

3 Threads

3.1 Grundlagen

- lokale und globale Variable erhöhen (count_sequential.c)

```

...
#define MAX_OUT_LOOP    4          /* number of outer loops      */
#define MAX_IN_LOOP     5000000   /* number of inner loops     */
#define NUM_CALL        3          /* number of function calls  */

long    global_cnt;               /* global counter             */
time_t  s_time, e_time;          /* start/end time of counting */
clock_t CPU_time;                /* used CPU time for counting */

void count (long *p_local_cnt);  /* increments some variables  */
void prt_result (long local_cnt[]); /* prints some results        */

int main (int argc, char *argv[])
{
    long local_cnt[NUM_CALL];     /* local counters            */
    int i;                        /* loop variable             */

    global_cnt = 0;
    for (i = 0; i < NUM_CALL; ++i)
    {
        local_cnt[i] = 0;
    }

    /* call functions (later: create processes or threads) */
    s_time = time (NULL);
    CPU_time = clock ();
    for (i = 0; i < NUM_CALL; ++i)
    {
        count (&local_cnt[i]);
    }
    CPU_time = clock () - CPU_time;
    e_time = time (NULL);
    prt_result (local_cnt);
    return EXIT_SUCCESS;
}

void count (long *p_local_cnt)
{
    int i, j;                      /* loop variables            */

    for (i = 0; i < MAX_OUT_LOOP; ++i)
    {
        printf ("Process %ld is counting ... \n", (long) getpid ());
        for (j = 0; j < MAX_IN_LOOP; ++j)
        {
            (*p_local_cnt)++;
            global_cnt++;
        }
    }
}
...

```

- Ausgabe des Programms (Pentium M, 1.6 GHz, Cygwin 1.5.25, gcc-3.4.3)

Process 3116 is counting ...

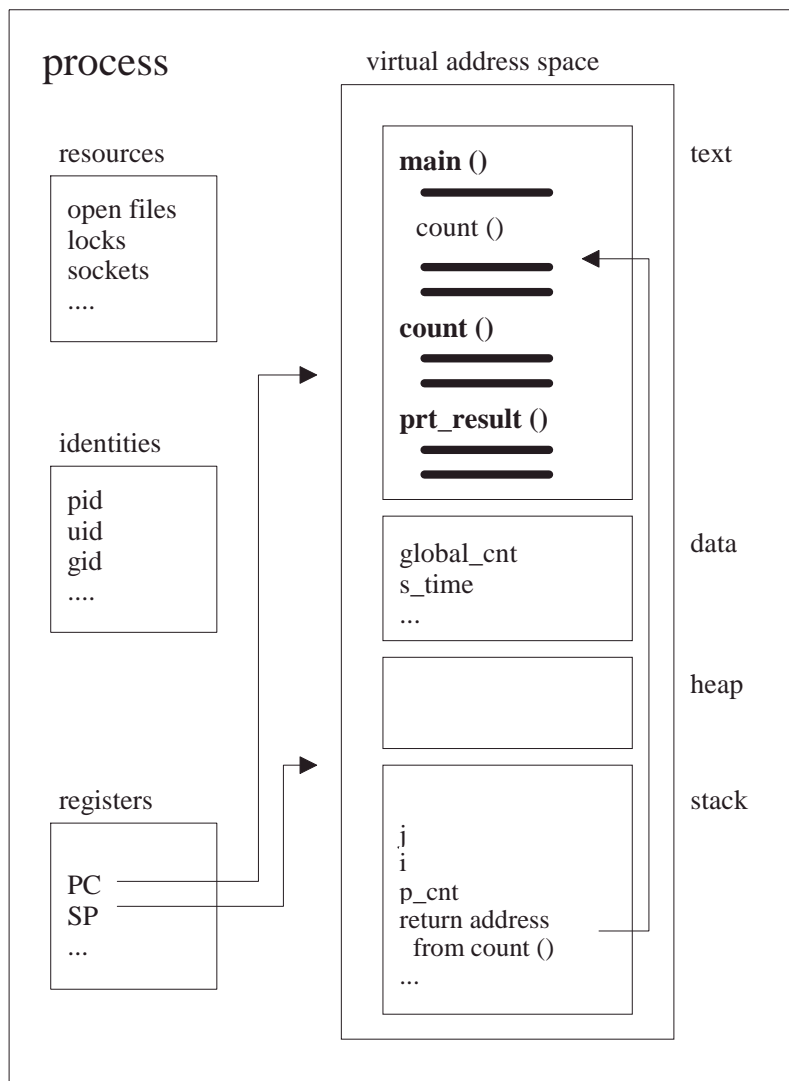
...

Value of 0. counter: 20000000
 Value of 1. counter: 20000000
 Value of 2. counter: 20000000
 Sum of all local counters: 60000000
 Value of global counter (should equal the above sum): 60000000

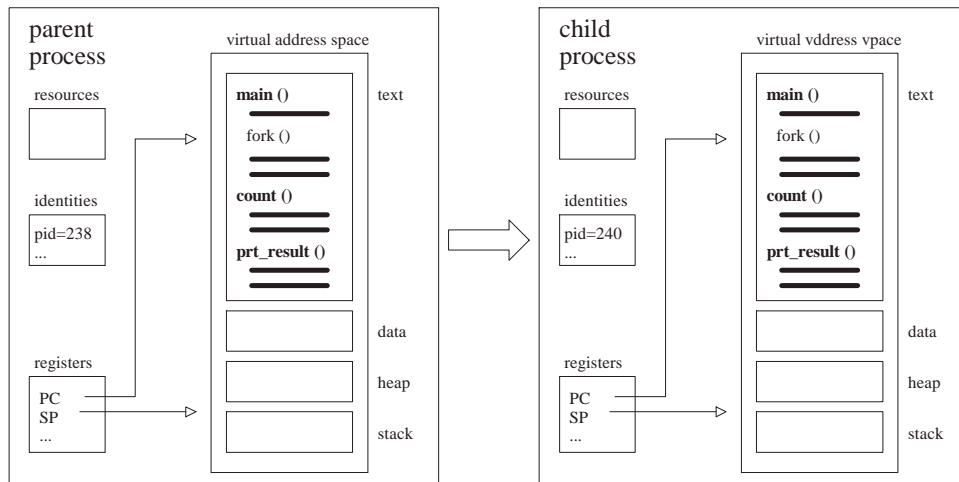
Duration: 1.0 seconds

Used CPU time: 0.3 seconds

- Struktur des Prozesses



- parallele Prozesse ohne Synchronisation (count_parallel?.c)



```

...
for (i = 0; i < NUM_CLD; ++i)
{
    fork_pid[i] = fork ();                /* create child process */
    switch (fork_pid[i])
    {
        case -1:                          /* error: no process created */
            ...
        case 0:                            /* child process */
            count (&(shm_addr->local_cnt[i]));
            exit (EXIT_SUCCESS);          /* nothing more to do */
        default:                            /* parent process */
            if (i == (NUM_CLD - 1))      /* all childs created ? */
            {                             /* yes -> wait for termination */
                for (j = 0; j < NUM_CLD; ++j)
                {
                    if (waitpid (fork_pid[j], NULL, 0) == (pid_t) -1)
                    {
                        ...
                    }
                }
            }
    }
}
...

void count (long *p_local_cnt)
{
    ...
    for (i = 0; i < MAX_OUT_LOOP; ++i)
    {
        for (j = 0; j < MAX_IN_LOOP; ++j)
        {
            ...
            (*p_local_cnt)++;
            /* try the long way to produce access conflicts resulting in
             * a wrong value. This way it will be obvious that synchroni-
             * zation of concurrent / parallel accesses is needed.
             */
            shm_addr->global_cnt = shm_addr->global_cnt + 1;
        }
    }
}
...

```

- Ausgabe ohne Semaphore

(Pentium M, 1.6 GHz, Cygwin 1.5.25, gcc-3.4.3. Führen Sie das Programm so oft aus, bis ein Fehler auftritt. Manchmal muss die Anzahl der inneren und/oder äußeren Schleifendurchläufe erhöht werden, um einen Fehler zu produzieren, z. B. von 5 auf 40 Millionen.)

