

# Betriebssysteme 1

SS 2020

Prof. Dr.-Ing. Hans-Georg Eßer  
Fachhochschule Südwestfalen

Foliensatz C:

- Threads

V2.0, 2020/04/23

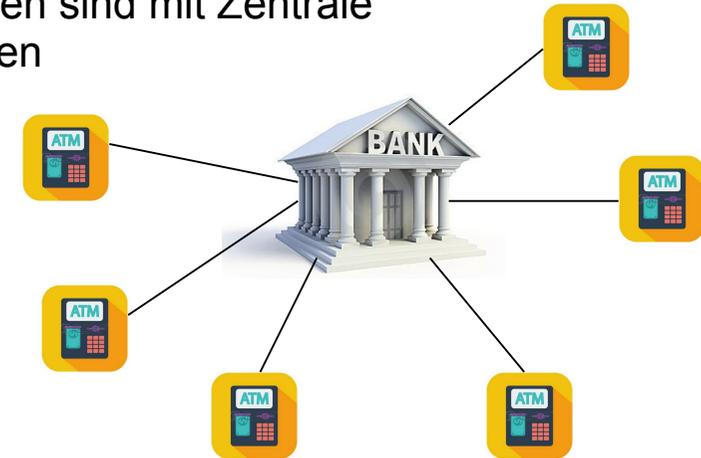
23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-1

## Motivation (1)

- Szenario: Bankzentrale + Geldautomaten
- Automaten sind mit Zentrale verbunden



23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-2

## Motivation (2)

- Zentrale führt Server-Anwendung aus, vereinfacht:

```

init (accounts[]);           // Konten-DB

do forever {
  conn = accept_atm_connection ();
  acc_no = recv (conn);
  pin    = recv (conn);
  if (!auth(acc_no,pin)) { terminate (conn); continue; }

  euro = accounts[acc_no].euro;   // Abfrage
  send (conn, euro);
  recv (conn, new_euro);
  accounts[acc_no].euro = new_euro; // Aktualisierung
  terminate (conn);
}

```

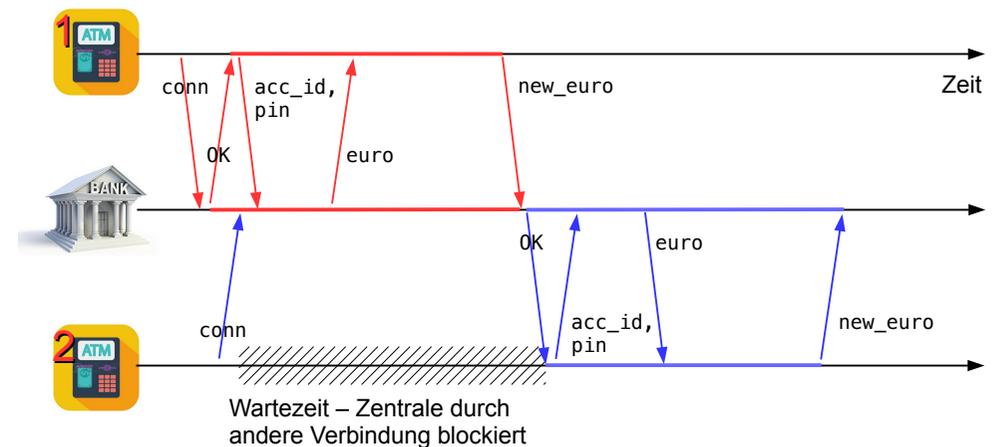
23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-3

## Motivation (3)

- Verbindung eines Automaten blockiert Server



23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

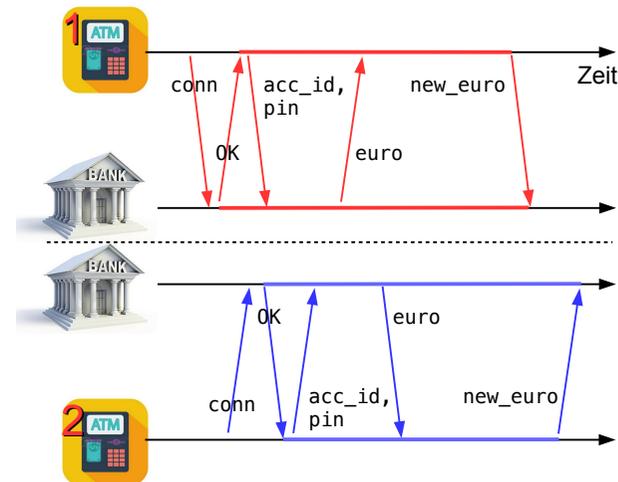
Folie C-4

## Motivation (4)

- Wartezeiten verkürzen → parallele Verbindung mehrerer Automaten zulassen
- Erste Idee: mehrere Server-Prozesse, Anfrage an einen freien Server-Prozess weiter reichen

