

Betriebssysteme

Kap. 2: Prozesse und Threads

2.2 Threads

Stand: WS 10/11 (23.11.10)

Prof. Dr. Wolfgang Kuchlin

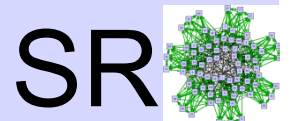
Dipl.-Inform., Dr. sc. techn. (ETH)

**Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät**

Universität Tübingen

**Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)**

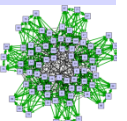
**Wolfgang.Kuechlin@uni-tuebingen.de
<http://www-sr.informatik.uni-tuebingen.de>**



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

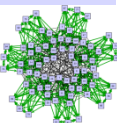
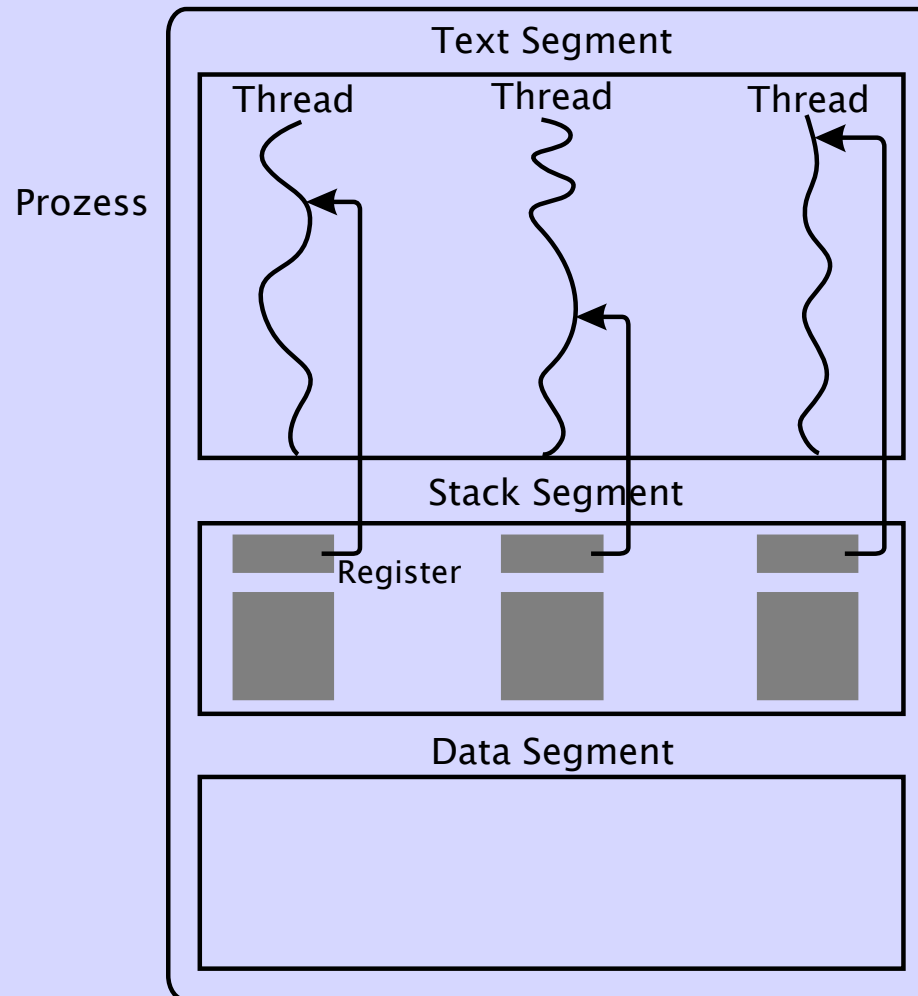
- Effiziente Nutzung von shared-memory Multiprozessoren
 - Nutzung von Parallelität in großen Programmen
- Effiziente Abbildung natürlicher Nebenläufigkeit
 - Bei Maschinensteuerungen
 - Bei Web Servern

➤ Lösungen auf Basis von UNIX Prozessen

- Mehrere Prozesse mit Shared Memory
- Kommunikation über Signale und shared memory

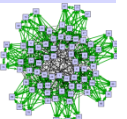


Speicherbild: Prozess mit Threads



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
- Thread
 - Einheit für die Ausführung eines Funktionsaufrufs



Multithreading

➤ Thread

- Ausführungseinheit für einen (nebenläufigen) Funktionsaufruf 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, ...)