Process 2572 is counting ...

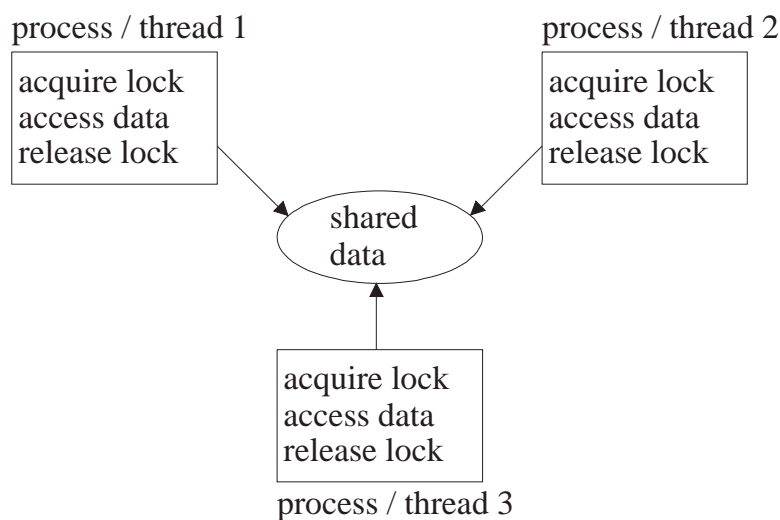
...

Value of 0. counter:	160000000
Value of 1. counter:	160000000
Value of 2. counter:	160000000
Sum of all local counters:	480000000
Value of global counter (should equal the above sum):	409024443

Duration: 4.0 seconds

Used CPU time: 3.9 seconds

⇒ der Zugriff auf gemeinsame Daten **muss** synchronisiert werden



- mit Synchronisation liefert das Programm immer korrekte Ergebnisse
- Laufzeit hängt von der Granulation der synchronisierten Bereiche ab

- kleiner synchronisierter Bereich (count_parallel_1.c)

```

...
void count (long *p_local_cnt)
{
    ...
    for (i = 0; i < MAX_OUT_LOOP; ++i)
    {
        printf ("Process %ld is counting ...\n", (long) getpid ());
        for (j = 0; j < MAX_IN_LOOP; ++j)
        {
            #ifdef USE_SEM
                /* fine-grained locking results in poor performance, because
                 * the execution time for "semop" is much larger than that for
                 * the increment-operations and the function doesn't do any-
                 * thing else.
                 */
                ret = semop (sem_id, &p, (size_t) 1);
                TestMinusOne (ret, __LINE__, "semop");
                (*p_local_cnt)++;
                (shm_addr->global_cnt)++;
                ret = semop (sem_id, &v, (size_t) 1);
                TestMinusOne (ret, __LINE__, "semop");
            #else
                ...
            #endif
        }
    }
    ...
}
...

```

- großer synchronisierter Bereich (count_parallel_2.c)

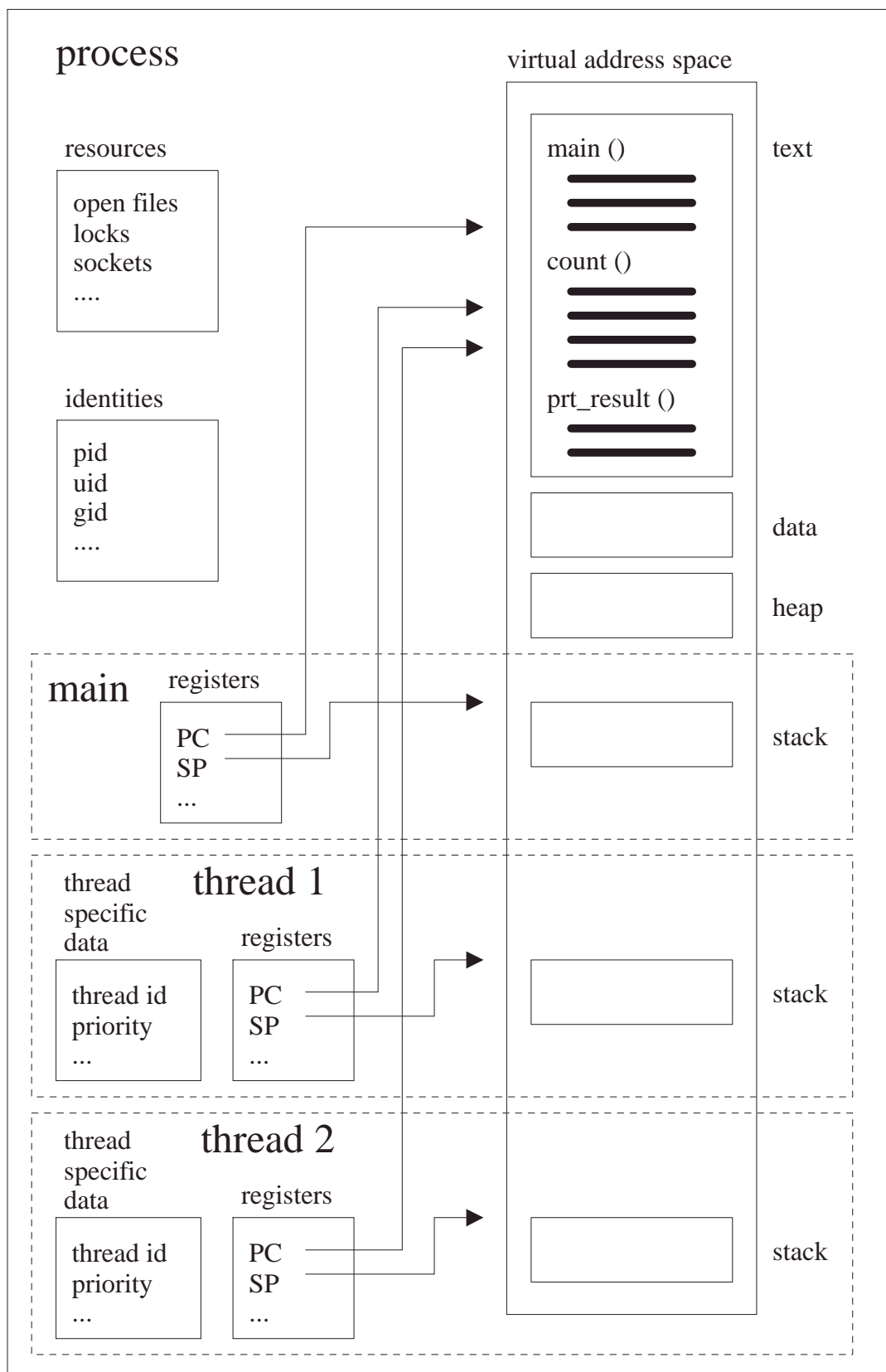
```

...
void count (long *p_local_cnt)
{
    ...
    #ifdef USE_SEM
        /* coarse-grained locking results in better performance, but leads
         * to sequential execution, because the whole work is protected.
         */
        ret = semop (sem_id, &p, (size_t) 1);
        TestMinusOne (ret, __LINE__, "semop");
        for (i = 0; i < MAX_OUT_LOOP; ++i)
        {
            printf ("Process %ld is counting ...\n", (long) getpid ());
            for (j = 0; j < MAX_IN_LOOP; ++j)
            {
                (*p_local_cnt)++;
                (shm_addr->global_cnt)++;
            }
        }
        ret = semop (sem_id, &v, (size_t) 1);
        TestMinusOne (ret, __LINE__, "semop");
    #else
        ...
    #endif
}
...

```

⇒ das Programm ist für eine Parallelverarbeitung nicht geeignet

- Benutzung von *Threads* anstelle von parallelen Prozessen



- es gibt sehr viele verschiedene *Thread*-Schnittstellen
 - ◆ UNIX International Threads (SunOS)
 - ◆ C Threads (Mach)
 - ◆ DCE Threads (Open Software Foundation, Draft 4 der Pthreads)
 - ◆ Pthreads (POSIX 1003.1c-1995)
 - ◆ Windows NT Threads
 - ◆ OS/2 Threads
 - ◆ ...

