Betriebssysteme
Kap. 2: Prozesse und Threads
2.2 Threads

Stand: WS 08/09 (19.11.08)

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

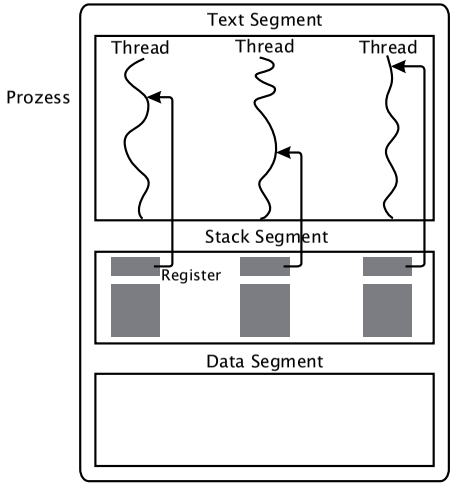
Kap 2.2 Threads of Control

- **Inhalt**
  - Konzept der Threads
  - C-Threads / Pthreads API
  - Multi-Threading
  - User Level / Kernel Level Threads
  - Betriebssysteme für Multiprozessoren / Multicores
  - Threads und Realzeit

Leichte Prozesse: Threads of Control

- Prozess
  - Einheit für Belegung von Ressourcen
  - Einheit für Scheduling / Dispatching
- Motivation für Erweiterung des Prozesskonzepts
  - Effiziente Nutzung von shared-memory Multiprozessoren
    - Nutzung von Parallelität in großen Programmen
  - Effiziente Abbildung natürlicher Nebenläufigkeit
    - Bei Maschinensteuerungen
    - Bei Web Servern
- Lösungen auf Basis von UNIX Prozessen
  - Mehrere Prozesse mit Shared Memory
  - Kommunikation über Signale und shared memory

Speicherbild: Prozess mit Threads



Multithreading

- BS unterstützt mehrere Ausführungseinheiten (**Threads of Control**) innerhalb eines Prozesses
- Prozess
  - Einheit für allozierte Ressourcen und Schutz
    - Memory-Mapping (Virtueller Adressraum)
    - Files
    - I/O
    - IPC


Multithreading

- Thread
  - Ausführungseinheit innerhalb eines Prozesses
  - privater HW-Kontext (Program Counter, Register)
  - privater Stack
  - private Scheduling-Infos
    - (Zustand, Prioritäten)
      evtl. Thread-lokale statische Daten
  - Gemeinsam: Speicher/Ressourcen des Prozesses
    - Code (Text), Daten
    - BS-Ressourcen (Files, Mailboxes, ...)


Multithreading

- Vorteile
  - Programm-Design: Form follows function
  - Thread-Verwaltung schneller als Prozessverwaltung
    - Kein unnötiges Kopieren des Speichers
    - Kein Verlust des TLB bei Kontext-Wechsel
    - Schnellere Synchronisation: Signale brauchen das BS
- Leichter Prozess (**Lightweight Process, LWP**)


C-Thread API (Vorläufer von POSIX)

1. cthread_t cthread_fork (func,arg)
   any_t* func(), any_t* arg;
   Erzeugt neuen thread, der die (Haupt-)Funktion func(arg) ausführt. Das Resultat ist die ID des threads.

2. cthread_exit ()
   Terminiert den laufenden thread. Ein cthread_exit wird implizit am Ende der Hauptfunktion des threads ausgeführt.

3. any_t cthread_join (id)
   cthread_t id;
   Wartet auf das Ende des threads mit ID id. Das Resultat ist das Resultat der Hauptfunktion von thread id.

4. cthread_detach ()
   Deklariert, daß niemand auf das Ergebnis des laufenden thread warten wird.

5. cthread_t cthread_self ()
   Liefert die ID des laufenden thread.

6. cthread_yield ()
   Nimmt freiwillig den laufenden thread von der CPU herunter. (Benutzer Scheduling).


Beispiel ohne Threads

Beispiel: Sortiere (divide & conquer)

```
                Sortiere (A[1...n],n)
/*Sortiere array A von Länge n
 */
{//(1) [Trivialfall.]
  if (n<=1) return;
 //(2) [Teile.] Errechne k und verteile A intern um,
 //     so daß  A[i]≤A[j] ∀ i,j, i≤k<j
 //(3) [Herrsche.]
                    Sortiere (A[1...k],k),
                    Sortiere (A[k+1,...n],n-k);
 //(4) [Ende.]
  return
}
```