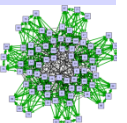
Multithreading

➤ Vorteile

- Programm-Design: direkte Abbildung von Nebenläufigkeit
- Thread-Verwaltung schneller als Prozessverwaltung
 - Kein unnötiges Kopieren des Speichers
 - Kein Verlust des TLB bei Kontext-Wechsel
 - Schnellere Synchronisation direkt über shared memory statt über system calls

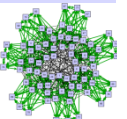
➤ Leichter Prozess (**Lightweight Process, LWP**)

- allgemein: Thread of Control = LWP
- Solaris Terminologie: LWP = Kernel Thread + appl. context
- Kernel Thread: scheduling Einheit des Kerns
- User level thread: auf Benutzerebene realisiert (Library oder JVM), unsichtbar für den Kern



C-Thread API (Vorläufer von POSIX)

1. `pthread_t pthread_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. `pthread_exit ()`
Terminiert den laufenden thread. Ein `pthread_exit` wird implizit am Ende der Hauptfunktion des threads ausgeführt.
3. `any_t pthread_join (id)`
`pthread_t id;`
Wartet auf das Ende des threads mit ID `id`. Das Resultat ist das Resultat der Hauptfunktion von thread `id`.
4. `pthread_detach ()`
Deklariert, daß niemand auf das Ergebnis des laufenden thread warten wird.
5. `pthread_t pthread_self ()`
Liefert die ID des laufenden thread.
6. `pthread_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  $A[i] \leq A[j] \quad \forall i,j, i \leq k < j$ 
  //(3) [Herrsche.]
      Sortiere (A[1...k],k),
      Sortiere (A[k+1,...n],n-k);

  //(4) [Ende.]
  return
}
```



Beispiel mit Threads

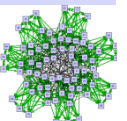
```

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

/*Sortiere Array A[1..n] im Intervall [l,r]
*/
{ //(1) [Trivialfall.] Sortiere A[1..n] im Intervall [l,r]
  if (r<=l) return;
  //(2) [Teile.] Errechne k,  $l \leq k < r$  und verteile A intern um,
  //    so daß  $A[i] \leq A[j] \quad \forall i,j, i \leq k < j$  und  $l \leq k < r$ 
  //(3) [Herrsche parallel.]
    { any_t a[3]; pthread_t id;
      a[0] = A[1..n]; a[1] = l; a[2]=k
      id = pthread_fork (Sortiere_jacket, a);
      Sortiere (A[1..n],k+1,n);
    }
  //(4) [Ende.]
    pthread_join(id);
  }
}