- jeder *Thread* besteht aus einer Folge von Anweisungen, die unabhängig voneinander ausgeführt werden können

- jeder *Thread* besitzt einen eigenen Programmzähler, einen eigenen Registerbelegungssatz und einen eigenen *Stack*
 - ⇒ *Threads* können unabhängig voneinander ausgeführt werden

- *Threads* teilen sich den Adressraum und die anderen Betriebsmittel des umfassenden Prozesses
 - ⇒ falls ein *Thread* globale Daten ändert, kann die Änderung von allen anderen *Threads* gesehen werden
 - ⇒ falls ein *Thread* *exit()* aufruft, werden **alle** *Threads* beendet
 - ⇒ falls ein *Thread* eine Datei öffnet, kann ein anderer *Thread* sie lesen oder beschreiben
 - (sie können sich in unvorhersehbarer Weise beeinflussen, wenn das Programm nicht sehr sorgfältig erstellt wurde)

- Lösung mit POSIX *Threads* (count_with_threads.c)

```

#define _REENTRANT                /* must precede any "#include"!/
#ifdef ANSI_C
#define _POSIX_C_SOURCE 199506L    /* standard: June 1995      */
#else
#define _POSIX_C_SOURCE 200112L    /* standard: December 2001 */
#endif
...
#include <pthread.h>
...
#ifdef USE_MTX
pthread_mutex_t LockCnt;
#endif
...
int main (int argc, char *argv[])
{
    ...
pthread_attr_t attr;                /* attributes for new threads */
    ...
#ifdef USE_MTX
ret = pthread_mutex_init (&LockCnt, NULL);
TestNotZero (ret, __LINE__, "pthread_mutex_init");
#endif
ret = pthread_attr_init (&attr);
TestNotZero (ret, __LINE__, "pthread_attr_init");
ret = pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE);
TestNotZero (ret, __LINE__, "pthread_attr_setdetachstate");
#if defined(BOUNDED) && !defined(Cygwin)
ret = pthread_attr_setscope (&attr, PTHREAD_SCOPE_SYSTEM);
TestNotZero (ret, __LINE__, "pthread_attr_setscope");
#endif /* create concurrent threads for counting */
    ...
for (i = 0; i < NUM_THR; ++i)
{
ret = pthread_create (&thr_id[i], &attr,
                    (void * (*) (void *)) count,
                    (void *) &local_cnt[i]);
TestNotZero (ret, __LINE__, "pthread_create");
}

/* wait for all threads and print result */
for (i = 0; i < NUM_THR; ++i)
{
ret = pthread_join (thr_id[i], NULL);
TestNotZero (ret, __LINE__, "pthread_join");
}
    ...
/* clean up all things */
ret = pthread_attr_destroy (&attr);
TestNotZero (ret, __LINE__, "pthread_attr_destroy");
#ifdef USE_MTX
ret = pthread_mutex_destroy (&LockCnt);
TestNotZero (ret, __LINE__, "pthread_mutex_destroy");
#endif
return EXIT_SUCCESS;
}

```

```

void count (long *p_local_cnt)
{
    ...
    for (i = 0; i < MAX_OUT_LOOP; ++i)
    {
        printf ("Process %ld: Thread %ld is counting ...\n",
                (long) getpid (), thr);
        for (j = 0; j < MAX_IN_LOOP; ++j)
        {
            #ifdef USE_MTX
                ret = pthread_mutex_lock (&LockCnt);
                TestNotZero (ret, __LINE__, "pthread_mutex_lock");
                (*p_local_cnt)++;
                global_cnt++;
                ret = pthread_mutex_unlock (&LockCnt);
                TestNotZero (ret, __LINE__, "pthread_mutex_unlock");
            #else
                (*p_local_cnt)++;
                /* try the long way to produce access conflicts resulting in
                 * a wrong value. This way it will be obvious that synchroni-
                 * zation of concurrent / parallel accesses is needed.
                 */
                global_cnt = global_cnt + 1;
            #endif
        }
    }
    ...
}
...

```

- Ausgabe des Programms (Pentium M, 1.6 GHz, Cygwin 1.5.25, gcc-3.4.3)

```

Process 1972: Thread 0 is counting ...
Process 1972: Thread 0 is counting ...
Process 1972: Thread 0 is counting ...
Process 1972: Thread 0 is counting ...
Process 1972: Thread 1 is counting ...
Process 1972: Thread 1 is counting ...
Process 1972: Thread 1 is counting ...
Process 1972: Thread 1 is counting ...
Process 1972: Thread 2 is counting ...
Process 1972: Thread 2 is counting ...
Process 1972: Thread 2 is counting ...
Process 1972: Thread 2 is counting ...

```

```

Value of 0. counter:          20000000
Value of 1. counter:          20000000
Value of 2. counter:          20000000
Sum of all counters:          60000000
Value of global counter (should equal the above sum): 60000000

```

```

Duration:      0.0 seconds
Used CPU time: 0.3 seconds

```

- Zusammenfassung (alle Zeiten (*used cpu time*) in Sekunden)

	count_sequential	count_parallel_1 (synchronized)	count_parallel_2 (synchronized)	count_with_threads (synchronized)
Apple – Mac mini, Intel Core 2 duo, 2.0 GHz, Darwin 9.6.0, gcc 4.0.1	0.2	425	0.3	848
Sun Fire X2100, AMD Opteron dual core, 2.6 GHz, Solaris 10 x86, Sun C 5.9	0	46	0	4
Sun Fire X2100, AMD Opteron dual core, 2.6 GHz, Linux 2.6.18, gcc 4.2.0	0.6	199	0.7	33
Sun Fire V480, 4 UltraSparc III processors, 900 MHz, Solaris 10, Sun C 5.9	0	1252	0	118
Sun Fire V480, 4 UltraSparc III processors, 900 MHz, Solaris 10, gcc 4.2.0	1.3	1184	1.3	68
Sun Ultra 45, 2 UltraSparc IIIi processors, 1.6 GHz, Solaris 10, Sun C 5.9	0	545	0	43
Acer TravelMate 800, Intel Pentium M, 1.6 GHz, Cygwin 1.5.25 on top of Windows XP Professional SP3, gcc 3.4.4	0.3	5113 (wall clock time: 23363, nearly 7 hours !)	0.6	40 (wall clock time: 41)

- warum sind *Threads* schneller als parallele Prozesse?
 - ein Prozess besteht im Wesentlichen aus folgenden Teilen
 - ◆ Code (text segment)
 - ◆ Daten (data segment)
 - ◆ Stapelspeicher (stack segment)
 - ◆ Dateitabelle (file I/O table)
 - ◆ Unterbrechungsvektor-Tabelle (signal table)
 - ◆ Hardware-Kontext (CPU-Register, ...)
 - nach einem *fork()* kann nur der Code gemeinsam benutzt werden
 - bei einem Prozesswechsel müssen alle Prozessdaten ausgetauscht werden
 - Prozesse können nur über *Pipes*, *Shared Memory* oder ähnlich teure Mechanismen Informationen austauschen
(bei jeder Aktion ist das Betriebssystem beteiligt)
 - *Threads* reduzieren diesen Verwaltungsaufwand erheblich, indem sie zusätzlich folgende Teile gemeinsam nutzen
 - ◆ Datenbereich (data segment)
 - ◆ Dateitabellen
 - ◆ offene Dateien
 - ◆ Unterbrechungsvektor-Tabelle
 - ◆ Prozess-ID (nicht bei LinuxThreads)
 - ⇒ *Thread*-Wechsel erfolgen viel effizienter
 - ⇒ Informationen können einfach gemeinsam benutzt werden
(das Betriebssystem ist bei diesen Aktionen **nicht** beteiligt)

- Dauer für 12.000.000 Prozess-/Thread-Wechsel in Sekunden
(einzige Anweisungen der Prozesse/Threads: *for*-Schleife mit *sched_yield()*, *sched_fork.c*, *sched_thr.c*)

