..() aufruft oder terminiert
  - Der Kindprozess nutzt den Code und die Daten des Elternprozesses, allerdings darf er diese nicht verändern (→ _exit() statt exit(), sofern er noch im Adressraum des Elternprozess tätig ist)

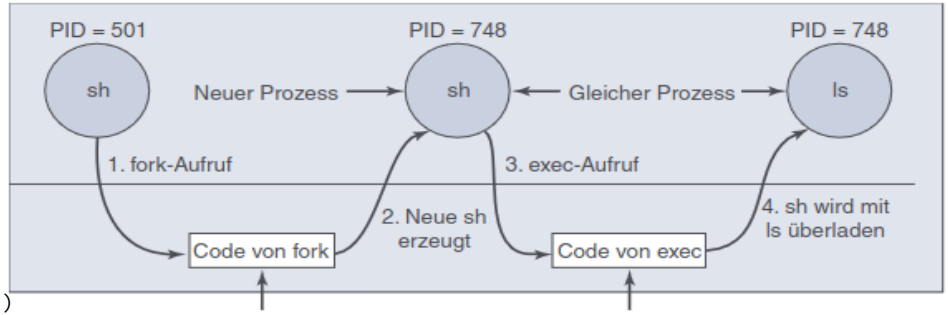
- Heutige Lösung: **copy-on-write**
  - **Der Prozess Control Block (PCB) wird kopiert**
  - Eltern- und Kindprozess teilen sich aber das selbe Code- und Datensegment im Speicher
  - Erst wenn der Kindprozess Daten verändert, wird das entsprechende Segment kopiert (was nicht passiert, wenn direkt exec..() aufgerufen wird)

Starten eines neuen Prozesses / Programms

- Mittels fork-exec
  (erst fork(), dann exec..())
  - Vorteil der Trennung: Nach dem fork() und vor dem exec..() können Verwaltungsarbeiten durchgeführt werden (z.B. Pipe erzeugen)
  - Durchaus auch Kritik an diesem “nicht mehr zeitgemäße” Konzept
    (https://www.microsoft.com/en-us/research/uploads/prod/2019/04/fork-hotos19.pdf)

- Alternative posix_spawn()
  - POSIX bietet eine Alternative zu fork-exec, die in Linux genutzt werden kann, jedoch nicht verbreitet ist
  - Nutzung des clone system call mit den Flags CLONE_VM und CLONE_VFORK

Tanenbaum, Fig. 10.8: Achritte bei Ausführung des Kommandos *ls*



Task-Struktur für Kindprozess belegen
Task-Struktur des Kindprozesses mit der des Elternprozesses füllen
Stack und Benutzerbereich von Kindprozess erzeugen
Benutzerbereich von Elternprozess kopieren
PID für Kindprozess reservieren
Kind- und Elternprozess teilen sich Textsegment
Seitentabellen für Stack und Daten kopieren
Geöffnete Dateien werden gemeinsam benutzt
Register des Elternprozesses zu Kind kopieren

Ausführbares Programm finden
Ausführungsrechte prüfen
Header lesen und prüfen
Argumente und Umgebung in Kern kopieren
Alten Adressraum freigeben
Neuen Adressraum belegen
Argumente und Umgebung auf Stack kopieren
Signale zurücksetzen
Register initialisieren

Systemaufruf: Clone

Nur informativ

```
int __clone (int (*fn) (void *arg), void *child_stack, int flags, void *args);
```

- Flags
  - CLONE_VM – share memory
  - CLONE_FILES – share file descriptors
  - CLONE_SIGHAND – share signal handlers
  - CLONE_FS – share filesystem
  - CLONE_VFORK – allow child to signal parent on exit
  - …

- Sehr flexibler system call
  - Erlaubt die Erzeugung von Prozessen (mittels fork und posix_spawn) und von POSIX Threads ...
    ... und allerhand Linux-spezifischer Dinge
    durch die Verwendung unterschiedlicher Kombinationen von Flags

Gewicht von Prozessen

- Das “Gewicht” als bildlicher Ausdruck für die Größe des Kontexts eines Prozesses
  - Prozesserzeugung, -umschaltung und -kommunikation sind teuer
    (rechenzeitaufwendig zur Laufzeit und zusätzlich Verluste durch Cache-Misses)

