

J.8 Kontrollfäden / Aktivitätsträger (*Threads*)

- Mehrere Prozesse zur Strukturierung von Problemlösungen
 - ◆ Aufgaben einer Anwendung leichter modellierbar, wenn sie in mehrere kooperierende Prozesse unterteilt wird
 - z. B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
 - z. B. Anwendungen mit vielen gleichzeitigen Aufgaben (Webbrowser)
 - ◆ Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
 - früher nur bei Hochleistungsrechnern (Aerodynamik, Wettervorhers.)
 - durch Multicoresysteme jetzt massive Verbreitung
 - ◆ Client-Server-Anwendungen unter UNIX:
 - pro Anfrage wird ein neuer Prozess gestartet
 - z. B. Webserver

1 Prozesse mit gemeinsamem Speicher

■ Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse

▲ Nachteile

◆ viele Betriebsmittel zur Verwaltung eines Prozesses notwendig

- Dateideskriptoren
- Speicherabbildung
- Prozesskontrollblock

◆ Prozessumschaltungen sind aufwändig

★ Vorteil

◆ in Multiprozessorsystemen sind echt parallele Abläufe möglich

2 Threads in einem Prozess

★ Lösungsansatz:

Kontrollfäden / Aktivitätsträger (*Threads*) oder **leichtgewichtige Prozesse** (*Lightweighted Processes, LWPs*)

- ◆ Jeder Thread repräsentiert einen eigenen aktiven Ablauf:
 - eigener Programmzähler
 - eigener Registersatz
 - eigener Stack
- ◆ eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln (gemeinsame Ausführungsumgebung)
 - Instruktionen
 - Datenbereiche (Speicher)
 - Dateien, etc.
- ➔ letztlich wird das Konzept des Prozesses aufgespalten:
eine Ausführungsumgebung für mehrere (parallele oder nebenläufige) Abläufe

2 Threads (2)

- ◆ Umschalten zwischen zwei Threads einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung.
 - es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf).
 - Speicherabbildung muss nicht gewechselt werden.
 - alle Systemressourcen bleiben verfügbar.
- ein klassischer UNIX-Prozess ist ein Adressraum mit einem Thread
- Implementierungen von Threads
 - ◆ User-level Threads
 - ◆ Kernel-level Threads

3 User-Level-Threads

■ Implementierung

- ◆ Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
- ◆ Realisierung durch Bibliotheksfunktionen
- ◆ Betriebssystem sieht nur einen Kontrollfaden

★ Vorteile

- ◆ keine Systemaufrufe zum Umschalten erforderlich
- ◆ effiziente Umschaltung (einige wenige Maschinenbefehle)
- ◆ Schedulingstrategie in der Hand des Anwendungsprogrammierers

▲ Nachteile

- ◆ bei blockierenden Systemaufrufen bleibt die ganze Anwendung (und damit alle User-Level-Threads) stehen
- ◆ kein Ausnutzen eines Multiprozessors möglich

4 Kernel-Level-Threads

■ Implementierung

- ◆ Betriebssystem kennt Kernel-Level-Threads
- ◆ Betriebssystem schaltet zwischen Threads um

★ Vorteile

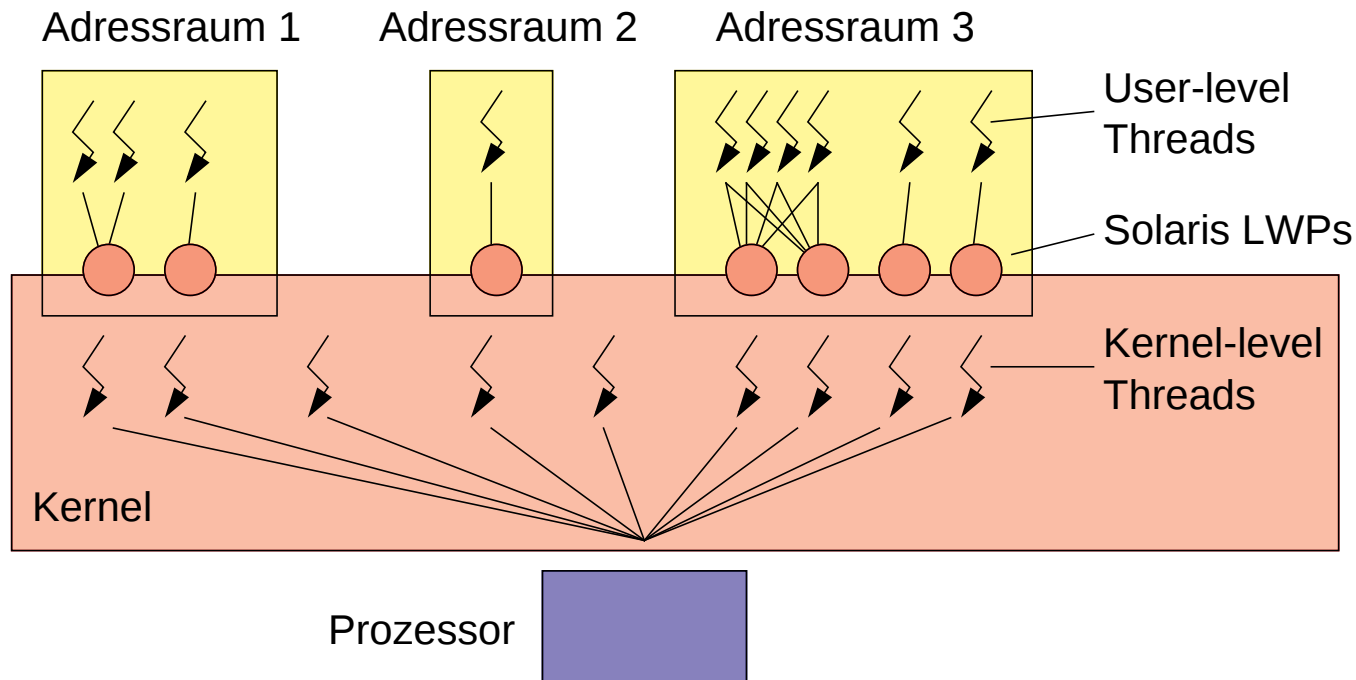
- ◆ kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
- ◆ Betriebssystem kann mehrere Threads einer Anwendung gleichzeitig auf verschiedenen Prozessoren laufen lassen

