Einführung in die Betriebssysteme

Martin Spörl

Multitasking

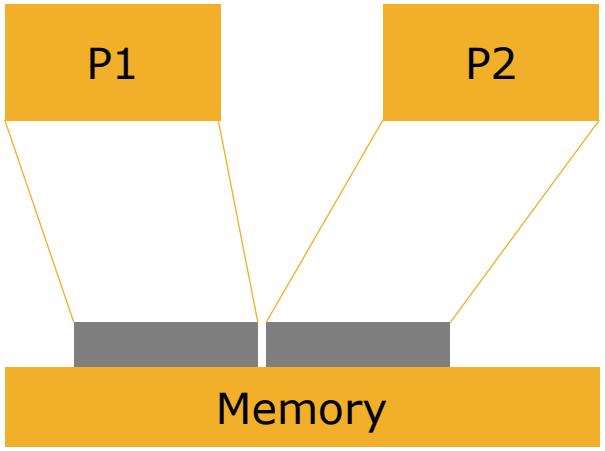
Grundlagen

- Definition & Grundlagen
    - Grundidee = maximale Auslastung des Prozessors
    - Problem:
        - Prozessor sehr schnell
        - Externer Input / externe Ereignisse sehr langsam
        - Ergebnis: Prozessor muss häufig warten
    - Leerlauf des Prozessors soll verhindert werden

Begriffsdefinition

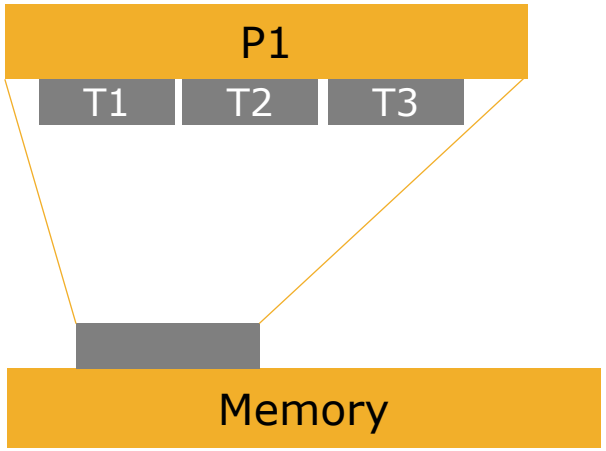
Multitasking

- Nebenläufige Prozesse
- pro Prozess eigener Adressraum & Umgebung
- OS Verwaltet Prozesse & deren Umschaltung



Multithreading

- Nebenläufige Befehlsströme („Programmfäden“) == Threads
- Alle in gleichem Adressraum & Umgebung
- Prozess verwaltet Threads



Hyper-Threading

- von Intel entwickelt
- Nutzung von „Leerlauf“ in Prozessor
- Instruktionen werden parallel geladen
  - Nicht genutzte Teile (z.B. Alu) können so für 2ten Befehl bereits genutzt werden
- OS sieht 2ten logischen Prozessor als 2ten physikalischen Prozessor
- theoretisch 2 fache Geschwindigkeit

Forking I - Begriffsdefinition

Abspaltung von Projekten

- häufig in Open Source Szene genutzt
- Alternative Fortführung von Entwicklung
- Bekannte Beispiele:
  - Libav ist ein Fork von ffmpeg
  - Nextcloud ist Fork von ownCloud
- Gründe
  - Differenzen in Entwicklungsteams
  - Firmenaufspaltungen
  - Unterschiedliche Ziele der einzelnen Entwicklungen
  - Weiterentwicklung von eingestellten Projekten

Abspaltung von Prozessen

- Prozess der von einem anderen Prozess erstellt wurde
  - Erstellter Prozess = Kind-Prozess („Child Process“)
  - Erzeugender Prozess = Eltern-Prozess („Parent Process“)
- Gründe
  - parallele Ausführung von Aufgaben
  - Bessere Auslastung
  - Zugänglichmachen von Funktionen (z.b. sshd & shell)
- Nur in Linux / Unix bekannt
- Windows kennt „CreateProcess“ (spezielle Art von Fork)

Forking II - Ablauf

- Prozess ruft „fork()“ auf (Systemfunktion)
- „fork()“ erzeugt neue Prozessumgebung
- „fork()“ erzeugt neuen Adressraum
- „fork()“ kopiert Elternadressraum in Kindadressraum
- „fork()“ setzt PC auf gleiche Stelle im Code
- „fork()“ setzt Kind-Prozess auf „ready“

erzeugt 1:1 Kopie

1:1 Kopie?

Unterscheidung?!