- Diverse Nachteile
  - Nutzung mehrerer Prozessoren für eine Applikation erfordert mehrere kooperierende Prozesse
  - Strukturierung eines Server-Prozesses, der Anforderungen von mehreren Klienten bedienen kann erfordert Multiplexing für verschiedene Klienten

    → muss vom Applikationsprogrammierer ausprogrammiert werden

- Lösung
  - Leichtgewichtige Prozesse
  - Federgewichtige Prozesse

Threads

Threads (Tanenbaum Kapitel 2.2)

- Threads (Aktivitätsträger)
  - Idee einer parallel ausgeführten Programmfunktion

- Threads
  - besitzen einen eigenen Prozessor-Context (Registerinhalte usw.)
  - besitzen einen eigenen Stack (i.d.R. zwei, getrennt für user und kernel mode)
  - besitzen einen eigenen kleinen privaten Datenbereich (Thread Local Storage)
  - nutzen gemeinsam Programm, Adressraum und alle Betriebsmittel (offene Dateien, Signale, usw.)

- Vorteile / Nachteile
  - + Aufwändige Operationen können in einen leichtgewichtigen Hilfsprozess ausgelagert werden, während der Elternprozess erneut auf Eingabe reagieren kann. (Beispiel: Webserver)
  - + Programme, die aus mehreren unabhängigen Kontrollflüssen bestehen, profitieren unmittelbar von Multiprozessor-Hardware
  - - Programmierung ist schwierig: Zugriff auf gemeinsame Daten muss koordiniert werden.

Anwendungsbeispiel Threads

- Mehrere Threads arbeiten auf den gleichen Betriebsmitteln (Adressraum, Dateien, Tastatur, Bildschirm)

- Beispiel für eine Lösung mittels Threads: Textverarbeitungsprogramm
  - Ein Thread behandelt Nutzereingaben
  - Ein Thread ist für das periodische Speichern zuständig
  - Ein Thread erledigt eine Formatierungsaufgabe im Hintergrund
  - Ein Thread führt periodisch eine Rechtschreibprüfung durch und markiert fehlerhaften Text

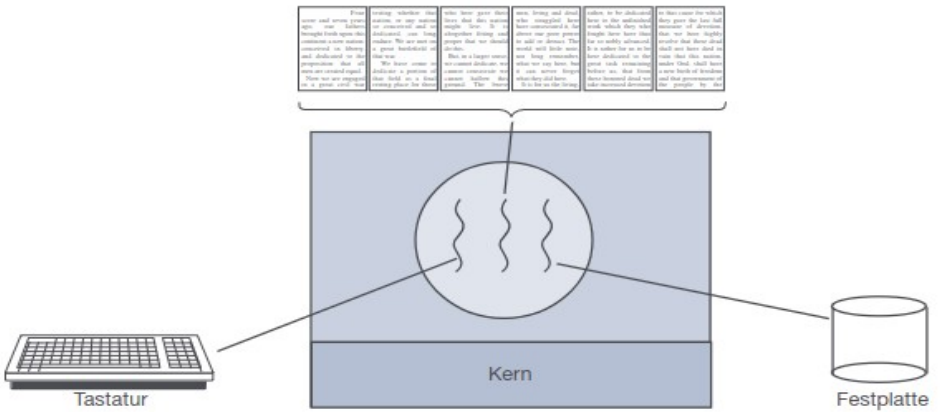
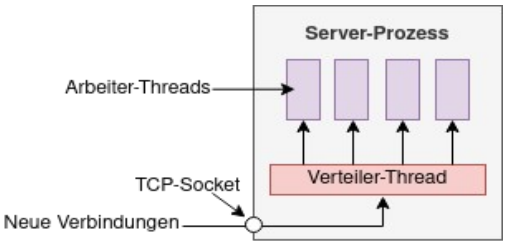


Abbildung 2.7: Ein Textverarbeitungsprogramm mit drei Threads

Kooperationsformen

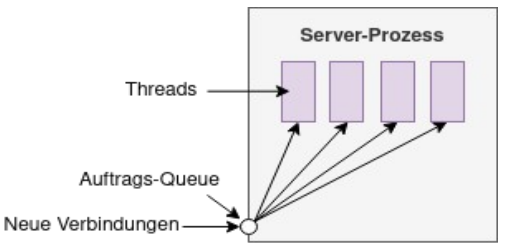
Dispatcher-Worker (Verteiler-Arbeiter) Model