	Prozesse	Threads
Apple – Mac mini, Intel Core 2 duo, 2.0 GHz, Darwin 9.6.0, gcc 4.0.1, 32-bit program	11	11
Apple – Mac mini, Intel Core 2 duo, 2.0 GHz, Darwin 9.6.0, gcc 4.0.1, 64-bit program	6	5
Sun Fire X2100, AMD Opteron dual core, 2.6 GHz, Solaris 10 x86, Sun C 5.9	8	8
Sun Fire X2100, AMD Opteron dual core, 2.6 GHz, Linux 2.6.18, gcc 4.2.0	3	3
Sun Fire V480, 4 UltraSparc III processors, 900 MHz, Solaris 10, Sun C 5.9	15	15
Sun Ultra 45, 2 UltraSparc IIIi processors, 1.6 GHz, Solaris 10, Sun C 5.9	14	14
Acer TravelMate 800, Intel Pentium M, 1.6 GHz, Cygwin 1.5.25 on top of Windows XP Professional SP3, gcc 3.4.4	30	20

- es gibt unterschiedliche *Thread*-Implementierungen
 - ◆ separate Prozess- und *Thread*-Tabellen mit eigenem *Scheduler*
 - ⇒ was passiert, wenn ein *Thread fork()*, *execvp()* oder etwas ähnliches aufruft?
 - a) der ganze Prozess mit allen *Threads* wird dupliziert bzw. ersetzt
 - b) nur der *Thread*, der die Funktion aufruft, wird dupliziert bzw. ersetzt

(POSIX fordert: Der *fork*-Aufruf erzeugt einen Prozess mit einem *Thread*. In einem Programm mit mehreren *Threads* soll der neue Prozess nur eine Kopie des die Funktion aufrufenden *Threads* inkl. der ganzen Umgebung sein. Der Aufruf einer beliebigen *exec*-Funktion soll das Ende aller *Threads* zur Folge haben. Es wird nur noch das neue Programm ausgeführt.)
 - ◆ eine gemeinsame Prozess-/Thread-Tabelle mit einem gemeinsamen *Scheduler*

- Warum wurden *Threads* eingeführt?
 - 1) effizientere Bearbeitung eines Programms in Mehrprozessorsystemen mit *kernel-level threads*
(auf Einprozessorsystemen sind diese Programme im Allgemeinen langsamer als ein traditionelles Programm, da die Verwaltung der *Threads* Zeit kostet)

 - 2) Anwendungsprogramme können häufig einfacher entwickelt werden, wenn die einzelnen Aufgaben als in sich geschlossene Teile implementiert werden können
(moderne Fensteroberflächen bieten z. B. viele Funktionen über sogenannte *Widgets* oder *Icons* an, die separat entwickelt werden können)

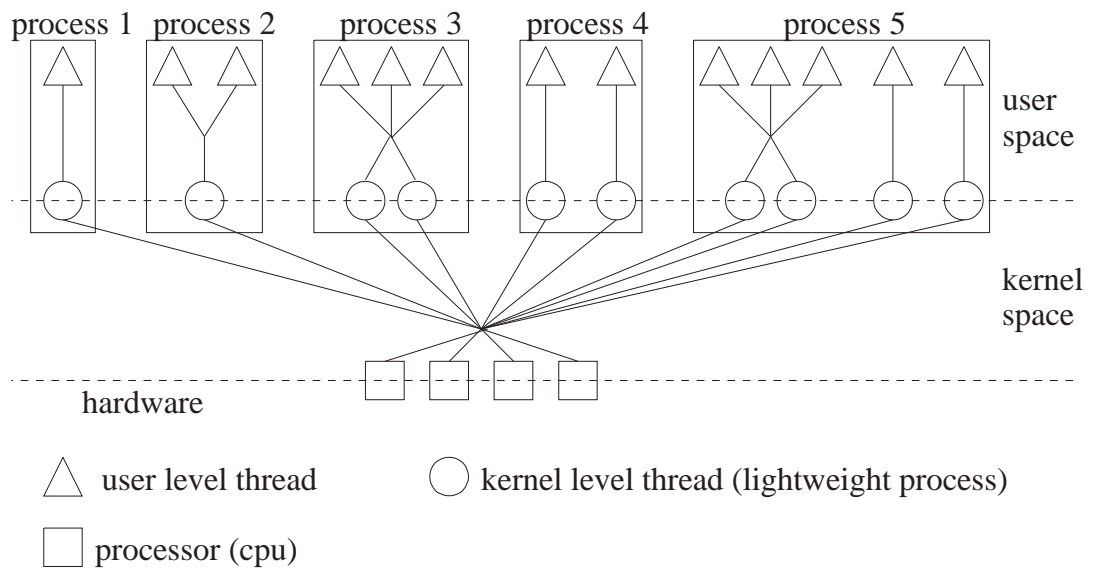
- *user-level threads*
 - sind dem Betriebssystemkern unbekannt
 - werden vollständig im Adressraum des Programms (*user space*) verwaltet (*Thread-Tabelle, Scheduling*)
 - verbrauchen daher keine Ressourcen des Betriebssystemkerns
 - Wechsel zwischen *Threads* ist sehr effizient
 - gut geeignet, wenn die *Threads* meistens auf Arbeit warten
 - können nur den einen Prozessor ihres Prozesses benutzen
 - falls ein *Thread* bei einer Ein-/Ausgabe blockiert wird, wird im Allgemeinen der ganze Prozess mit allen *Threads* blockiert
 - ein *Thread*, der keine Funktion der *Thread*-Bibliothek aufruft und dauernd rechenwillig ist, kann u. U. alle Zeitscheiben des Prozesses alleine verbrauchen, so dass alle anderen *Threads* an ihrer Arbeit gehindert werden

- *kernel-level threads*
 - werden im Adressraum des Betriebssystems verwaltet (*kernel space*)
 - verbrauchen Ressourcen des Betriebssystemkerns
(jeder Prozess hat eine *Thread*-Tabelle oder jeder *kernel-level thread* belegt einen Eintrag in der Prozesstabelle)
 - *Thread*-Wechsel im Allgemeinen effizienter als Prozesswechsel, aber schlechter als bei *user-level threads*
 - können alle Prozessoren eines Multiprozessorsystems benutzen
 - falls ein *Thread* bei einer Ein-/Ausgabe blockiert wird, führt dies zu keiner Blockade der anderen *kernel-level threads*
 - das Betriebssystem entscheidet, welcher *Thread* als nächster ausgeführt wird
 - ⇒ ein *Thread* kann nicht alle anderen *Threads* an der Arbeit hindern
 - ⇒ *kernel-level threads* sind für dauernd rechenwillige *Threads* gut geeignet

- *Thread-Scheduling*
 - *1 : 1 Scheduling*
 - ◆ ein *user-level thread* pro *kernel-level thread*
(siehe Prozess 1 und 4 in der zusammenfassenden Grafik weiter unten)
 - ◆ Vor- und Nachteile der *kernel-level threads*
 - ◆ dieses Modell (`PTHREAD_SCOPE_SYSTEM`, *bounded threads*) wird z. B. von *Microsoft Windows*, *OS/2*, *LinuxThreads*, *Linux Native POSIX Thread Library* (NPTL) und *SunOS* (ab *Solaris 9*) benutzt
 - ◆ da jeder *kernel-level thread* Betriebsmittel des Betriebssystemkerns belegt, kann nur eine begrenzte Anzahl *Threads* erzeugt werden
 - *n : 1 Scheduling*
 - ◆ viele *user-level threads* pro *kernel-level thread* (siehe Prozess 2)
 - ◆ Vor- und Nachteile der *user-level threads*
 - ◆ dieses Modell (`PTHREAD_SCOPE_PROCESS`, *unbounded threads*) wurde/wird z. B. von *IRIX* benutzt
 - ◆ die Anzahl der *Threads* ist theoretisch nur durch den virtuellen Speicher begrenzt