Rückgabewert von fork()

- Eltern-Prozess
  - -1 falls Fehler
  - PID („Process ID“) des Kindes bei Erfolg
- Kind-Prozess
  - 0

Forking III - Beispiel

```
int main(int argc, char** argv){
        int child_pid = fork();
        if(child_pid == 0){
                printf("I am the Child %d\n", getpid());
        }
        else
        {
                 printf("PID %d - I am your Father!\n", child_pid);
        }
        return 0;
}
```

```
pi@raspberrypi:~ $ gcc fork_1.c -o fork_1
pi@raspberrypi:~ $ ./fork_1
PID 12874 - I am your Father!
I am the Child 12874
```

Forking IV - Synchronisation

- Grundproblem
  - Eltern-Prozess und Kind-Prozess laufen „unabhängig“
  - Eltern-Prozess wartet nicht auf Kind-Prozess
  - Kind-Prozess kann verwaisen / Zombie werden

Aufsichtspflicht der Eltern

Synchronisation

Warte-Funktionen

- waitpid(pid_t pid, int *status, int options);
  - wartet auf Statusänderung des Kind-Prozess *pid*
  - *pid* = -1 -> wartet auf beliebigen Kind-Prozess
  - *status* -> waitpid speichert status des Kind-Prozesses an dieser Adresse
  - *options* -> beinflussen des waitpid() verhaltens
  - gibt PID des geänderten Kind-Prozesses zurück

- wait(int *status)
  - wartet bis eines der Kind-Prozess terminiert
  - equivalent zu waitpid(-1, &status, 0);
  - gibt PID des terminierten Kind-Prozesses zurück

Forking V - Synchronisation

```
int main(int argc, char** argv){
        int child_pid = fork();
        if(child_pid == 0){
                printf("I am the Child %d\n", getpid());
           printf("Not now....\n");
           sleep(10);
           printf("Ok I am done!\n");
         }
         else
         {
           printf("PID %d - I am your Father!\n", child_pid);
           printf("I am sure my child does nothing...\n");
           sleep(2);
           printf("Ok lets see if they are still running\n");
           int status = -1;
           int return_pid = 0;
           return_pid = wait(&status);
           printf("My child %d returned with %i\n",return_pid,status);
         }
         return 0;
}
```

```
pi@raspberrypi:~ $ gcc fork_2.c -o fork_2
pi@raspberrypi:~ $ ./fork_2
PID 23460 - I am your Father!
I am the Child 23460
Not now....
I am sure my child does nothing...
Ok lets see if they are still running
Ok I am done!
My child 23460 returned with 0
```

Exkurs: waitpid & Status Codes

Grundproblem

- waitpid gibt status nicht 1:1 wieder
- „status" wird mit diversen Informationen gefüllt

```
martin@ubuntu:~/Desktop/Vorlesung$ gcc fork_2.1.c -o fork_2.1
martin@ubuntu:~/Desktop/Vorlesung$ ./fork_2.1
PID 16225 - I am your Father!
I am the Child 16225
I will return 20!
My child 16225 returned with 5120
```

Macros um Status zu ermitteln

```
int main(int argc, char** argv){
        int child_pid = fork();
        if(child_pid == 0){
                printf("I am the Child %d\n", getpid());
                printf("I will return 20!\n");
                return 20;
        }
        else
        {
                printf("PID %d - I am your Father!\n", child_pid);
                int status = -1;
                int return_pid = 0;
                return_pid = wait(&status);
                printf("My child %d returned with %i\n",return_pid,status);
        }
        return 0;
}
```

Exkurs: waitpid & Status Codes

Macros

- WIFEXITED – gibt „true“ zurück, wenn Kind-Prozess normal beendet wurde
- WIFSIGNALED gibt „true“ zurück, wenn Kind-Prozess durch Signal terminiert wurde (z.B. kill -9 <pid>)
- WEXITSTATUS – gibt den Status Code zurück, mit dem der Kind-Prozess terminierte. **Sollte nur genutzt werden wenn vorher WIFEXITED „true“ zurückgab!**

```
martin@ubuntu:~/Desktop/Vorlesung$ gcc fork_2.1.c -o fork_2.1
martin@ubuntu:~/Desktop/Vorlesung$ ./fork_2.1
PID 16261 - I am your Father!
I am the Child 16261
I will return 20!
My child 16261 returned with 20
```