Beispiel mit Threads

```
                Sortiere (A[1...n],1,r)

/*Sortiere Array A[1..n] im Intervall [1,r]
 */
{ //(1) [Trivialfall.] Sortiere A[1..n] im Intervall [1,r]
  if (r<=1) return;
  //(2) [Teile.] Errechne k, l≤k<r und verteile A intern um,
  //    so daß  A[i]≤A[j]  ∀ i,j,i≤k<j  und l≤k<r
  //(3) [Herrsche parallel.]
       { any_t a[3]; cthread_t id;
         a[0] = A[1..n]; a[1] = l; a[2]=k
         id = cthread_fork (Sortiere_jacket, a);
         Sortiere (A[1..n],k+1,n);
  //(4) [Ende.]
         cthread_join(id);
       }
}
         Sortiere_jacket(arg)
  any_t arg[3];
{        Sortiere (arg[0], arg[1], arg[2]); }
```


POSIX Threads (Pthreads)

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX-like operating systems (Solaris, Linux, Mac OS X)

- **Int pthread_create (..., func, args);**
- **void pthread_exit (void *status);**
- **int pthread_join (int thr_id, void **status);**

- **int pthread_detach (int thr_id);**
- **int pthread_cancel (int thr_id);**
- **int pthread_kill (int thr_id, int sig);**


User-Level Threads

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Gesamte Thread-Verwaltung in User-Library
  - thread_create,
  - thread_join
- BS sieht nur den Prozess
  - Nicht auf spezielle BS-Funktionalität angewiesen
- Thread-Wechsel (**yield**) nur an vorgesehenen Stellen, ohne System Call (--> schnell)
- Blockierender Thread System Call blockiert den Prozess und daher alle anderen Threads ebenfalls
  - Test, ob Ressource bereit, andernfalls yield


User-Level Threads

- Kein Scheduling durch BS
  - Scheduling durch User-Level Paket, evtl. applikationsspezifisch
- Verwendung von mehreren Prozessoren nur bei Abbildung auf mehrere Prozesse
- Implementation
  - Thread Control Block als struct
  - Pro Thread ein Stack-Segment
  - Kontextwechsel über setjmp / longjmp


Kernel-Level Threads

- Kernel verwaltet Threads (inkl. Scheduling)
  - Applikation in 1 Prozess läuft auf mehreren Prozessoren
  - Keine Prozess-Blockade durch blockierenden Thread
- Beispiele
  - Mach (und daher MacOS X)
  - Modernes UNIX (seit Solaris 2, HP UX 10, AIX, Tru64)
  - OS/2
  - Windows (seit NT)
  - Linux

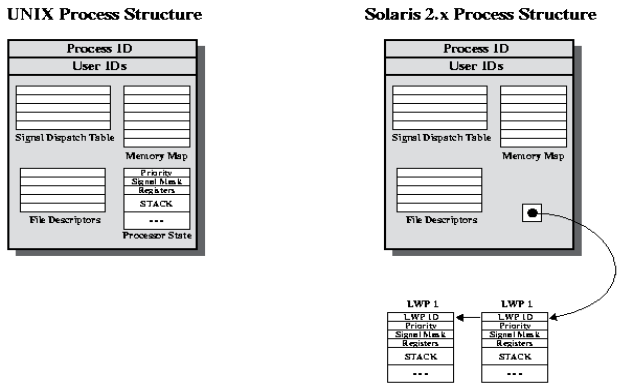



Figure 4.15 Process Structure in Traditional UNIX and Solaris 2.x [LEWI96]

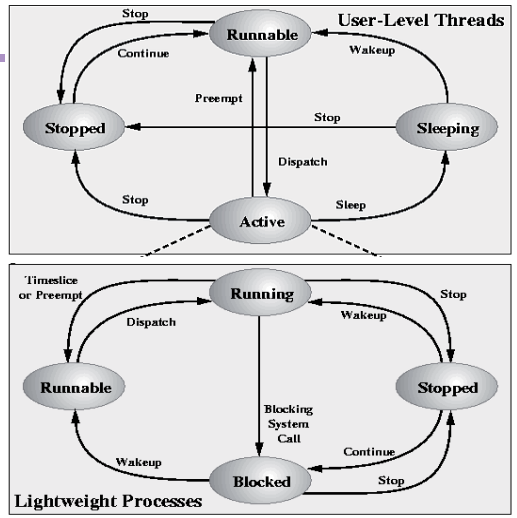



Figure 4.16 Solaris User-Level Thread and LWP States


Vergleich User-Level mit Kernel-Level Threads

- Zeiten für Verwaltung (µs, UNIX-VAX)

	User	Kernel	Prozeß
Null Fork	34	948	11‘300
Signal-Wait	37	441	1‘840

- Pro User-Level
  - Context Switches in User-Level Threads brauchen keinen System Call (Einsparung von 2 Mode- Wechseln)
  - Applikationsspezifisches Scheduling in User-Level Threads
  - Sehr viele Threads möglich (zehntausende)
  - Keine Anforderungen von User-Level Threads an BS