## Motivation (5)

- Mehrere Server-Prozesse



- Problem: Prozess-Daten (DB) doppelt → Abstimmung der Prozesse erforderlich

## Threads: Theorie

## Threads (1)

### Was ist ein Thread?

- Aktivitätsstrang in einem Prozess
- einer von mehreren
- Gemeinsamer Zugriff auf Daten des Prozess
- aber: Stack, Befehlszähler, Stack Pointer, Hardware-Register (Kontext) separat pro Thread
- Scheduler verwaltet auch Threads – oder nicht (→ Kernel- oder User-Level-Threads)

## Threads (2)

### Warum Threads?

- Multi-Prozessor- bzw. Multi-Core-System: Mehrere Threads echt gleichzeitig aktiv
- Ist ein Thread durch I/O blockiert, arbeiten die anderen weiter
- Besteht Programm logisch aus parallelen Abläufen, ist die Programmierung mit Threads einfacher

## Threads (3): Beispiele

### Zwei unterschiedliche Aktivitätsstränge: Komplexe Berechnung mit Benutzeranfragen

Ohne Threads:

```
while (1) {
    rechne_ein_bisschen ();
    if benutzereingabe (x) {
        bearbeite_eingabe (x);
    }
}
```

## Threads (4): Beispiele

### Komplexe Berechnung mit Benutzeranfragen

Mit Threads:

```
T1:
while (1) {
    rechne_alles ();
}
```

```
T2:
while(1) {
    if benutzereingabe (x) {
        bearbeite_eingabe (x);
    }
}
```

## Threads (5): Beispiele

### Server-Prozess, der viele Anfragen bearbeitet

- Prozess öffnet Port
- Für jede eingehende Verbindung: Neuen Thread erzeugen, der diese Anfrage bearbeitet
- Nach Verbindungsabbruch Thread beenden
- Vorteil: Keine Prozess-Erzeugung (Betriebssystem!) nötig

## Threads (6): Beispiel MySQL

Ein Prozess, neun Threads:

```
[esser:~]$ ps -eLf | grep mysql
UID      PID     PPID    LWP  C  NLWP  STIME TTY          TIME CMD
-----
root     27833   1 27833  0   1 Jan04 ?        00:00:00 /bin/sh /usr/bin/mysqld_safe
mysql    27870  27833  27870  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27872  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27873  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27874  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27875  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27876  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27877  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27878  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
mysql    27870  27833  27879  0   9 Jan04 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
```

[esser:~]\$

PID: Process ID  
 PPID: Parent Process ID  
 LWP: Light Weight Process ID (Thread-ID)  
 NLWP: Number of Light Weight Processes

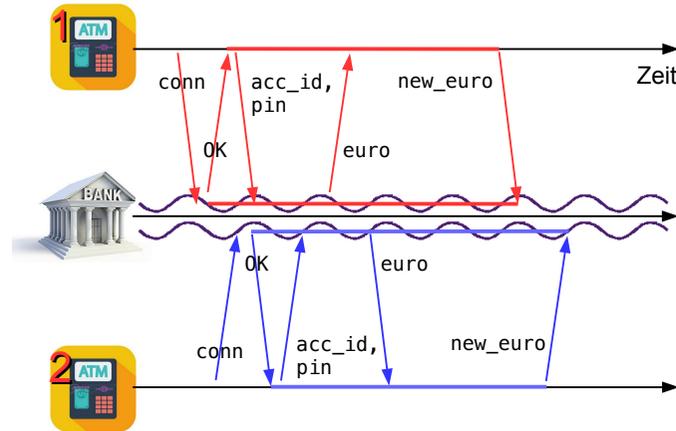
23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-13

## Threads (7): Geldautomaten

- Server-Prozess mit mehreren Threads (~~~~)