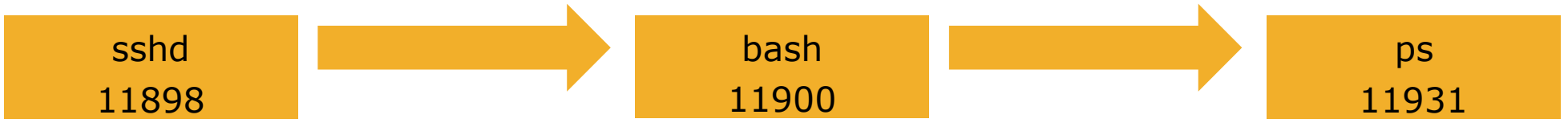
```
int main(int argc, char** argv){
        int child_pid = fork();
        if(child_pid == 0){
                printf("I am the Child %d\n", getpid());
                printf("I will return 20!\n");
                return 20;
        }
        else
        {
                printf("PID %d - I am your Father!\n", child_pid);
                int status = -1;
                int return_pid = 0;
                return_pid = wait(&status);
                if(WIFEXITED(status)){
                        printf("My child %d returned with
                                %i\n",return_pid,WEXITSTATUS(status));
                }
        }
        return 0;
}
```

Forking VI

Forks in Linux

„ps lx“

```
pi@raspberrypi:~ $ ps lx
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4  1000  1241     1  20   0   5104  3320 SyS_ep Ss   ?          0:00 /lib/system
5  1000  1245  1241  20   0   7040  1456 -      S    ?          0:00 (sd-pam)
4  1000  1251   644  20   0   6500  4424 wait_w S+   ttyl       0:00 -bash
4  1000  1254  1226  20   0  52244 12060 poll_s Ssl  ?          0:10 /usr/bin/lx
1  1000  1308  1254  20   0   3696   224 -      Ss   ?          0:12 /usr/bin/ss
1  1000  1311     1  20   0   3284  1504 poll_s S    ?          0:00 /usr/bin/db
1  1000  1312     1  20   0   5488  2196 SyS_ep Ss   ?          0:00 /usr/bin/db
0  1000  1318     1  20   0  31232  5552 poll_s Sl   ?          0:00 /usr/lib/gv
0  1000  1335     1  20   0  49060  5360 futex_ Sl   ?          0:00 /usr/lib/gv
1  1000  1482     1  20   0   1908   100 wait   S    ?          0:00 /bin/sh /us
0  1000  1483  1482  20   0   5844  2004 poll_s S    ?          0:00 /usr/bin/xp
1  1000  1505     1  20   0  30232  6016 poll_s Ssl  ?          0:00 /usr/lib/me
0  1000  1543     1  20   0  60744  7304 poll_s Sl   ?          0:08 /usr/lib/gv
5  1000 11898 11892  20   0  12340  3060 -      S    ?          0:00 sshd: pi@pt
0  1000 11900 11898  20   0   6488  4428 wait   Ss   pts/0      0:00 -bash
0  1000 11931 11900  20   0   4284  1780 -      R+   pts/0      0:00 ps lx
```



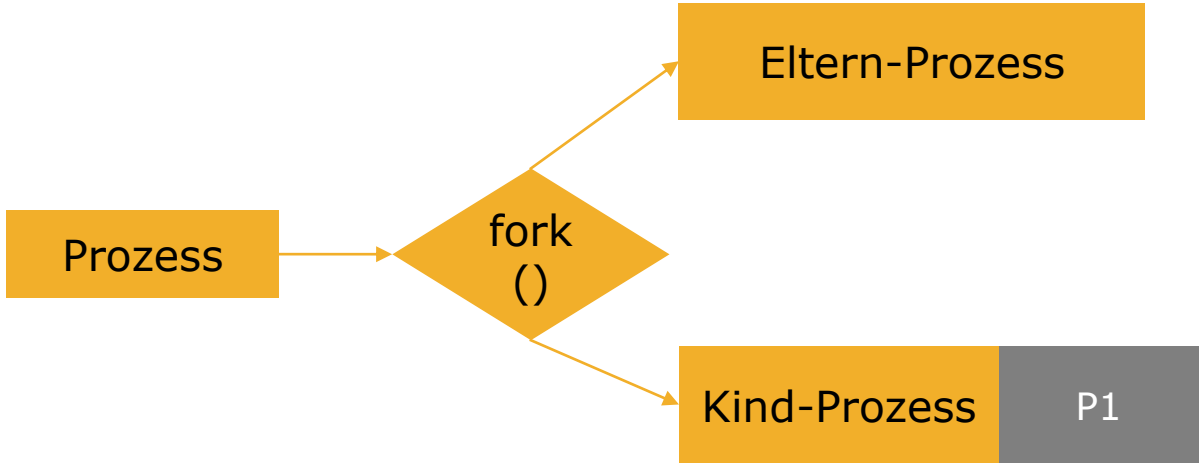
Forking VII

Problem

- fork() erzeugt 1:1 Kopie
- Unterschiedliche Tasks => sehr komplex

Lösung

- fork() erzeugt 1:1 kopie
- Im Kind-Prozess wird exec() aufgerufen
  - Ersetzt Kind-Prozess mit neuem „Image“