- $m : n$ *Scheduling* (2-level scheduling)
 - ◆ viele *user-level threads* werden auf vielen *kernel-level threads* ausgeführt (siehe Prozess 3)
 - ◆ Vorteile der *user-level* **und** *kernel-level threads*
 - ◆ die Anzahl der *kernel-level threads* ist im Allgemeinen wesentlich kleiner als die Anzahl der *user-level threads*
 - ⇒ das *Scheduling* erfolgt hauptsächlich im *user space*
 - ⇒ spart Ressourcen des Betriebssystemkerns
 - ◆ die Anzahl der *kernel-level threads* passt sich an die Anforderungen des Prozesses an
 - ⇒ ein Prozess beginnt im Allgemeinen mit einem *kernel-level thread*
 - ⇒ falls es viele aktive *user-level threads* gibt, erzeugt das Betriebssystem nach einiger Zeit weitere *kernel-level threads*
 - ⇒ das Betriebssystem erzeugt so viele *kernel-level threads* wie benötigt werden
(es kann passieren, dass jeder *user-level thread* nach einiger Zeit seinen eigenen *kernel-level thread* hat und sich damit ein $1 : 1$ *Scheduling* ergibt)
 - ⇒ unbenutzte *kernel-level threads* werden nach einiger Zeit wieder entfernt
 - ◆ Modell wird z. B. von SunOS (vor Solaris 9), HP Tru64 UNIX (vorher: Compaq Tru64 UNIX, davor: Digital UNIX) und *Linux New Generation POSIX Threads* (ngpt) benutzt

- zusammenfassende Darstellung der *Scheduling*-Verfahren



- jeder Prozess beginnt als normaler *single-threaded* Prozess
- danach kann er einige *user-level threads* erzeugen
- falls das Betriebssystem *2-level Scheduling* unterstützt und es mehrere aktive *user-level threads* gibt, können weitere *kernel-level threads* erzeugt werden
- der Prozess könnte nach seinem Start auch *kernel-level threads* erzeugen
- er könnte auch alle Verfahren kombinieren und würde dann nach einiger Zeit wie Prozess 5 in der obigen Darstellung aussehen

- wie viele *Threads* sollten erzeugt werden?
 - im Allgemeinen macht es keinen Sinn bei n Prozessoren mehr als n *kernel-level threads* pro Prozess zu erzeugen
(würden sich gegenseitig die Prozessoren streitig machen und damit die Leistungsfähigkeit des Programms reduzieren)
 - bei Ein-/Ausgabe-intensiven Programmen kann es Sinn machen mehr *kernel-level threads* zu erzeugen als Prozessoren vorhanden sind, damit nicht der ganze Prozess blockiert wird, wenn ein *Thread* blockiert wird
 - es werden so viele *Threads* erzeugt wie unbedingt notwendig sind
(ein *Internet-Server* wird vielleicht für jeden Verbindungswunsch einen eigenen *Thread* erzeugen und erzeugt dann unter Last vielleicht einige Hundert *Threads*)

- welche Programme eignen sich für *Threads* ?
 - *Server-Programme*
(für jeden *Client* kann ein *Thread* gestartet werden, der alle Anfragen des *Clients* bearbeitet)

 - Anwendungsprogramme, die parallelisiert werden können und eine hohe Leistung erbringen müssen

 - Anwendungsprogramme, bei denen Ein-/Ausgaben und Berechnungen überlappt ausgeführt werden können
(falls ein *Thread* bei einer Ein-/Ausgabe blockiert wird, kann ein anderer rechnen)

 - Anwendungsprogramme, die einfacher und übersichtlicher implementiert werden können, wenn das Problem in Teilprobleme geteilt wird
(*Threads* können das logische Design eines Programms verbessern, z. B. könnten einige *Threads* für die Interaktion mit dem Benutzer über eine grafische Schnittstelle zuständig sein, während andere die eigentliche Arbeit erledigen)

- welche Nachteile hat die Programmierung mit *Threads* ?
 - der Entwicklungsprozess des Programms wird komplexer, speziell im Bereich
 - ◆ Algorithmen und
 - ◆ Synchronisation der Datenzugriffe
 - Programme mit mehreren *Threads* (*multithreaded applications*) sind schwer zu testen
(da einige Fehler nur selten auftreten, sind sie schwer zu reproduzieren)
 - Synchronisationsfehler (*race conditions*) können auftreten, wenn die Datenzugriffe nicht richtig synchronisiert sind
 - Verklemmungen (*deadlocks*) können auftreten, wenn ein Datenzugriff eine Mehrfach-Synchronisation (*multiple lock*) benötigt und keine exakten Regeln für die Synchronisation existieren
(entsprechende Probleme wurden z. B. in der Lehrveranstaltung *Betriebssysteme* diskutiert)
 - es gibt immer noch Umgebungen, die nicht *thread-safe* sind
(z. B. ältere *X Window* Entwicklungsumgebungen)

3.2 POSIX Threads

- POSIX hat viele Optionen, die eine Implementierung unterstützen kann aber nicht unterstützen muss
 - Präfix aller Optionen: `_POSIX_`
 - "man sysconf" liefert die entsprechenden symbolischen Namen
 - weitere Hinweise stehen in `"/usr/include/unistd.h"`
(bei Linux: `"/usr/include/bits/posix_opt.h"`)
- ⇒ im Programm müssen symbolische Namen überprüft werden, um zu entscheiden, ob eine Option implementiert ist

- einige Datentypen (definiert in `/usr/include/pthread.h`)
 - `pthread_t`
 - `pthread_attr_t`
 - `pthread_cond_t`
 - `pthread_mutex_t`
- ⇒ die Realisierung der Typen ist implementierungsabhängig
- ⇒ Komponenten strukturierter Typen dürfen im Allgemeinen nicht direkt benutzt werden
- ⇒ ein Typ, der in einer Implementierung als Basistyp realisiert ist, kann in anderen Implementierungen strukturiert sein
(z. B.: `pthread_t`)

- "POSIX Thread"-Funktionen werden in Funktionsgruppen eingeteilt (die Funktions-Prototypen sind in `/usr/include/pthread.h` definiert; weitere Informationen über "man pthreads")

Präfix der Funktion	Funktionsgruppe
<code>pthread_</code>	verschiedene Funktionen, die in keine der anderen Gruppen hinpassen
<code>pthread_attr_</code>	Attribute der <i>Threads</i> festlegen
<code>pthread_barrier_</code>	Benutzung von <i>Barrier</i> -Variablen
<code>pthread_barrier_attr_</code>	Attribute für <i>Barrier</i> -Variablen festlegen
<code>pthread_cond_</code>	Benutzung von Bedingungsvariablen
<code>pthread_condattr_</code>	Attribute für Bedingungsvariablen festlegen
<code>pthread_key_</code>	<i>Thread</i> -spezifische Daten verwalten
<code>pthread_mutex_</code>	Benutzung von <i>Mutex</i> -Variablen
<code>pthread_mutexattr_</code>	Attribute für <i>Mutex</i> -Variablen festlegen
<code>pthread_rwlock_</code>	Benutzung von Leser-/Schreibersperren
<code>pthread_rwlockattr_</code>	Attribute für Leser-/Schreibersperren festlegen
<code>pthread_spin_</code>	Benutzung von <i>Spin-lock</i> -Variablen

3.2.1 *Thread*-Attribute

- folgende Attribute können für einen *Thread* festgelegt werden:
 - Adresse und Größe des *Stacks*
 - *Scheduling* erfolgt innerhalb des Prozesses (*unbounded thread*) oder systemweit (*bounded thread*)
 - auf das Ende des *Threads* wird gewartet oder nicht gewartet
 - die *Scheduling*-Parameter werden geerbt oder explizit gesetzt
 - die Priorität des *Threads*
 - die *Scheduling*-Strategie
- das Attribut-Objekt muss zuerst initialisiert werden, bevor einzelne Komponenten geändert werden können
- ein Attribut-Objekt kann zur Erzeugung mehrerer *Threads* mit gleichem Verhalten benutzt werden
- Änderungen am Attribut-Objekt haben keinen Einfluss auf bereits erzeugte *Threads*
- ein nicht mehr benötigtes Attribut-Objekt muss freigegeben werden, damit versteckte Speicherbereiche wieder zur Verfügung stehen