- Ein Verteiler-Thread nimmt neue Aufträge entgegen und übergiebt diese sofort zur Bearbeitung an die Arbeiter.
- Im Beispiel eines HTTP-Server kann z.B. jeder Arbeiter-Thread jeweils einen Clienten bedienen



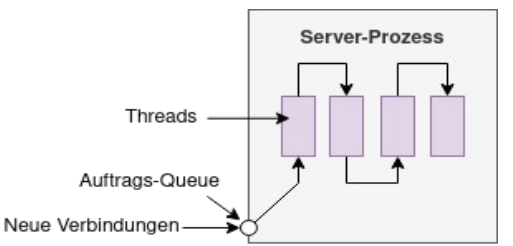
Team Model

- Threads arbeiten unabhängig voneinander (keine Verteiler-Arbeiter Beziehung)
- Jeder Thread kann z.B. eigenständige Tätigkeiten erledigen, die nichts mit den anderen Threads zutun haben



Pipeline (Fließband) Model

- Die Ausgabe eines Threads wird als Eingabe an den nächsten Thread übergeben (Erzeuger-Verbraucher Modell)
- Zum Beispiel mehrere aufeinanderfolgende CPU-intensive Arbeiten, die somit parallelisiert werden kännen



Implementierung (Tanenbaum Kapitel 2.2.4 - 2.2.6)

- Vollständig im Benutzer-Adressraum (User-Level-Threads → Federgewichtige Threads)
  - Betriebssystem weiß nichts davon, dass mehrere Threads existieren
  - Eigene Verwaltungsdatenstruktur innerhalb des jeweiligen Prozesses benötigt
  - Threads laufen so lange abwechselnd, bis dem Prozess die CPU entzogen wird
  - Hat Vorteile (i.d.R. gute Performance) aber auch viele Nachteile und Probleme

- Im Kernel (Kernel-Level-Threads)
  - Verwaltungsdatenstruktur für Threads im Kernel (genau wie Prozess-Tabelle)
  - Blockiert ein Thread kann der Kernel den nächsten Thread (aus dem gleichen oder einem anderen) Prozess auswählen
  - Höhere Kosten (größeres Gewicht) bei Kontextwechseln
  - Auch hier verschiedenste Probleme zu lösen
    - z.B., was passiert mit den Threads eines Prozesses bei fork()?

- Auch hybrider Ansatz ist möglich

Threads: Schnittstellen

- Keine einheitliche bzw. übergreifende Schnittstelle
  - Beispiel: Java
    - Klasse Thread in java.lang
    - z.B. Implementierung der Schnittstelle java.lang.Runnable und Implementierung von Methode run() Thread-Modell in JVM implementiert
  - Beispiel: C++
    - Boost Threads verbreitet
  - Beispiel: Windows
    - C-Schnittstelle für Windows API, u.a. CreateThread(...)
  - Beispiel: LinuxThreads (nicht portabel)
    - Gehört nicht zum Unix Standard

Threads: Schnittstellen

- POSIX Threads (Pthreads)
  - Verbreitete Standard-Programmierschnittstelle für Threading und entsprechende Synchronisation
  - Standardisiert in IEEE POSIX 1003.1c (1995)
  - Pthreads: Implementierungen dieses Standards

- Verbreitung
  - Auf vielen Systemen, insbesondere auch Multiprozessorsystemen (z.B. Linux, Solaris, MacOS X, FreeBSD, (Windows NT bis XP/2003))
  - Neben C / C++ auch für andere Programmiersprachen: z.B. Fortran
  - Teilweise Bestandteil der libc

POSIX Threads

Nur informativ

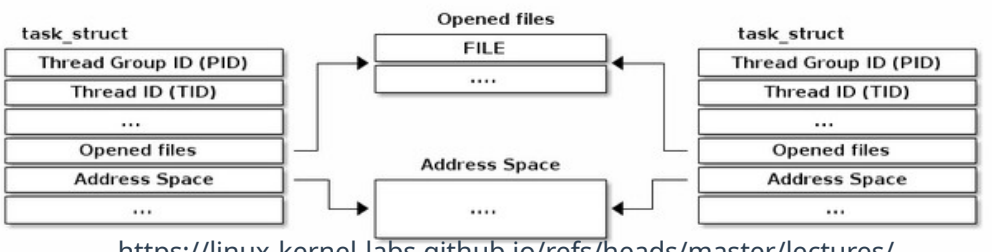
- **`int pthread_create (pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg);`**
  - Erzeugen eines Threads
- **`void pthread_exit (void *retval);`**
  - Sich selbst beenden
- **`pthread_t pthread_self (void);`**
  - Thread-Identifier des aktuellen Threads ermitteln
- **`int pthread_join (pthread_t th, void **thread_return);`**
  - Warten auf Beendigung eines Threads
- **`int pthread_cancel (pthread_t thread);`**
  - Beenden eines anderen Threads
- **`int sched_yield (void);`**
  - Den Prozessor abgeben