Forking VIII

```
int main(int argc, char** argv){
        int child_pid = fork();
        if(child_pid == 0){
                char *args[]={"./hello_world",NULL};
                execvp(args[0],args);
        }
        else
        {
                printf("PID %d - I am your Father!\n", child_pid);
                printf("I am sure my child does nothing...\n");
                sleep(2);
                printf("Ok lets see if they are still running\n");
                int status = -1;
                int return_pid = 0;
                return_pid = wait(&status);
                printf("My child %d returned with %i\n",return_pid,status);
        }
        return 0;
}
```

```
pi@raspberrypi:~ $ gcc hello_world.c -o hello_world
pi@raspberrypi:~ $ gcc fork_3.c -o fork_3
pi@raspberrypi:~ $ ./fork_3
PID 25859 - I am your Father!
I am sure my child does nothing...
Hello World!!!!!
Ok lets see if they are still running
My child 25859 returned with 0
```

Exkurs: Forking in Windows

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

int main(int argc, char** argv){
		STARTUPINFO sInfo;
		PROCESS_INFORMATION pInfo;
		ZeroMemory( &sInfo, sizeof(sInfo));
		sInfo.cb = sizeof(sInfo);
		ZeroMemory (&pInfo, sizeof(pInfo));

		if(! CreateProcess( NULL, "C:\\process.exe", NULL, NULL, FALSE, 0, NULL, NULL,  &sInfo, &pInfo)) {
				printf("Could not run CreateProcess!\n");
				return -1;
		}
		else
		{
				WaitForSingleObject(pInfo.hProcess, INFINTE);
				printf("Process finished!!!\n");
				CloseHandle(pInfo.hProcess);
				CloseHandle(pInfo.hTrhead);
		}
		return 0;
}
```

Threads I

Recap - Fork

- „fork()“ erzeugt neue Prozessumgebung
- „fork()“ erzeugt neuen Adressraum
- „fork()“ kopiert Eltern-Adressraum in Kind-Adressraum
- „fork()“ setzt PC auf gleiche Stelle im Code
- Eltern- und Kind-Prozess unabhängig

Langsam / Ineffizient

Lösung: Threads

- Prozess hat mehrere Befehlsstränge
- Befehlsstränge nutzen die gleiche Prozessumgebung und gleichen Adressraum

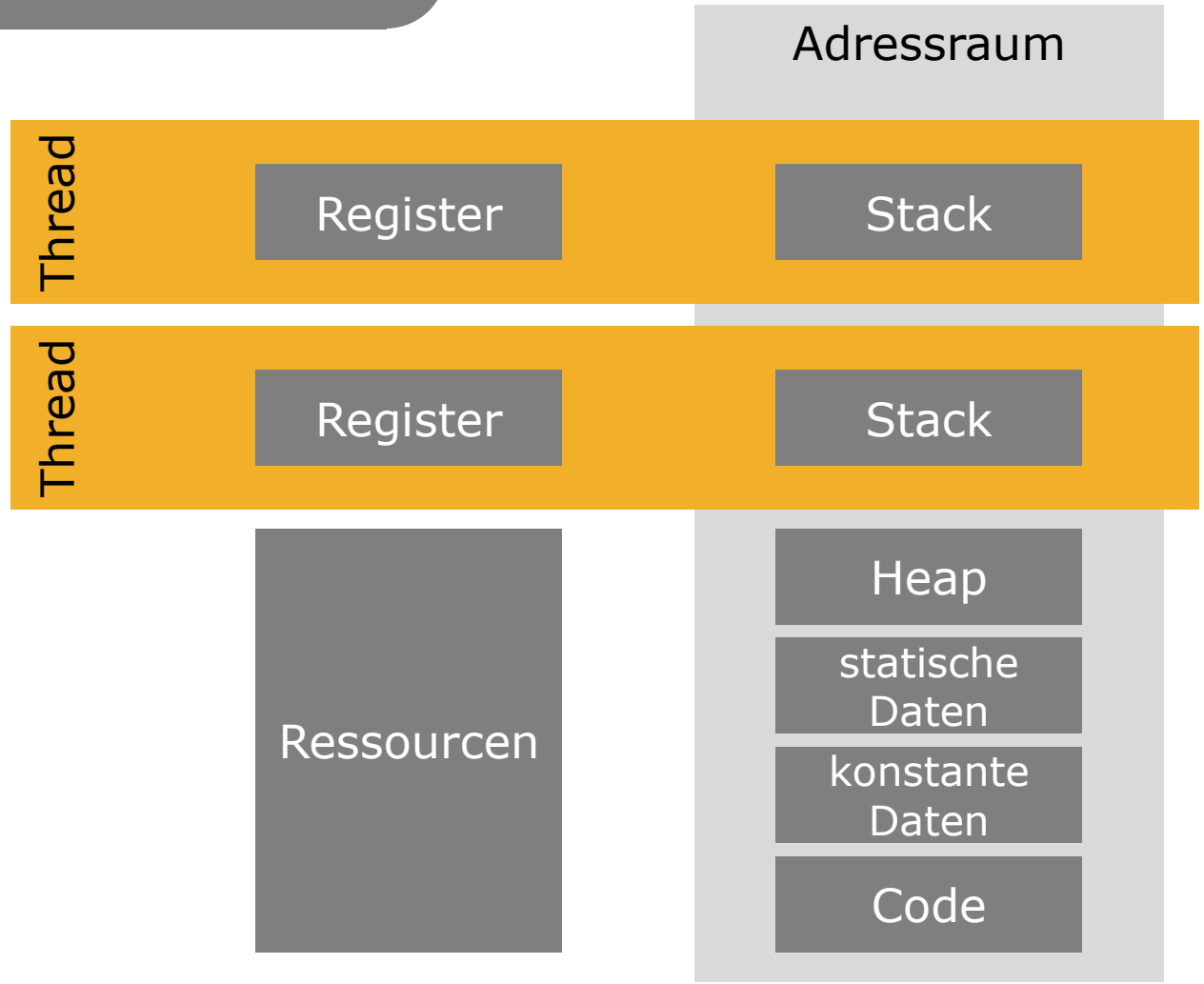
- Threads eines Prozesses greifen auf gleiche Daten zu
- Schnelles Erstellen / Löschen, da leichtgewichtig
- Schnelles Umschalten, da kein Kontextwechsel

Threads II

Aufbau

- eigener Stack (innerhalb des Prozessspaces!)
- Ausführungszustand (ready, running, …)
- eigene Kopie der Prozessorregister (z.b. PC, SP)
- Speicher für Thread-lokale Variablen
- geteilt