23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-14

## Threads (8): Geldautomaten

```
function main () {
  init (accounts[]);           // Konten-DB
  do forever {
    conn = accept_atm_connection ();
    create_thread (handler, conn);
  }
}

function handler (int conn) {
  acc_no = recv (conn);
  pin    = recv (conn);
  if (!auth(acc_no,pin)) { terminate (conn); return; }

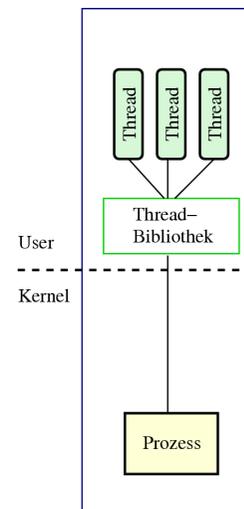
  euro = accounts[acc_no].euro; // Abfrage
  send (conn, euro);
  recv (conn, new_euro);
  accounts[acc_no].euro = new_euro; // Aktualisierung
  terminate (conn);
}
```

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-15

## User Level Threads



User Level Threads

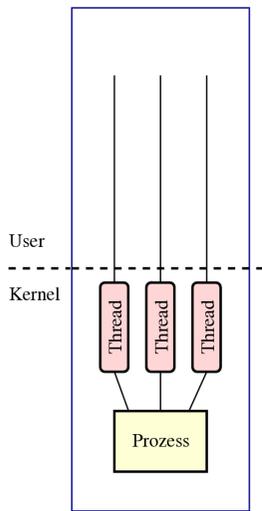
- BS kennt kein Thread-Konzept, verwaltet nur Prozesse
- Programm bindet Thread-Bibliothek ein, zuständig für:
  - Erzeugen, Zerstören
  - Scheduling
- Wenn ein Thread wegen I/O wartet, dann der ganze Prozess
- Ansonsten sehr effizient

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-16

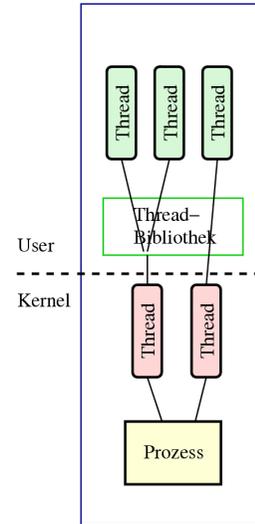
## Kernel Level Threads



Kernel Level Threads

- BS kennt Threads
- BS verwaltet die Threads:
  - Erzeugen, Zerstören
  - Scheduling
- I/O eines Threads blockiert nicht die übrigen
- Aufwendig: Context Switch zwischen Threads ähnlich komplex wie bei Prozessen

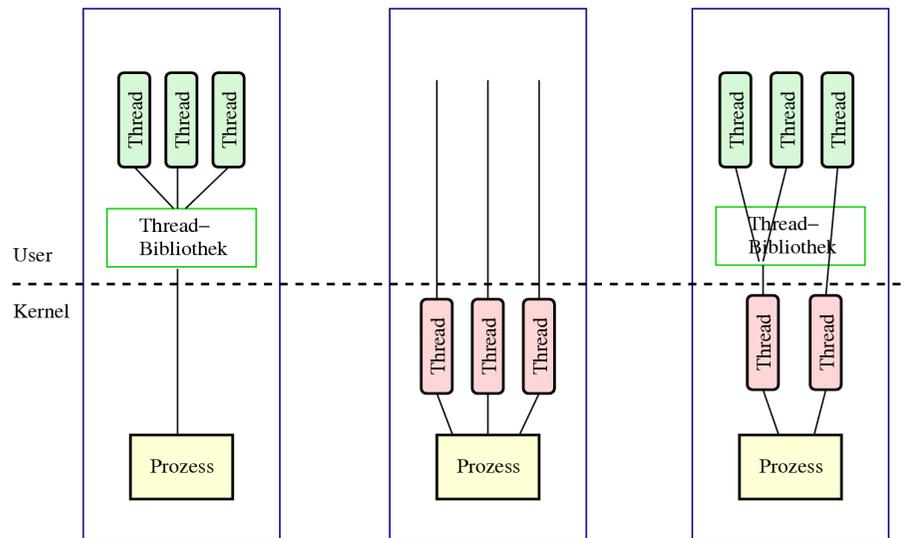
## Gemischte Threads



Gemischtes Modell

- Beide Ansätze kombinieren
- KL-Threads + UL-Threads
- Thread-Bibliothek verteilt UL-Threads auf die KL-Threads
- z.B. I/O-Anteile auf einem KL-Thread
- Vorteile beider Welten:
  - I/O blockiert nur einen KL-Thread
  - Wechsel zwischen UL-Threads ist effizient
- SMP: Mehrere CPUs benutzen