Sortiere_jacket(arg)
any_t arg[3];
{
  Sortiere (arg[0], arg[1], arg[2]); }

```

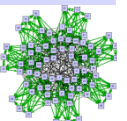


POSIX Threads (Pthreads)

- 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);**



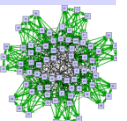
User-Level Threads

- BS sieht nur den Prozess
 - Nicht auf spezielle BS-Funktionalität angewiesen,
 - Thread Library „einfach“ ein Programm
- Implementation
 - Thread Control Block als struct
 - Pro Thread ein Stack-Segment (erzeugt mit malloc())
 - Gesamte Thread-Verwaltung in User-Library
 - thread_create, thread_join, yield, ...
 - Kontextwechsel über C Funktionen setjmp() / longjmp()
- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads



User-Level Threads

- Betriebssystem sieht die User-Level Threads nicht
 - Kein Thread-Scheduling durch BS
 - Scheduling durch User-Level Paket, evtl. applikationsspezifisch
 - Thread-Wechsel (**yield**) nur an vorgesehenen Stellen, ohne System Call (--> schnell)
 - Kein per-thread kernel stack
 - Blockierender System Call eines Threads blockiert den Prozess und daher alle anderen Threads ebenfalls
 - Test nötig, ob Ressource bereit, andernfalls yield
- Verwendung von mehreren Prozessoren nur bei Abbildung auf mehrere Prozesse (mit shared memory)



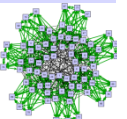
Kernel-Level Threads

➤ Kernel verwaltet Threads (inkl. Scheduling)

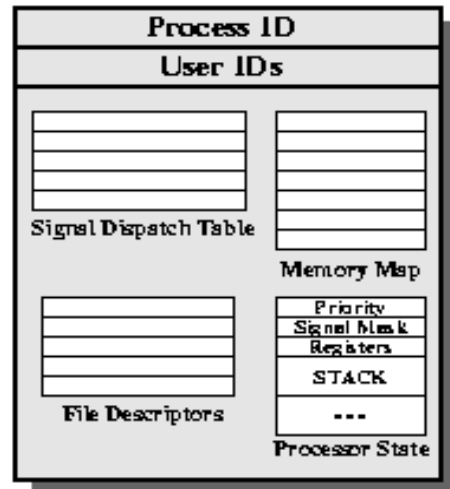
- Per-Thread Kernel Stack
- Applikation in 1 Prozess kann auf mehreren Prozessoren laufen
- Keine Prozess-Blockade durch blockierenden Thread
- simultane system calls möglich

➤ Beispiele

- Mach
- Modernes UNIX
 - SUN Solaris 2, HP UX 10, AIX, Tru64, FreeBSD 5.2
 - Linux, MacOS X
- OS/2
- Windows (seit NT)



UNIX Process Structure



Solaris 2.x Process Structure

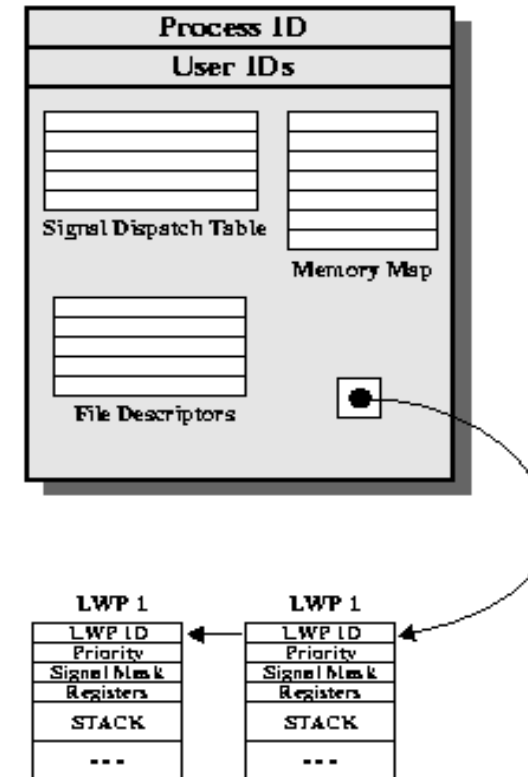
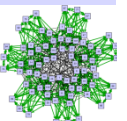


Figure 4.15 Process Structure in Traditional UNIX and Solaris 2.x [LEW196]



Vergleich User-Level mit Kernel-Level Threads

➤ Zeiten für Verwaltung (μ s, 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

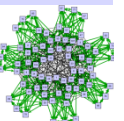


Vergleich User-Level mit Kernel-Level Threads

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