  - Heap
  - statische & konstante Daten
  - Code

Ältere OS kennen Threads nicht



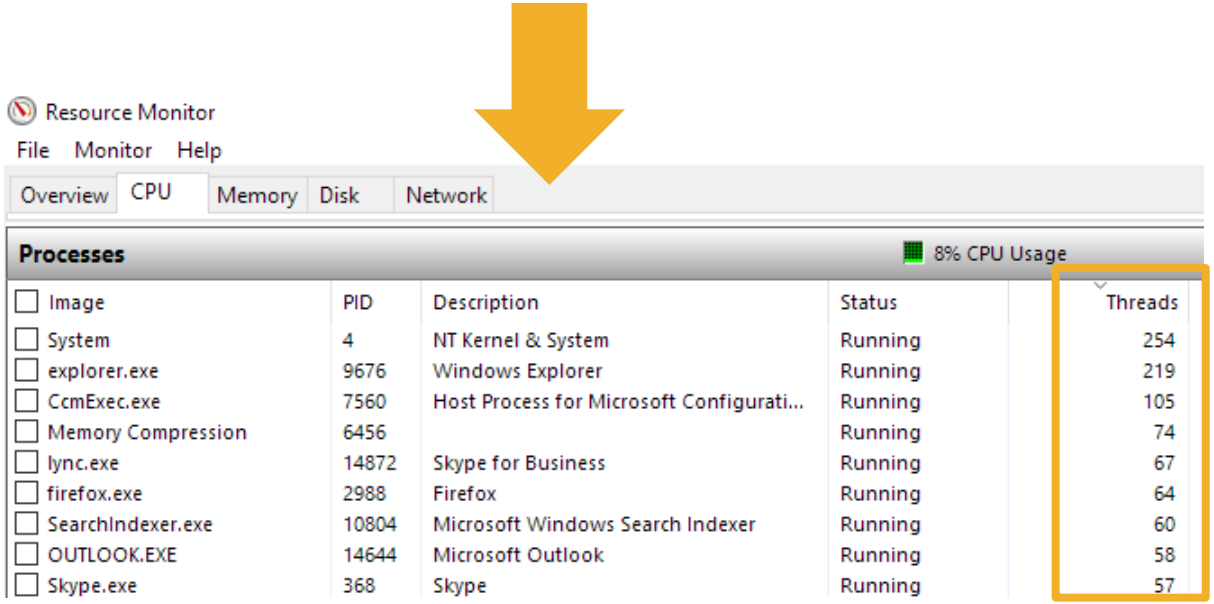
Threads III – use Cases

Use Cases

- Regelmäßige Datensicherung im Hintergrund
- „gleichzeitige" Vordergrund und Hintergrundaktivitäten
  - z.B. während Benutzereingabe, Kontrolle der Eingabe / Berechnung im Hintergrund (z.B. Excel)
- Animation-rendering
  - Pro Frame ein Thread – parallele Berechnung von Frames
  - Frame in Quadranten einteilen – pro Quadrant ein Frame
- GUI vs. Programmlogik
  - Trennen der GUI vom Programm, so dass GUI immer reagieren kann

Reale Beispiele

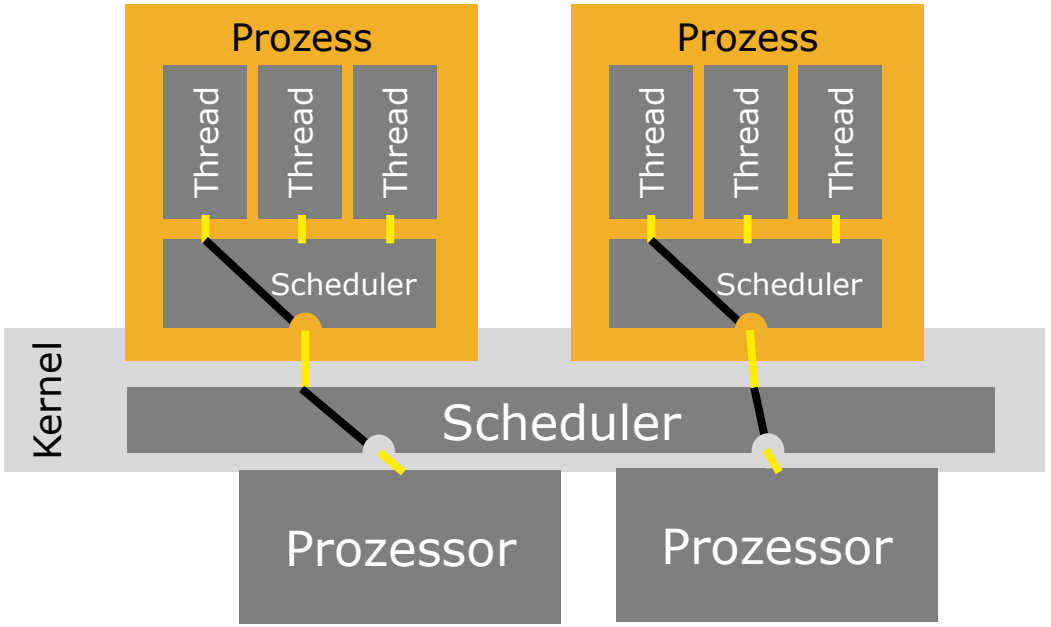
- Apache Webserver hat pro Verbindung i.d.R 1 Thread
- Java Swing trennt mit Threads UI vom Programm
- Windows CMD Tool „robocopy" nutzt mehrere Threads um kopieren zu beschleunigen



Threads IV – Arten

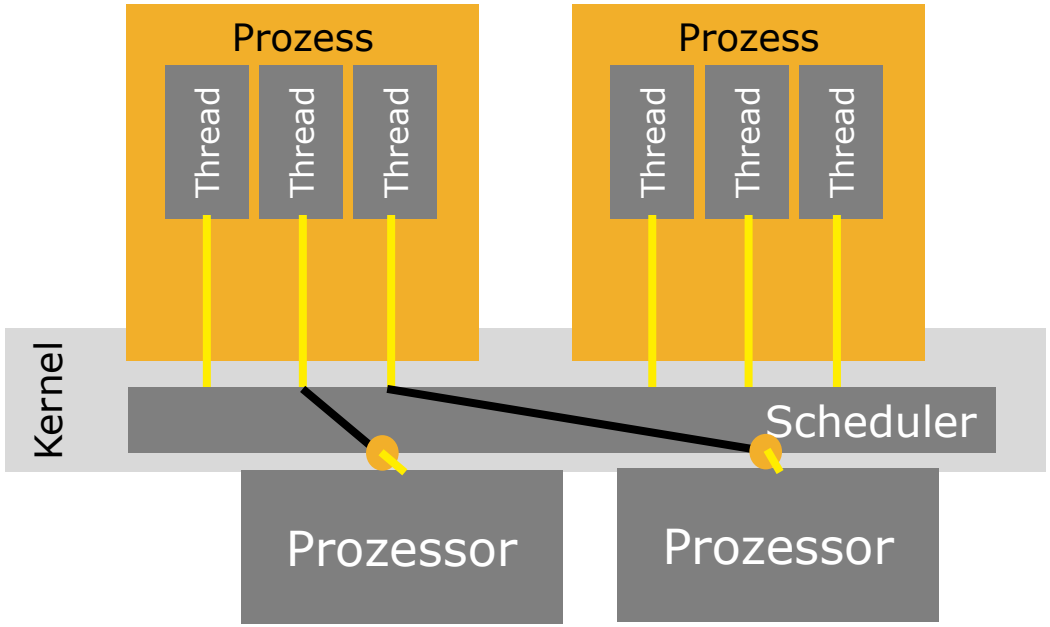
User-Level-Threads

- OS muss Threads nicht unterstützen
- Threads werden in Bibliothek implementiert
- Verschiedene Threads eines Prozesses können nicht auf unterschiedlichen Prozessoren laufen



Kernel-Level-Threads

- OS muss Threads unterstützen
- Systemfunktionen werden genutzt um Threads zu implementieren