## Thread-Typen, Übersicht



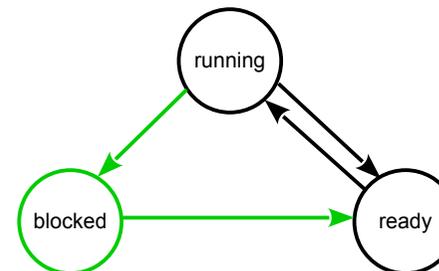
User Level Threads

Kernel Level Threads

Gemischtes Modell

## Thread-Zustände

- Prozess-Zustände suspended, swapped etc. nicht auf Threads übertragbar (warum nicht?)
- Darum (i. W.) nur drei Thread-Zustände



(Zustand *blocked* nur bei Kernel-Level-Threads!)

## Prozesse und Threads (Praxis, Entwicklersicht)

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-21

## Prozesse und Threads erzeugen (1/11)

Neuer Prozess: `fork ( )`

```
main() {
    int pid = fork(); /* Sohnprozess erzeugen */
    if (pid == 0) {
        printf("Ich bin der Sohn, meine PID ist %d.\n", getpid() );
    }
    else {
        printf("Ich bin der Vater, mein Sohn hat die PID %d.\n", pid);
    }
}
```

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-22

## Prozesse und Threads erzeugen (2/11)

Anderes Programm starten: `fork + exec`

```
main() {
    int pid=fork(); /* Sohnprozess erzeugen */
    if (pid == 0) {
        /* Sohn startet externes Programm */
        execl( "/usr/bin/gedit", "/etc/fstab", (char *) 0 );
    }
    else {
        printf("Es sollte jetzt ein Editor starten...\n");
    }
}
```

Andere Betriebssysteme oft nur: „spawn“

```
main() {
    WinExec("notepad.exe", SW_NORMAL); /* Sohn erzeugen */
}
```

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-23

## Prozesse und Threads erzeugen (3/11)

Warten auf Sohn-Prozess: `wait ( )`

```
#include <unistd.h> /* sleep() */

main()
{
    int pid=fork(); /* Sohnprozess erzeugen */
    if (pid == 0)
    {
        sleep(2); /* 2 sek. schlafen legen */
        printf("Ich bin der Sohn, meine PID ist %d\n", getpid() );
    }
    else
    {
        printf("Ich bin der Vater, mein Sohn hat die PID %d\n", pid);
        wait(); /* auf Sohn warten */
    }
}
```

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-24

## Prozesse und Threads erzeugen (4/11)

Wirklich mehrere Prozesse:

Nach `fork ( )` zwei Prozesse in der Prozessliste

```
> pstree | grep simple
... -bash---simplefork---simplefork

> ps w | grep simple
25684 pts/16 S+      0:00 ./simplefork
25685 pts/16 S+      0:00 ./simplefork
```

## Prozesse und Threads erzeugen (5/11)

Linux: pthread-Bibliothek (POSIX Threads)

	Thread	Prozess
Erzeugen	<code>pthread_create()</code>	<code>fork()</code>
Auf Ende warten	<code>pthread_join()</code>	<code>wait()</code>

- Bibliothek einbinden:  
`#include <pthread.h>`
- Kompilieren:  
`gcc -lpthread -o prog prog.c`

## Prozesse und Threads erzeugen (6/11)

- Neuer Thread:  
`pthread_create ( )` erhält als Argument eine Funktion, die im neuen Thread läuft.
- Auf Thread-Ende warten:  
`pthread_join ( )` wartet auf einen *bestimmten* Thread.