Vergleich User-Level mit Kernel-Level Threads

	User	Kernel	Prozeß
Null Fork	34	948	11‘300
Signal-Wait	37	441	1‘840

- Pro Kernel-Level
  - Kein Blockieren von anderen Threads bei System Calls
  - Ausnutzen von mehreren Prozessoren
  - BS unterstützt Scheduling

User-Level & Kernel-Level Threads

- Linux (> 2.0.33): Sowohl User- als auch Kernel-Threads
  - Kernel-Threads über clone implementiert
    - clone = weiterentwickeltes fork
- Solaris 2: Kombinierter Ansatz
  - Verwendung der Threads über User-Level Threads Paket
  - Schnittstelle zu Kernel Threads: „LWP“ (Solaris Lightweight Process) = Kernel Thread + User Kontext
  - Für jeden Solaris-LWP genau 1 Kernel-Thread
  - Angabe der Zahl zu verwendender LWP‘s oder direktes Binden eines User-Level Threads an LWP

Abbildung User Level → Kernel Level Threads

- Many-to-One
  - Many user-level threads mapped to single kernel thread
  - Examples:
    - Solaris Green Threads
    - GNU Portable Threads
- One-to-One
  - Each user-level thread maps to kernel thread
  - Examples
    - Windows NT/XP/2000
    - Linux
    - Solaris 9 and later
- Many-to-Many

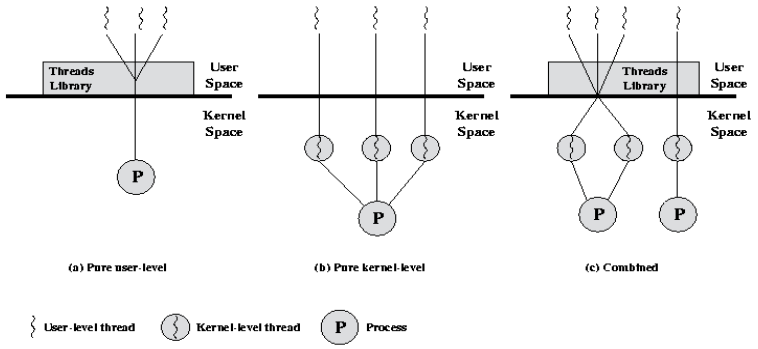
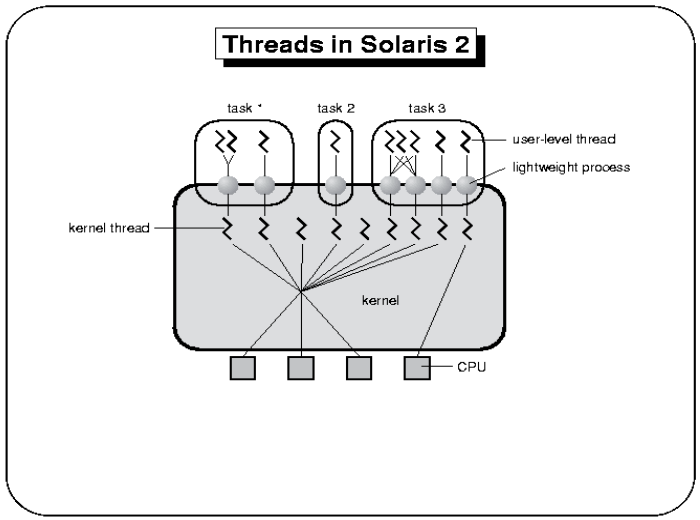


Figure 4.6 User-Level and Kernel-Level Threads



Thread Pools

- Create a number of threads (workers) in a pool where they await work
- Advantages:
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Usually slightly faster to service a request with an existing thread than to create a new thread
- Worker Pool concept works on many levels
  - Threads and work (micro-tasks)
  - Kernel threads and user-level threads
  - processes and user level threads

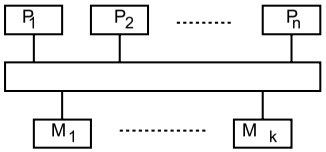
Signals and Threads

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Symmetric Multiprocessing

- Traditionelle Sicht: 1 Prozessor
  - Computer = sequentielle Maschine
- Architekturen mit mehreren Prozessoren
  - SIMD = Single Instruction, Multiple Data (Vektorrechner)
  - MIMD = Multiple Instruction, Multiple Data
    - Verteilter Speicher (Distributed Memory)
    - Gemeinsamer Speicher (Shared Memory)
      - SMP
    - Neu: mehrkernige Prozessoren: 2+ Prozessoren auf einem Chip

MIMD mit SMP (Shared Memory Multiprozessor)



- Standard im Bereich Server und Mainframes
- Windows / Intel: 4 Prozessoren
- UNIXS / RISC: 8 / 16 / 32 Proz., high end bis 64 / 128
- Vervielfacht sich mit mehrkernigen Prozessoren und multi-thread Architekturen
  - SUN (2006): 8 Kerne mit je 4 Threads pro Chip