- `int pthread_attr_init (pthread_attr_t *attr)`
 - *attr* Zeiger auf ein Attribut-Objekt
 - Rückgabewert: = 0 erfolgreiche Durchführung
 != 0 Fehler

 - initialisiert ein Attribut-Objekt mit Standardwerten
 - ◆ *Thread* ist selbstständig oder auf Ende kann gewartet werden
 - PTHREAD_CREATE_DETACHED
 - PTHREAD_CREATE_JOINABLE (*default*)

 - ◆ Adresse und Größe des *Stacks*

 - ◆ *Scheduling* (abhängig von Implementierung)
 - PTHREAD_SCOPE_PROCESS
 - PTHREAD_SCOPE_SYSTEM

 - ◆ Vererbung/Festlegung der *Scheduling*-Parameter (abhängig von Implementierung)
 - PTHREAD_INHERIT_SCHED
 - PTHREAD_EXPLICIT_SCHED

 - ◆ *Scheduling*-Strategie
 - SCHED_FIFO
 - SCHED_RR
 - SCHED_OTHER

 - ◆ die Priorität des *Threads*
(im Allgemeinen die Priorität des Vater-*Threads*)

 - weitere Einzelheiten stehen in der Handbuchseite

- `int pthread_attr_destroy (pthread_attr_t *attr)`
 - *attr* Zeiger auf ein Attribut-Objekt
 - Rückgabewert: = 0 erfolgreiche Durchführung
 != 0 Fehler
 - gibt ein Attribut-Objekt frei
 - das Objekt kann nicht mehr zur *Thread*-Erzeugung benutzt werden
 - weitere Einzelheiten stehen in der Handbuchseite

- `int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)`
 - *attr* Zeiger auf ein Attribut-Objekt
 - *detachstate* PTHREAD_CREATE_DETACHED

(*Thread* hinterlässt keine Spuren; alle Ressourcen werden automatisch freigegeben und stehen dann wieder zur Verfügung; auf das Ende des *Threads* kann nicht gewartet werden)
 - PTHREAD_CREATE_JOINABLE

(die Ressourcen stehen erst dann wieder zur Verfügung, wenn *pthread_join()* erfolgreich ausgeführt worden ist; auf das Ende des *Threads* **muss** gewartet werden)
 - Rückgabewert: = 0 erfolgreiche Durchführung
 != 0 Fehler
 - legt fest, welche Verbindung der erzeugende *Thread* zum neuen *Thread* hat
 - weitere Einzelheiten stehen in der Handbuchseite

- `int pthread_attr_setscope (pthread_attr_t *attr, int contentionscope)`
 - *attr* Zeiger auf ein Attribut-Objekt
 - *contentionscope* PTHREAD_SCOPE_PROCESS
(erzeugt *unbounded threads (user-level threads)*)
PTHREAD_SCOPE_SYSTEM
(erzeugt *bounded threads (kernel-level threads)*)
 - Rückgabewert: = 0 erfolgreiche Durchführung
 != 0 Fehler
 - legt den *Thread*-Typ fest
 - weitere Einzelheiten stehen in der Handbuchseite

- weitere Funktionen zur Festlegung von *Thread*-Attributen können den Handbuchseiten entnommen werden

- **Aufgabe 3-1:**
Implementieren Sie ein kleines Programm, das die Attribute eines *Threads* ausgibt.

3.2.2 Threads erzeugen und beenden

- die Funktion *main ()* ist selbst ein *Thread*
- `int pthread_create (pthread_t *new_thread_id,
const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg)`
 - *new_thread_id* Zeiger auf eine Variable für die *Thread-ID*
 - *attr* Zeiger auf ein Attribut-Objekt
 - *start_routine* "Programm" für neuen *Thread*
 - *arg* Parameter für *start_routine*
 - Rückgabewert: = 0 erfolgreiche Durchführung
!= 0 Fehler
 - erzeugt einen neuen *Thread*
 - speichert die ID des *Threads* in die Variable, auf die *new_thread_id* zeigt
 - falls *attr* den Wert *NULL* hat, werden Standard-Attribute benutzt
 - *start_routine* ist die Adresse der Funktion, die als *Thread* ausgeführt werden soll
 - *arg* ist ein Verweis auf den Parameter oder eine Parameterstruktur für den neuen *Thread*
 - die Lebensdauer des *Threads* beginnt mit der erfolgreichen Ausführung von *pthread_create*

- die Lebensdauer endet, wenn folgendes eintritt:
 - ◆ normales Ende von *start_routine* ()
 - ◆ Aufruf von *pthread_exit* ()
 - ◆ Aufruf von *pthread_cancel* () durch einen anderen *Thread*
 - ◆ Beendigung des Prozesses durch Aufruf von *exit* ()
 - ◆ Beendigung der Routine *main* ()
- weitere Einzelheiten stehen in der Handbuchseite
- `int pthread_join (pthread_t wait_for, void **status)`
 - *wait_for* ID des *Threads*, auf den gewartet werden soll
 - *status* Zeiger auf Variable für den Ende-Status des *Threads*
 - Rückgabewert: = 0 erfolgreiche Durchführung
!= 0 Fehler
 - wartet auf das Ende eines *Threads*
 - falls *status* den Wert *NULL* hat, ist der *Thread* am Ende-Status des anderen *Threads* nicht interessiert
 - falls der *Thread* nicht existiert oder auf sein Ende nicht gewartet werden kann, wird ein Fehler gemeldet
 - falls mehrere *Threads* auf das Ende desselben *Threads* warten, durchläuft nur einer diese Funktion korrekt und die anderen erhalten die Fehlermeldung *ESRCH*
 - weitere Einzelheiten stehen in der Handbuchseite

- `void pthread_exit (void *status)`
 - *status* Rückgabewert des *Threads*
 - beendet den aufrufenden *Thread*
 - nachdem der *Thread* geendet hat, sind die Werte seiner lokalen Variablen undefiniert
 - ⇒ der Wert einer lokalen Variablen darf nicht als Wert für *status* benutzt werden
 - weitere Einzelheiten stehen in der Handbuchseite

- ein kleines Beispielprogramm (hello_1.c)

```

...
int main (void)
{
    pthread_t thr_id[2];                /* ID's of created threads */
    char *msg1 = "Hello ",
        *msg2 = "World\n";

    pthread_create (&thr_id[0], NULL, (void *(*)(void *)) print_msg,
                   (void *) msg1);
    pthread_create (&thr_id[1], NULL, (void *(*)(void *)) print_msg,
                   (void *) msg2);
    printf ("main() terminates\n");
    return 0;
}

void print_msg (char *msg)
{
    printf ("%s", msg);
    fflush (stdout);
}

```

- Erwartung:

- ◆ der erste *Thread* gibt *Hello* aus und endet
- ◆ der zweite *Thread* gibt *World* aus und endet
- ◆ *main()* gibt seine Meldung aus und endet

- Realität:

```

kea:threads > gcc hello_1.c -lpthread
kea:threads > a.out
Hello main() terminates
World

```

⇒ die Ausführungsreihenfolge der *Threads* ist ohne Synchronisation nicht vorhersagbar

- "verbesserte" Version des Programms (sequentiell!) (hello_2.c)

```

...
int main (void)
{
    pthread_t      thr_id[2];          /* ID's of created threads      */
    pthread_attr_t attr;              /* attributes for new threads   */
    uintptr_t      thr_ret[2];        /* return values                 */
    char *msg1 = "Hello ",
        *msg2 = "World\n";
    int  ret;                          /* return value                  */

    ret = pthread_attr_init (&attr);
    ret = pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE);
    ret = pthread_create (&thr_id[0], &attr,
        (void * (*) (void *)) print_msg,
        (void *) msg1);
    ret = pthread_join (thr_id[0], (void **) &thr_ret[0]);
    ret = pthread_create (&thr_id[1], &attr,
        (void * (*) (void *)) print_msg,
        (void *) msg2);
    ret = pthread_join (thr_id[1], (void **) &thr_ret[1]);
    printf ("main() terminates.\n"
        "Return value from Thread 0: %d\n"
        "Return value from Thread 1: %d\n",
        (int) thr_ret[0], (int) thr_ret[1]);
    ret = pthread_attr_destroy (&attr);
    return EXIT_SUCCESS;
}

int print_msg (char *msg)
{
    printf ("%s", msg);
    fflush (stdout);
    return 10;
}