## Prozesse und Threads erzeugen (7/11)

1. Thread-Funktion definieren:

```
void *thread_funktion(void *arg) {
    ...
    return ...;
}
```

2. Thread erzeugen:

```
pthread_t thread;

if ( pthread_create( &thread, NULL,
    thread_funktion, NULL) ) {
    printf("Fehler bei Thread-Erzeugung.\n");
    abort();
}
```

## Prozesse und Threads erzeugen (8/11)

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

void *thread_function1(void *arg) {
    int i;
    for ( i=0; i<10; i++ ) {
        printf("Thread 1 sagt Hi!\n");
        sleep(1);
    }
    return NULL;
}

void *thread_function2(void *arg) {
    int i;
    for ( i=0; i<10; i++ ) {
        printf("Thread 2 sagt Hallo!\n");
        sleep(1);
    }
    return NULL;
}

int main(void) {
    pthread_t mythread1;
    pthread_t mythread2;

    if ( pthread_create( &mythread1, NULL,
        thread_function1, NULL ) ) {
        printf("Fehler bei Thread-Erzeugung.");
        abort();
    }

    sleep(5);

    if ( pthread_create( &mythread2, NULL,
        thread_function2, NULL ) ) {
        printf("Fehler bei Thread-Erzeugung.");
        abort();
    }

    printf("bin noch hier...\n");

    sleep(5);

    printf("Thread 1 ist weg\n");

    if ( pthread_join ( mythread1, NULL ) ) {
        printf("Fehler beim Join.");
        abort();
    }

    printf("Thread 2 ist weg\n");

    if ( pthread_join ( mythread2, NULL ) ) {
        printf("Fehler beim Join.");
        abort();
    }

    printf("Thread 2 ist weg\n");

    exit(0);
}
```

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-29

## Prozesse und Threads erzeugen (9/11)

### Keine „Vater-“ oder „Kind-Threads“

- POSIX-Threads kennen keine Verwandtschaft wie Prozesse (Vater- und Sohnprozess)
- Zum Warten auf einen Thread ist Thread-Variable nötig: `pthread_join (thread, ..)`

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-30

## Prozesse und Threads erzeugen (10/11)

Prozess mit mehreren Threads:

- Nur ein Eintrag in normaler Prozessliste
- Status: „l“, multi-threaded
- Über `ps -eLf` Thread-Informationen
  - NLWP: Number of light weight processes
  - LWP: Thread ID

```
> ps auxw | grep thread
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
esser    12022  0.0  0.0  17976   436 pts/15  S1+  22:58   0:00 ./thread
```

```
> ps -eLf | grep thread
UID      PID  PPID  LWP  C  NLWP STIME TTY          TIME CMD
esser    12166  4031  12166  0   3  23:01 pts/15    00:00:00 ./thread1
esser    12166  4031  12167  0   3  23:01 pts/15    00:00:00 ./thread1
esser    12166  4031  12177  0   3  23:01 pts/15    00:00:00 ./thread1
```

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-31

## Prozesse und Threads erzeugen (11/11)

Unterschiedliche Semantik:

- Prozess erzeugen mit `fork ( )`
  - erzeugt zwei (fast) identische Prozesse,
  - beide Prozesse setzen Ausführung an gleicher Stelle fort (nach Rückkehr aus `fork`-Aufruf)
- Thread erzeugen mit `pthread_create (..., funktion, ...)`
  - erzeugt neuen Thread, der in die angeg. Funktion springt
  - erzeugender Prozess setzt Ausführung hinter `pthread_create`-Aufruf fort

23.04.2020

Betriebssysteme 1, SS 2020, Hans-Georg Eßer

Folie C-32

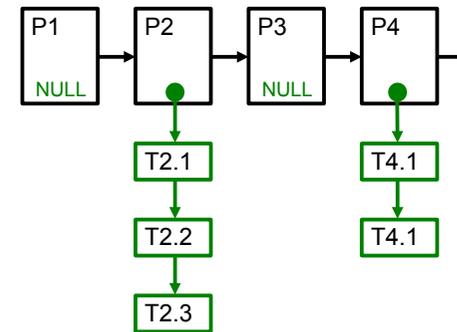
## Threads im Kernel (1/2)

- Linux verwendet für Threads und Prozesse die gleichen Verwaltungsstrukturen (task list)
- Thread: Prozess, der sich mit anderen Prozessen bestimmte Ressourcen teilt, z. B.
  - virtueller Speicher
  - offene Dateien
- Jeder Thread hat `task_struct` und sieht für den Kernel wie ein normaler Prozess aus

## Threads im Kernel (2/2)

Fundamental anders als z. B. Windows und Solaris

Modell 1:  
reine Prozesslisten



Modell 2 (Linux):  
Prozesse + Threads gemischt