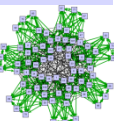
➤ Pro Kernel-Level

- 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: Kombiniertes 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



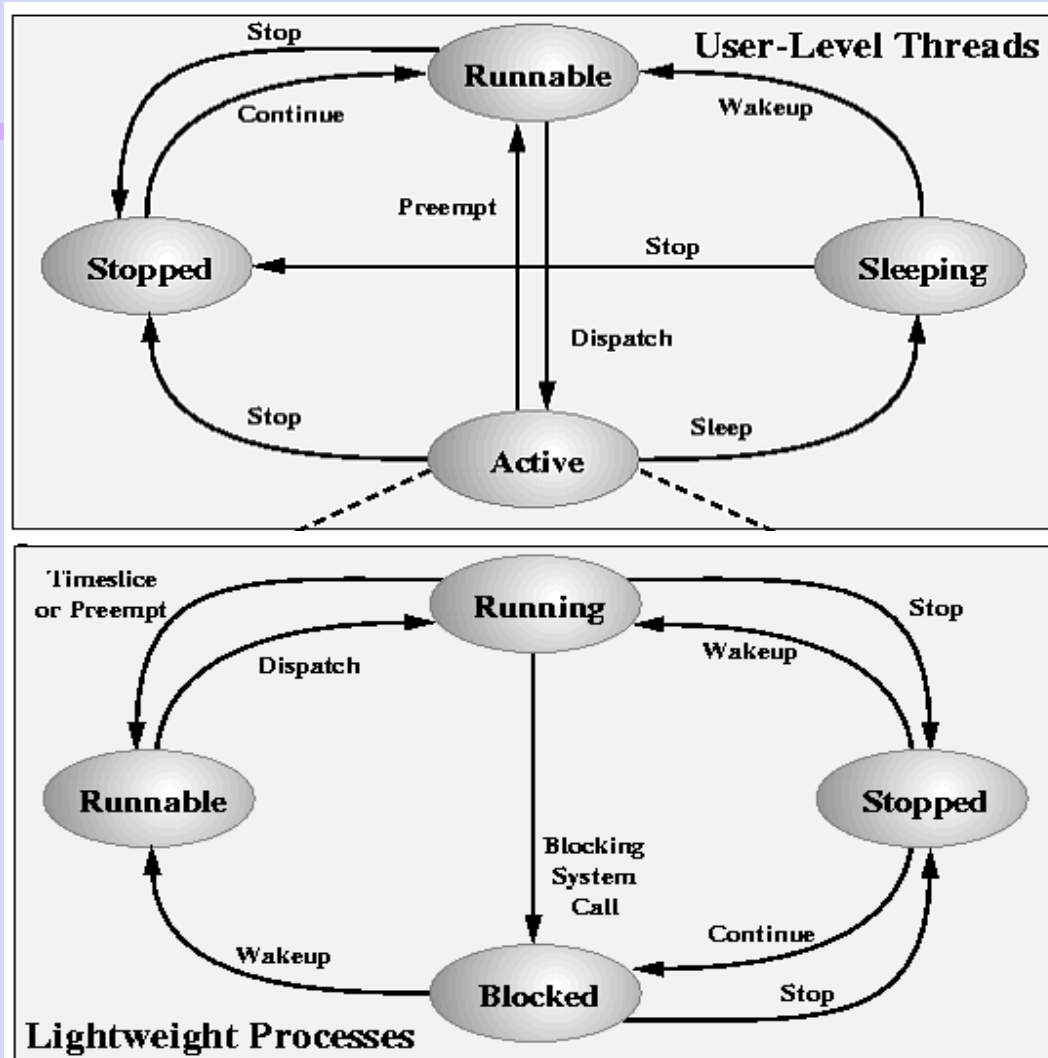


Figure 4.16 Solaris User-Level Thread and LWP States

- user-level control block + stack
 - moved between user-queues
 - cannot run or block by itself
 - active: attached to light-weight process (LWP)
- LWP=kernel thread+user data (syscall args, signal mask, tcb)
- assumes execution states while executing user-level thread
- kernel-level control block+stack
 - moved between kernel-queues
 - can run a user thread
 - can block in kernel doing sys-call for user
- kernel thread can run kernel services (interrupt handlers etc.)



Abbildung User Level → Kernel Level Threads

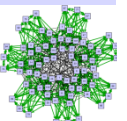
➤ Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads
 - Single processor JVM

➤ One-to-One

- Each user-level thread maps to kernel thread
- Examples
 - Windows NT/XP/2000
 - (Early) Linux
 - Solaris 9 and later

➤ Many-to-Many