- Jeder Thread eines Prozesses kann auf unterschiedlichen Prozessoren laufen



Threads V - Arten

User Threads vs. Kernel Threads

- Kernel Thread nutzt Systemaufruf - Umschalten in Kernel Mode nötig

Kernel Thread ist langsamer

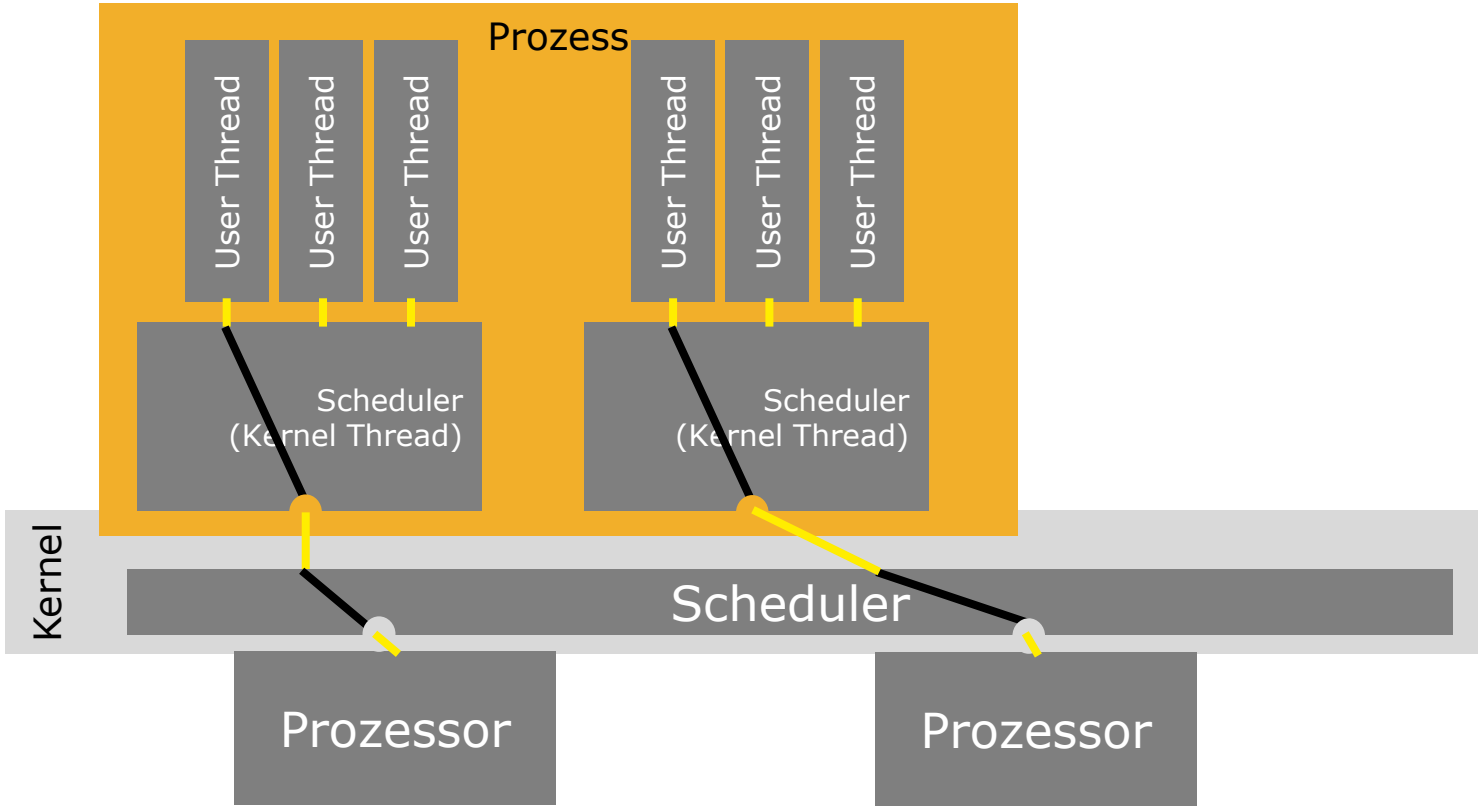
User Threads vs. Kernel Threads

- blockierender Kernel Thread kann durch Kern gestoppt und mit anderen Thread/Prozess ausgetauscht werden

Kernel Thread ist schneller

Prozesse können beide Arten kombinieren um Vorteile zu nutzen

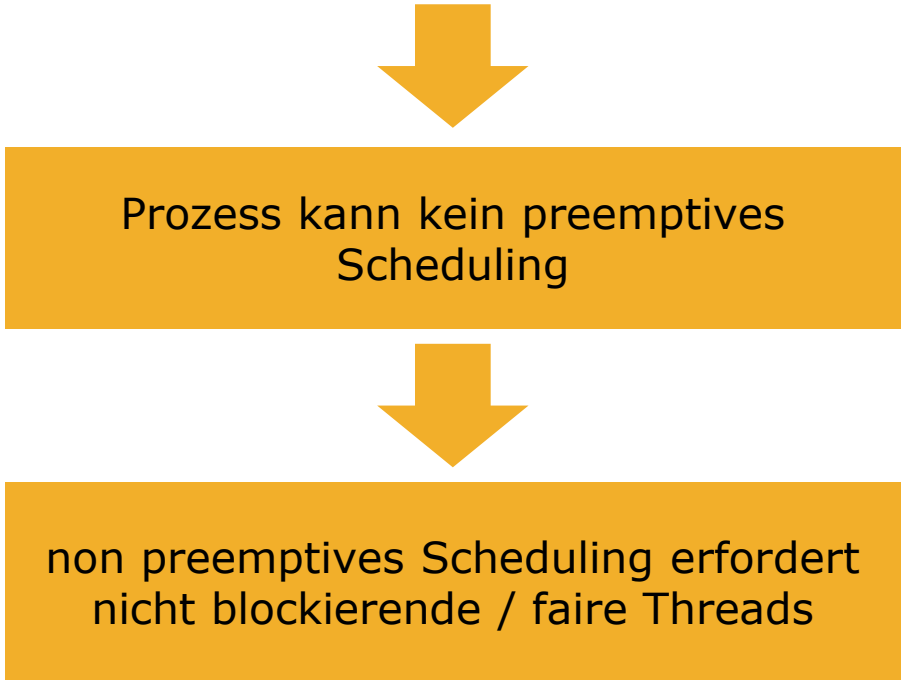
Threads VI


Prozess
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
Scheduler
(Kernel Thread)
Scheduler
(Kernel Thread)
Kernel
Scheduler
Prozessor
Prozessor

Threads VII - Scheduling

User Thread

- OS kennt Threads nicht!
- Scheduling muss vom Prozess gemacht werden



Kernel Thread

- OS weiß von diversen Threads
- Preemptives Scheduling kann angewandt werden

Threads VIII – In der Praxis

Bibliothek „pthreads“

- POSIX Standard (IEEE 1003.1c-1995)
- Implementierung für fast alle Unix basierenden OS
  - Linux
  - MacOS
  - Android
- Windows kennt nativ keine Posix Threads
  - pthread ist „Wrapper“ für Win-API