```

⇒ Ausgabe

```

kea:threads > gcc hello_2.c -lpthread
kea:threads > a.out
Hello World
main() terminates.
Return value from Thread 0: 10
Return value from Thread 1: 10

```

- **Aufgabe 3-2:**

Implementieren Sie ein Programm mit zwei *Threads*. Ein *Thread* erzeugt eine verkettete Liste und liefert als Rückgabewert einen Zeiger auf das erste Element der Liste. Die Anzahl der Elemente wird dem *Thread* als Parameter bei der Erzeugung übergeben. Der andere *Thread* druckt den Inhalt der verketteten Liste aus und gibt den Speicher wieder frei. Benutzen Sie folgende Funktionsprototypen für die *Threads*:

```

struct list *createList (int numElements);
void printList (struct list *head);

```

Vergessen Sie nicht, den Speicher der Listenelemente freizugeben.

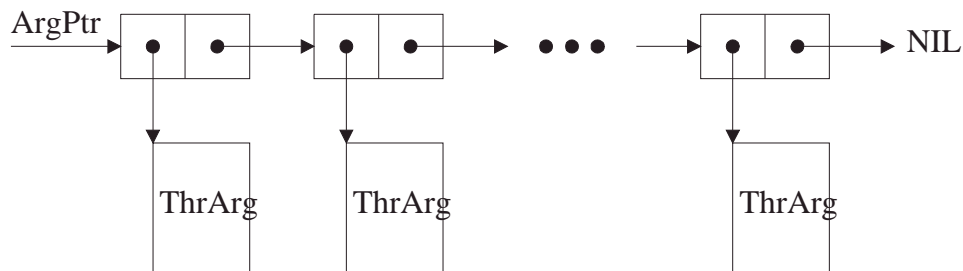
3.2.3 Wechselseitiger Ausschluss

- *Mutex*-Variablen schützen kritische Bereiche
(mutex: **mut**ual **ex**clusion lock)
- *Mutex*-Variablen müssen global definiert werden
- der wechselseitige Ausschluss kann innerhalb eines Prozesses oder prozessübergreifend realisiert werden
- die Attribute für eine *Mutex*-Variable werden mit Hilfe der Funktionen *pthread_mutexattr_** () initialisiert und modifiziert
(diese Funktionsgruppe wird hier nicht benötigt)

- `int pthread_mutex_lock (pthread_mutex_t *mp)`
`int pthread_mutex_unlock (pthread_mutex_t *mp)`
 - *mp* Zeiger auf eine *Mutex*-Variable
 - Rückgabewert: = 0 erfolgreiche Durchführung
 != 0 Fehler
 - sperrt die *Mutex*-Variable bzw. gibt sie frei
 - falls die Variable bereits gesperrt ist, wird der *Thread* blockiert, bis die Variable wieder freigegeben wird
 - falls ein *Thread* eine von ihm gesperrte *Mutex*-Variable noch einmal sperren will, kann dies zu einer Verklemmung führen
 - falls ein *Thread* eine *Mutex*-Variable freigeben will, die er nicht selbst gesperrt hat, ist das Verhalten des Programms undefiniert (Solaris liefert in diesem Fall eine Fehlermeldung)
 - weitere Einzelheiten stehen in der Handbuchseite

Aufgabe 3-3:

Berechnen Sie die Mandelbrotmenge mit Hilfe von POSIX *Threads*. Verwenden Sie als Strategie das *Workpile*-Modell. Das Hauptprogramm nimmt zunächst eine Parkettierung des ausgewählten Bereichs vor und speichert die erforderlichen Parameter zur Berechnung der einzelnen Teilbereiche in der Struktur *ThrArg* ab. Die einzelnen Parametersätze werden ihrerseits in einer verketteten Liste verwaltet. Benutzen Sie bitte die folgende Datenstruktur.



Danach startet das Hauptprogramm die *Threads* zur Berechnung der Mandelbrotmenge. Jeder *Thread* holt sich aus der Liste einen Parametersatz, berechnet den Farbwert für jeden Punkt und gibt den Punkt dann auf dem Bildschirm aus. Ein *Thread* beendet sich selbst, wenn keine weiteren Teilbereiche bearbeitet werden müssen.

Einige Hinweise zur Lösung der Aufgabe:

Benoît Mandelbrot hat die komplexe Funktion $f(z) = z^2 + c$ untersucht. Wenn $c = 0 + 0i$ gewählt wird, ergeben sich beim Startwert z_0 für die iterierten Werte $z_0, z_0^2, z_0^4, z_0^8, \dots$ die folgenden drei Möglichkeiten:

- 1) Wenn " $|z_0| < 1$ " ist, konvergiert die Folge gegen Null (*zero attractor*).
- 2) Wenn " $|z_0| > 1$ " ist, divergiert die Folge (*infinity attractor*).
- 3) Wenn " $|z_0| = 1$ " ist, liegen alle Werte auf dem Einheitskreis (die sogenannte *Julia Menge*)

Wenn für $c = p + qi$ ein beliebiger komplexer Wert ungleich Null gewählt wird, gibt es immer noch die obigen drei Möglichkeiten für die Iterationsfolge. Allerdings konvergiert die Folge dann nicht mehr gegen Null und die Grenze zwischen Konvergenz und Divergenz ist nicht mehr glatt.

Die Struktur der Attraktoren wird durch Farben dargestellt. Alle Anfangswerte, deren Iterationsfolge eine vorgegebene Grenze *BOUND* innerhalb von k Iterationsschritten überschreitet und somit gegen *unendlich* tendiert, werden mit derselben Farbe k dargestellt. Ein Anfangswert, dessen Iterationsfolge die Grenze selbst nach K_MAX Iterationen nicht überschreitet, erhält die Farbe *schwarz*, da seine Iterationsfolge konvergiert oder nur sehr langsam divergiert.

Eine Mandelbrotmenge stellt das Konvergenzverhalten einer komplexen Funktion bei festem Anfangswert $z_0 = x_0 + y_0i$ dar, wobei die Werte von $c = p + qi$ variiert werden. Für bestimmte Para-

meterbereiche von c muss die Anzahl der Iterationen groß genug sein (u. U. $K_MAX > MaxColor$), damit der Bildschirm nicht nur "schwarz" bleibt, da es sehr lange dauert, bis die Werte der Folge die vorgegebene Grenze $BOUND$ überschreiten. Die Anzahl der verfügbaren Farben reicht dann eventuell nicht aus, so dass die Farbe in diesen Fällen als $k \% (MaxColor + 1)$ gewählt werden muss.

$$f(z) = z_{k+1} = z_k^2 + c = (x_k + y_k i)^2 + (p + qi) \quad \text{liefert}$$

$$\begin{aligned} \text{für den Realteil:} \quad & x_{k+1} = x_k^2 - y_k^2 + p \quad \text{und} \\ \text{für den Imaginärteil:} \quad & y_{k+1} = 2x_k y_k + q. \end{aligned}$$

einige Wertebereiche für c :

p_min	p_max	q_min	q_max
-2.25	0.75	-1.5	1.5
-0.19920	-0.12954	1.01480	1.06707
-0.713	-0.4082	0.49216	0.71429
-1.251	-1.261	0.376	0.386
-0.95	-0.88333	0.23333	0.3
-1.781	-1.764	0.0	0.013
-0.75104	-0.7408	0.10511	0.11536
-0.74758	-0.74624	0.10671	0.10779

Benutzen Sie für die Tabelle z. B. folgende Datenstruktur:

```
struct fractal_regions { double p_min, p_max,
                        q_min, q_max;
} fractal[] = {...};
```