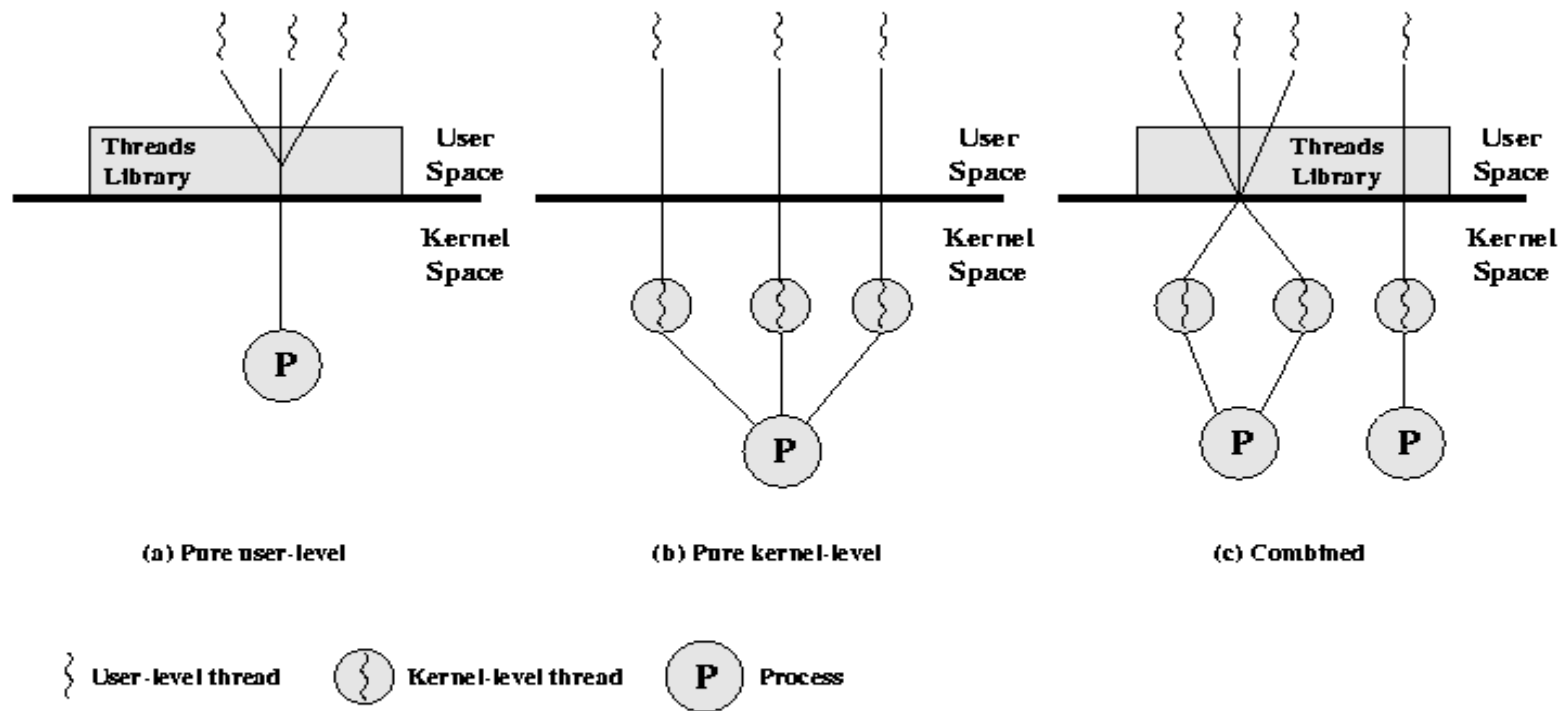
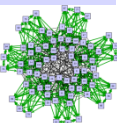
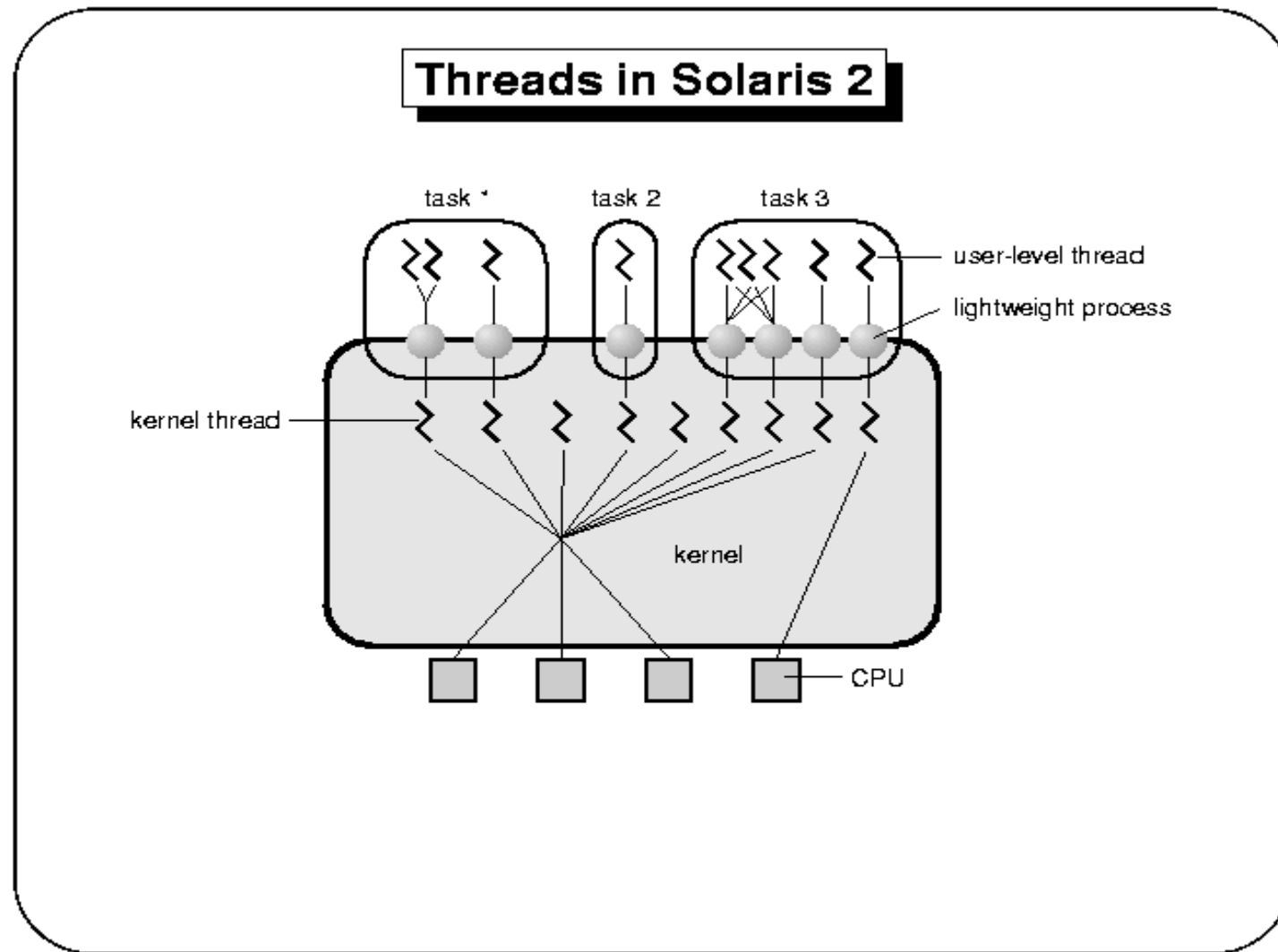


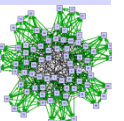
Figure 4.6 User-Level and Kernel-Level Threads





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 (system calls, malloc())
- Worker Pool concept works on many levels
 - Threads and work (micro-tasks)
 - Kernel threads and user-level threads
 - processes and user level threads



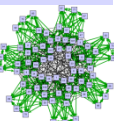
Signals and Threads

- 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
- Solaris: per LWP signal masks

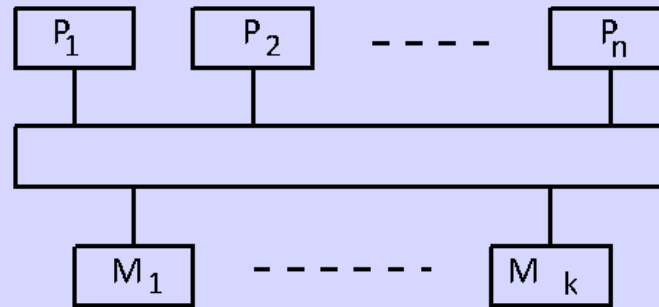


Symmetric Multiprocessing

- Traditionelle Sicht: 1 Prozessor
 - Computer = sequentielle Maschine
- Architekturen mit mehreren Prozessoren
 - **SIMD** = **S**ingle **I**nstruction, **M**ultiple **D**ata (Vektorrechner)
 - **MIMD** = **M**ultiple **I**nstruction, **M**ultiple **D**ata
 - Verteilter Speicher (Distributed Memory)
 - Gemeinsamer Speicher (Shared Memory)
 - SMP
 - Neu: mehrkernige Prozessoren: 2+ Prozessoren auf einem Chip
 - SPARC T1 und T2