Beispiel: Linux (ab 2.6)

- Native POSIX Thread Library (NPTL)
  - Nutzung zusammen mit GNU libc (glibc)
  - 1:1 Implementierung → 1:1-Beziehung der Threads mit Prozessen in den Scheduler-Queues des Kernels

- Konzept
  - Kernel verwaltet “Tasks” (Prozesse und Threads)
    Datenstruktur struct task_struct (siehe: https://github.com/torvalds/linux/blob/master/include/linux/sched.h)
  - pthread_create() führt zu neuem “Prozess” unter Nutzung von clone()
    - Systemcall clone() erlaubt sehr flexible Steuerung der zu kopierenden Prozessresourcen (Addressraum, Offene Dateien, Signalbehandlung, usw.)

- Ziele
  - Konformität zu POSIX Pthreads
  - Gute Multiprozessor-Performance
  - Niedrige Erzeugungskosten
  - Kompatibilität zu LinuxThreads



https://linux-kernel-labs.github.io/refs/heads/master/lectures/processes.html

Beispiel: Threaderzeugung (Minimalbeispiel)

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


void* print_hello (void*) {
    pthread_t tid = pthread_self ();
    printf ("Hello from thread %lu\n", tid);
    pthread_exit(EXIT_SUCCESS);
}

int main (int argc, char** argv) {
    pthread_t thread1, thread2;
    int ret1, ret2;

    ret1 = pthread_create (&thread1, NULL, print_hello, NULL);
    ret2 = pthread_create (&thread2, NULL, print_hello, NULL);

    pthread_join (thread1, NULL);
    pthread_join (thread2, NULL);