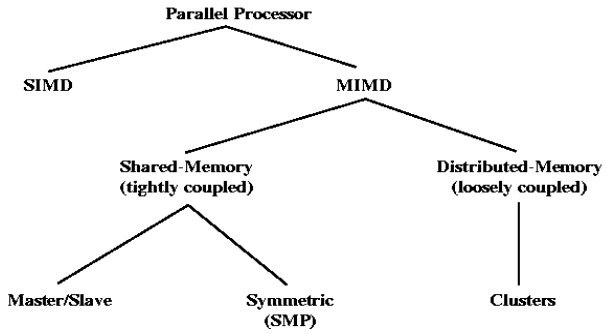


Figure 4.7 Parallel Processor Architectures

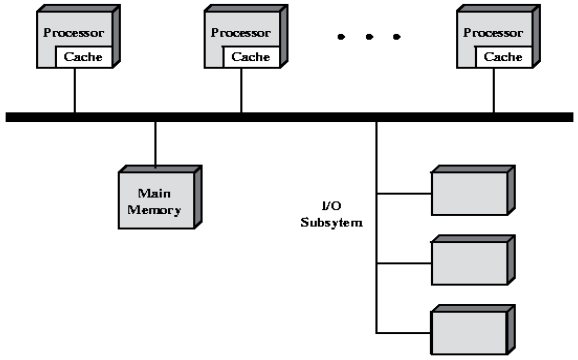


Figure 4.8 Symmetric Multiprocessor Organization

Betriebssystem für SMP

- Gleichzeitiges Bearbeiten von Systemaufrufen
  - Gleichzeitiges Blockieren der interrupts bzw. Systemaufrufe auf n Prozessoren nicht effizient oder nicht möglich
  - Kernel-Code muß **reentrant** sein
- MP Scheduling
  - processor affinity
- Effiziente Synchronisation
- Memory Management
  - **Cache Coherence**
- Vorteile
  - Zuverlässigkeit / Fehlertoleranz
  - Beschleunigung

SMP: Windows NT

- Threads auf alle Prozessoren verteilt
- **Soft Affinity**
  - Dispatcher versucht Thread auf gleichen Prozessor zu setzen
- **Hard Affinity**
  - Feste Bindung Thread zu Prozessor möglich

SMP: Solaris

- Interrupts durch Kernel (System) Threads behandelt
- Prozessor bekommt Interrupt
  - Laufender Thread wird unterbrochen
  - Interrupt handler läuft als system level Thread (aus Free-List)
  - Interrupt Thread wie gewöhnlicher Kernel Thread (ID, Kontext, Stack, ...)
  - Kernel kontrolliert Zugriff über Mutual Exclusion Mechanismen
  - Interrupt Thread hat höhere (Scheduling) Priorität als alle anderen Kernel Threads und kann nur durch Interrupt Thread mit höherer Priorität **preempted** werden

Threads und Realzeit

- Periodische Aufgaben
- A,B,C periodisch messen und behandeln
  - while (TRUE) { A; B; C; }

Zeitanforderungen/ Periodendauer:

A	20 msec
B	40 msec
C	80 msec

Mit Ausführungszeiten

Task	Periode	Laufzeit
A	20 msec	4 msec
B	40 msec	10 msec
C	80 msec	40 msec

Sequentieller Ablaufplan

Zeit	Aufgabe	Dauer
0	A	4
4	B	10
14	C_1	6
20	A	4
24	C_2	16
40	A	4
44	B	10
54	C_3	6
60	A	4
64	C_4	12
76	--	4

- **Probleme**
  - **aufwändiges und kompliziertes Design**
  - **Schlecht wartbar**
  - **Modifikation des Loops nötig: Zerlegung von C**

Aperiodische Aufgaben

- Zusätzliche Aufgabe D bei Alarm

Task	Laufzeit
D	1 msec

- Sofortige Ausführung erwünscht
  - Maximale Latenzzeit: 2msec
- Minimale Zeit zwischen zwei Aktionen
  - 80 msec
- while (TRUE) {
  A1;
  Polling_D;
  A2;
  Polling_D;
  B1;
  Polling_D;
  :
  :
  }

Parallele Lösung

- Jede der Aufgaben A,B,C,D als separater (zyklischer) Prozess / Thread
- Prozesse laufen parallel / nebenläufig
  - A,B,C: while (TRUE) {
    start = gettime();
    A; //resp. B oder C
    used = gettime() - start;
    sleep (periode - used);
    }
  - D: schlafender / blockierender Prozess, der durch externes Signal geweckt wird
- Preemptive Scheduling mit genügend kleiner Zeitscheibe garantiert Prozess-Prioritäten und Einhaltung der Zeitanforderungen.