- OS entscheidet ob User-Thread oder Kernel-Thread
  - z.B. Linux – fast alles ist ein Kernel-Thread

Wichtige Funktionen in „pthreads“

Funktion	Beschreibung
pthread_create	neue Thread wird erzeugt
pthread_exit	aufrufender Thread wird beendet
pthread_join	warten bis bestimmter Thread beendet ist
pthread_yield	gibt Prozessor für anderen Thread frei
pthread_kill	sendet Signal zu Thread
pthread_self	gibt die jeweilige Thread-ID zurück

Threads IX – Beispiel

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

void *thread1(void* value){
	int *input = (int *)value;
	printf("Hello %i from Thread!\n",*input);
	return 0;
}


int main(int argc, char* argv){
	int val = 42;
	pthread_t mythread;

	if(pthread_create(&mythread, NULL, &thread1,
&val)) {
		fprintf(stderr, "Failed to start thread\n");
		return 1;
	}

	if(pthread_join(mythread, NULL)) {
		fprintf(stderr, "Failed to join thread\n");
		return 2;
	}

	printf("We are done!\n");
}
```

Threads starten (mit Pointer zu Funktion)

mit -lpthread kompilieren!

```
~/Desktop/Vorlesung/threads$ gcc threads.c -o threads -lpthread
martin@ubuntu:~/Desktop/Vorlesung/threads$ ./threads
Hello 42 from Thread!
We are done!
```

Auf Thread warten