Für alle Bildschirmpunkte (np, nq) werden die folgenden Schritte ausgeführt:

Schritt 1 (Anfangsinitialisierung):

$p0 = p_min + np * delta_p$ aktuelle reelle Koordinaten
 $q0 = q_min + nq * delta_q$
 $k = 0$ Schleifenvariable
 $xk = 0.0$
 $yk = 0.0$

Schritt 2 ((k+1)-ten Wert bestimmen):

$xk1 = xk * xk - yk * yk + p0$ vergessen Sie nicht, xk und yk am Ende jeder Iteration
 $yk1 = 2 * xk * yk + q0$ zu aktualisieren
 $k = k + 1$

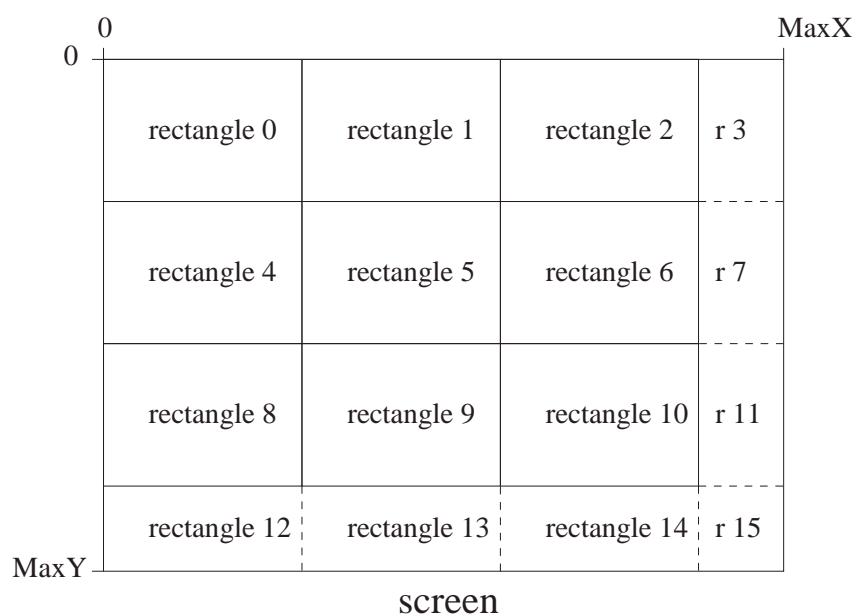
Schritt 3 (Farbe bestimmen):

$r = xk1 * xk1 + yk1 * yk1$
 $r > BOUND$ wähle Farbe " $k \% (MaxColor + 1)$ ", weiter mit Schritt 4
 $k == K_MAX$ wähle Farbe *schwarz*, weiter mit Schritt 4
weiter mit Schritt 2

Schritt 4 (Punkt zeichnen):

zeichne (np, nq) mit der ermittelten Farbe
weiter mit Schritt 1 für nächsten Bildschirmpunkt

Beachten Sie, dass Ihre Rechteckseiten im Allgemeinen keine ganzzahligen Teiler von (MaxX + 1) und (MaxY + 1) sind.



Zur Lösung der Aufgabe können die beiden Dateien *vga.h* und *gra_x11.c* benutzt werden, die sich im Programmarchiv *prog.zip* bzw. *prog.tar.gz* zu dieser Lehrveranstaltung befinden. In diesen Dateien sind die notwendigen Konstanten und Funktionen für die Grafikausgabe definiert. Das Programm wird dann z. B. mit dem Kommando

```
(g)cc -o fraktal_thr fraktal_thr.c gra_x11.c -lpthread -lX11
```

übersetzt. Der Benutzer darf die zu zeichnende Mandelbrotmenge aus der obigen Tabelle über die Kommandozeile auswählen.

Struktur des Programms (siehe auch "draw_line.c" im Programmarchiv):

Deklarationen und Bildauswahl

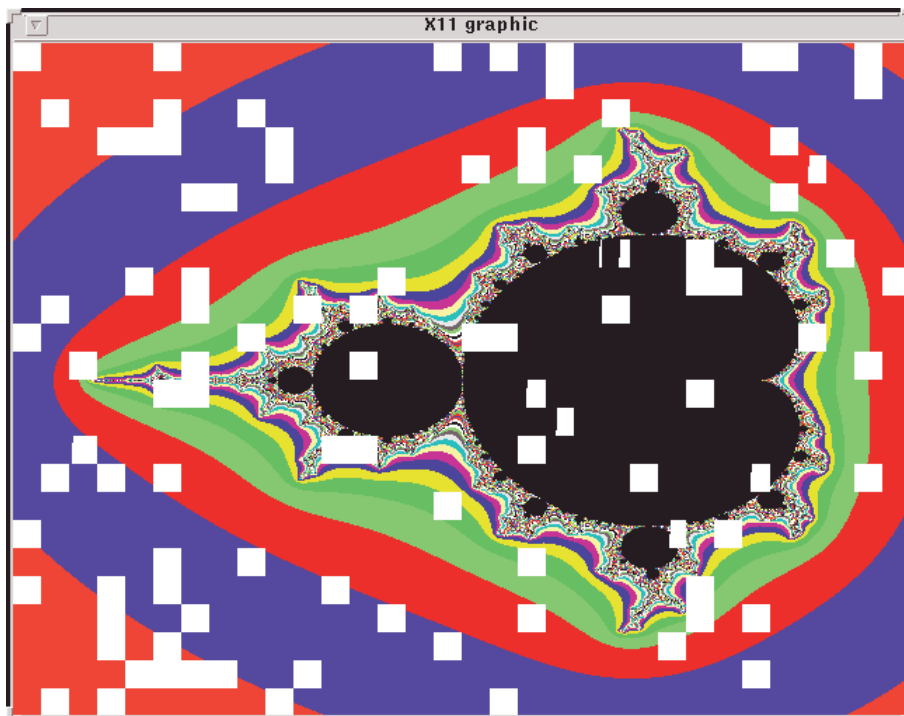
```
InitGraphic (...);
```

```
Mandelbrotmenge berechnen und darstellen ( SetPixel (...); )
```

```
wait_for_input ( );
```

```
CloseGraphic ( );
```

Die Ausgabe soll dann z. B. für den ersten Parametersatz folgendermaßen aussehen:



Aufgabe 3-4:

Auf einem Multiprozessorsystem kann es vorkommen, dass sich die einzelnen *Threads* bei der Ausgabe der Punkte gegenseitig behindern. Erweitern Sie das Programm um einen *Thread*, der für die Ausgabe der Punkte zuständig ist. Die Kommunikation zwischen den Rechen-*Threads* und dem Ausgabe-*Thread* soll nach dem *Workpile*-Modell erfolgen. Das Hauptprogramm erzeugt wieder eine Parkettierung des ausgewählten Bereichs, wobei es die Anzahl der Arbeitsbereiche nicht zählt, so dass der Ausgabe-*Thread* nicht weiß, wann seine Arbeit beendet ist (seine Warteschlange kann leer sein, obwohl noch Rechen-*Threads* arbeiten). Aus diesem Grund muss er durch das Hauptprogramm beendet werden, sobald alle Rechen-*Threads* ihre Arbeit beendet haben und alle Ausgaben erfolgt sind. Zur Lösung des Problems sollen Funktionen der Gruppe *Thread Cancellation* und Bedingungsvariablen benutzt werden. Der Ausgabe-*Thread* benutzt `pthread_cond_wait()`, wenn keine Arbeit vorhanden ist. Ein Rechen-*Thread* signalisiert neue Arbeit mit `pthread_cond_signal()`. Wenn der Ausgabe-*Thread* die Punkte eines Arbeitsbereiches ausgibt, darf er nicht beendet werden, d. h. er muss seine Beendigung vorher mit `pthread_setcancelstate()` verhindern. Sorgen Sie mit Hilfe der Funktionen `pthread_cleanup_push()` und `pthread_cleanup_pop()` dafür, dass alle Betriebsmittel freigegeben werden, wenn der Ausgabe-*Thread* in `pthread_cond_wait()` beendet wird.