 - Intel SCC (Single-Cloud-Computer) mit 48 Kernen, soll bis 1.000 Kerne skalieren.



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



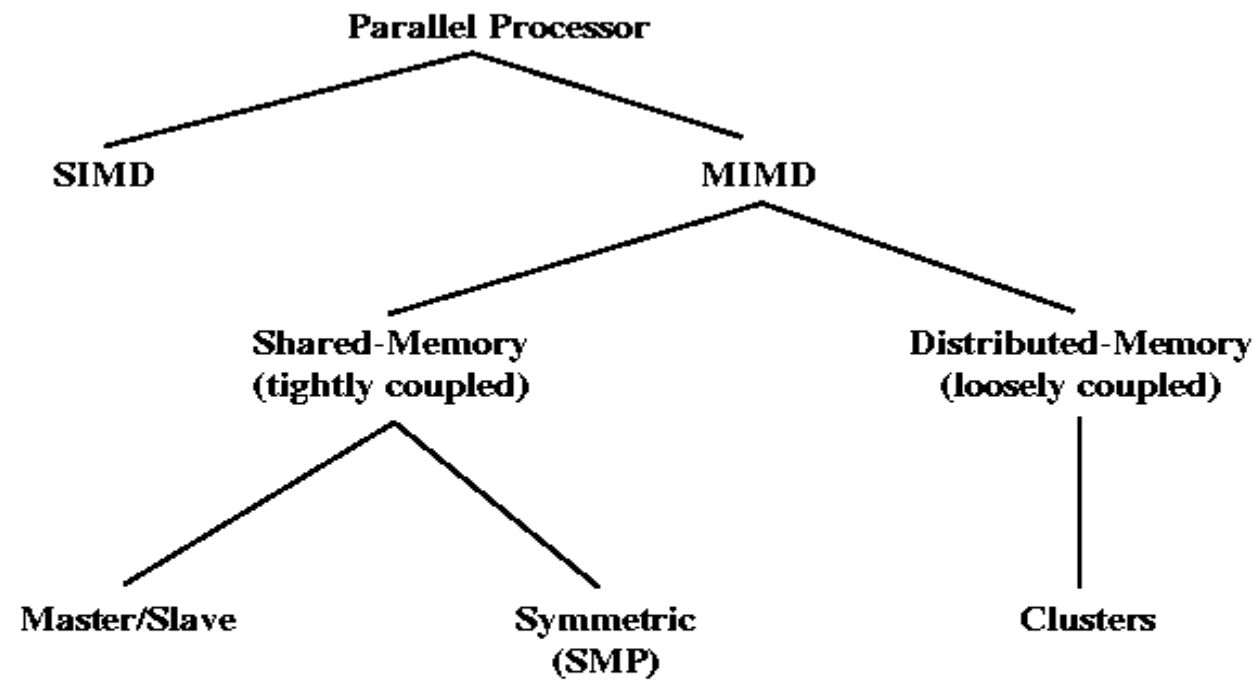
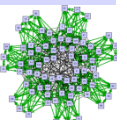


Figure 4.7 Parallel Processor Architectures



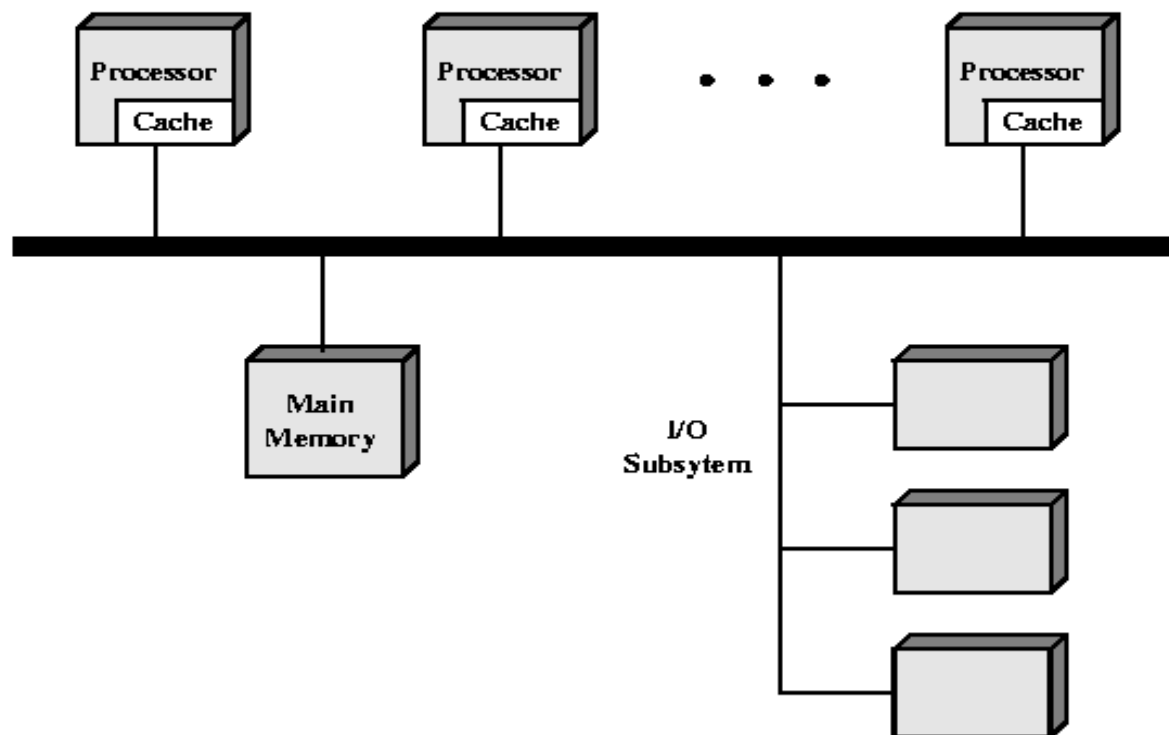
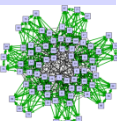
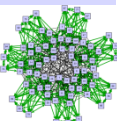


Figure 4.8 Symmetric Multiprocessor Organization



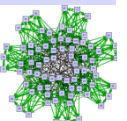
Betriebssystem für SMP

- Gleichzeitiges Bearbeiten von Systemaufrufen
 - Gleichzeitiges Blockieren der interrupts bzw. Systemaufrufe auf n Prozessoren nicht effizient oder nicht möglich
 - Kernel-Code muß **reentrant** sein: mehrere Syscalls gleichzeitig
- MP Scheduling
 - processor affinity: soft oder hard
 - **Soft**: Dispatcher *versucht* Thread auf gleichen Prozessor zu setzen
 - **Hard**: Feste Bindung Thread zu Prozessor
- Effiziente Synchronisation
- Memory Management
 - **Cache Coherence**
- Vorteile:
 - Zuverlässigkeit / Fehlertoleranz / Beschleunigung



SMP: Solaris

- Interrupts durch Kernel (System) Threads behandelt
- Prozessor bekommt Interrupt
 - Laufender Thread wird unterbrochen
 - Interrupt handler läuft auf Kernel Thread (aus Free-List)
 - Interrupt Thread wie gewöhnlicher Kernel Thread (ID, Kontext, Kernel-Stack, ...)
 - Kernel kontrolliert Zugriff zu Datenstrukturen (queues,...) 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



Beispiel: Threads und Realzeit

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

A;
B;
C;

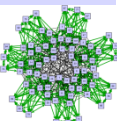
 }

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



Beispiel: 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



Parallele Behandlung aperiodischer Aufgaben

- Jede der Aufgaben A,B,C,D als separater (zyklischer) Prozess / Thread
- Prozesse laufen parallel / nebenläufig
 - A,B,C: while (TRUE) {

oder C

- start;

used);

start = gettime();

A; //resp. B

used = gettime();

sleep (periode -

}
 - D: schlafender / blockierender Prozess, der durch externes Signal geweckt wird
- Preemptive Scheduling mit genügend kleiner Zeitscheibe garantiert Prozess-Prioritäten und Einhaltung der Zeitanforderungen.