    exit (EXIT_SUCCESS);
}
```

Funktion, die von den Threads ausgeführt wird

Eigene Thread-ID ermitteln

Thread nach erledigter Arbeit beenden

2 Threads erzeugen

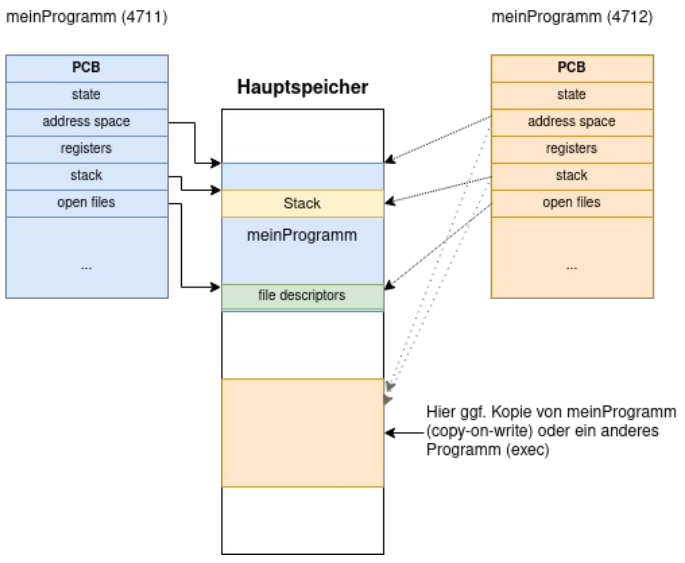
Auf deren Ende warten

Vergleich

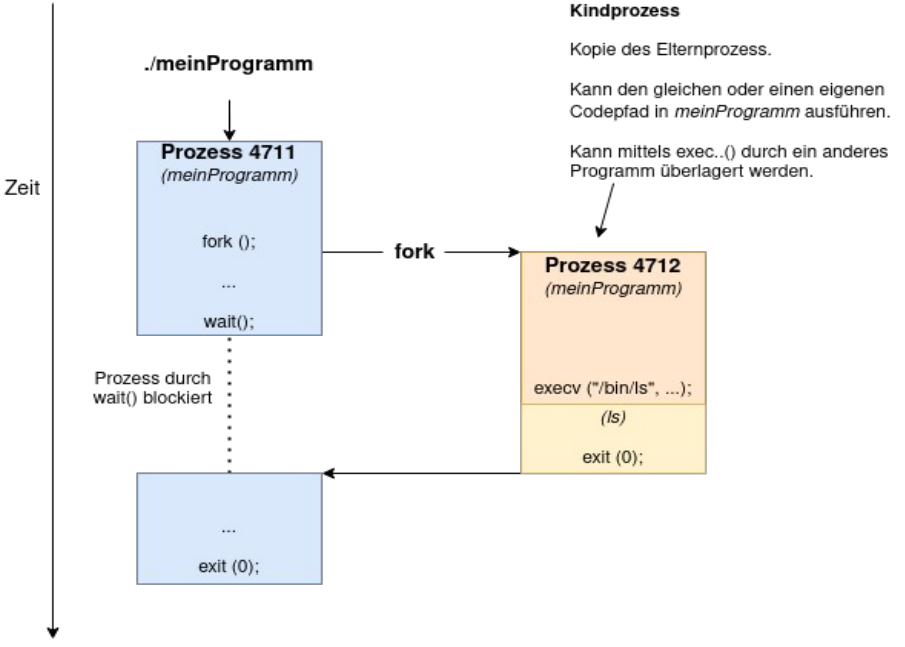
Beispiel: Prozesse

- Neuer PCB, der zunächst auf gleiche Datenstrukturen im Speicher zeigt
  - Durch Aufruf von exec, oder schreibenden Zugriff auf Programmdaten entsteht eine Kopie (neuer Speicherbereich) und die Referenzen des PCB werden entsprechend angepasst.

PCB und Speicherlayout



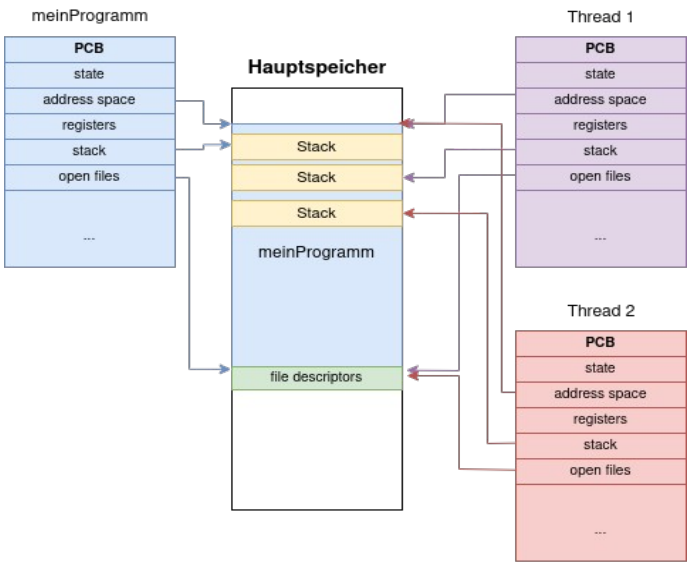
Programmfluss



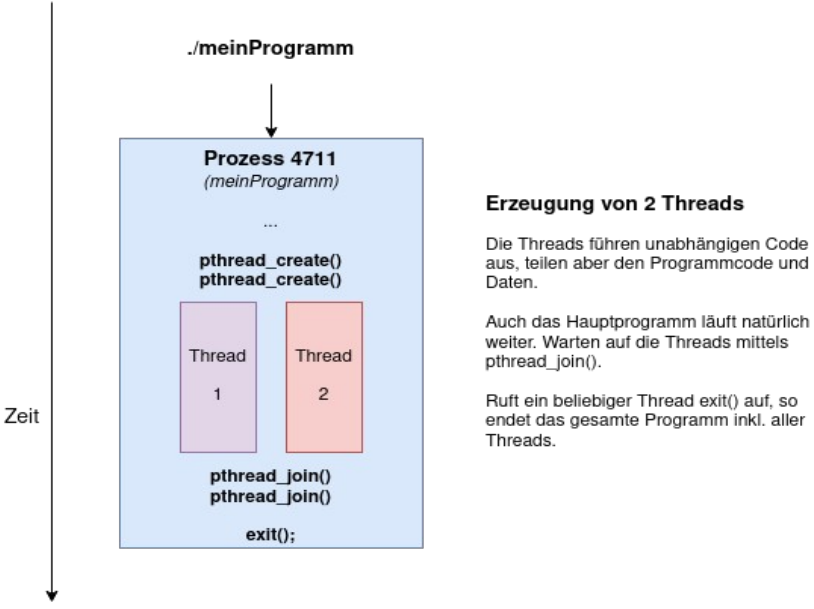
Beispiel: Threads

- Neuer PCB, der dauerhaft auf die gleiche Datenstrukturen im Speicher zeigt
  - Jeder Thread besitzt einen eigenen Stack und einen eigenen CPU-Kontext

PCB und Speicherlayout



Programmfluss



Erzeugung von 2 Threads

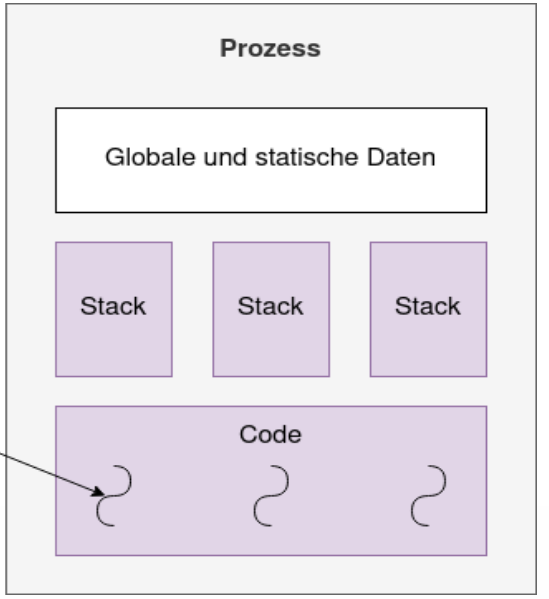
Die Threads führen unabhängigen Code aus, teilen aber den Programmcode und Daten.

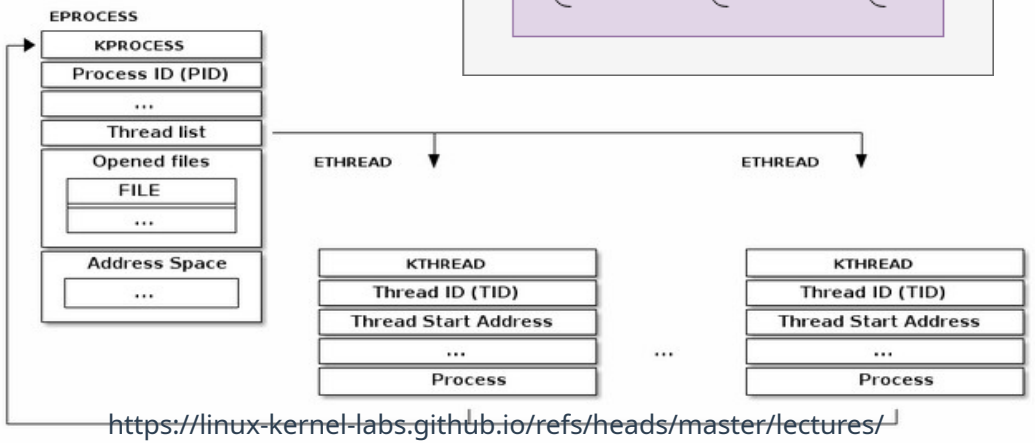
Auch das Hauptprogramm läuft natürlich weiter. Warten auf die Threads mittels pthread_join().

Ruft ein beliebiger Thread exit() auf, so endet das gesamte Programm inkl. aller Threads.

Beispiel: Windows

- Konzept
  - Prozess → Laufzeitumgebung für Threads
    - jeder Prozess besitzt mindestens einen Thread
  - Thread(s) → Eigentliche Code-Ausführungseinheit(en)
    - Jeder Thread besitzt eigene Register und Stack
- Eigenschaften
  - Threadmodell seit Windows NT integraler Bestandteil
  - Scheduler verwaltet Threads
    → Weist ihnen Rechenzeit zu
  - Alle Threads sind Kernel-Threads
  - User-Threads sind möglich, aber unüblich





https://linux-kernel-labs.github.io/refs/heads/master/lectures/processes.html

Zusammenfassung

- Prozesse
- Threads

Nächste Woche: Straight forward oder deep dive? ;)