▲ Nachteile

- ◆ weniger effizientes Umschalten zwischen Threads (Umschaltung in den Systemkern notwendig)
- ◆ Schedulingstrategie meist durch Betriebssystem vorgegeben

5 Beispiel: LWPs und Threads (Solaris)

- Solaris kennt Kernel-, User-Level-Threads und LWPs



Nach Silberschatz, 1994

- Mischform: wenige Kernel-level-Threads um Parallelität zu erreichen, viele User-level-Threads, um die unabhängigen Abläufe in der Anwendung zu strukturieren

J.9 Koordinierung / Synchronisation

- Threads arbeiten nebenläufig oder parallel auf Multiprozessor
- Threads haben gemeinsamen Speicher
- ➡ alle von Interrupts und Signalen bekannten Probleme beim Zugriff auf gemeinsame Daten treten auch bei Threads auf
- ★ Unterschied zwischen Threads und Interrupt-Service-Routinen bzw. Signal-Handler-Funktionen:
 - ◆ "Haupt-Kontrollfaden" der Anwendung und eine ISR bzw. ein Signal-Handler sind nicht gleichberechtigt
 - ISR bzw. Signal-Handler unterbricht den Haupt-Kontrollfaden aber ISR bzw. Signal-Handler werden nicht unterbrochen
 - ◆ zwei Threads sind gleichberechtigt
 - ein Thread kann jederzeit zugunsten eines anderen unterbrochen werden (Scheduler) oder parallel zu einem anderen arbeiten (MPS)
- ➡ Interrupts sperren oder Signale blockieren hilft nicht!

1 Koordinierungsprobleme

■ Grundlegende Probleme

- ◆ gegenseitiger Ausschluss (Koordination)
 - ein Thread möchte auf einen kritischen Datenbereich zugreifen und verhindern, dass andere Threads gleichzeitig zugreifen
- ◆ gegenseitiges Warten (Synchronisation)
 - ein Thread will darauf warten, dass ein anderer einen bestimmten Bearbeitungsstand erreicht hat

■ Komplexere Koordinierungsprobleme (Beispiele)

- ◆ Bounded Buffer
 - Threads schreiben Daten in einen Pufferspeicher (meist als Feld implementiert), andere entnehmen Daten (Zugriff auf den Puffer, Puffer leer, Puffer voll)
- ◆ Philosophenproblem
 - ein Thread reserviert sich zuerst Zugriff auf Datenbereich 1, dann auf Datenbereich 2, der andere Thread umgekehrt
 - ➡ kann zu Verklemmung führen

2 Gegenseitiger Ausschluss (*mutual exclusion*)

■ Einfache Implementierung durch *mutex*-Variablen

```
mutex m = 1;
int counter = 0;
```

Thread 1

```
...
lock(&m);
counter++;
unlock(&m);
...
```

Thread 2

```
...
lock(&m);
printf("%d\n", counter);
counter = 0;
unlock(&m);
...
```

■ Realisierung

```
void lock (mutex *m) {
    while (*m == 0) {
        /* warten */
    }
    m = 0;
}
```

```
void unlock (mutex *m) {
    *m = 1;
    /* ggf. wartende
    Threads wecken */
}
```

3 Semaphore

- Ein Semaphor (griech. Zeichenträger) ist eine Datenstruktur des Systems mit zwei Operationen (nach *Dijkstra*)

- ◆ P-Operation (*proberen; passieren; wait; down*)

- wartet bis Zugang frei

```
void P( int *s )
{
    while( *s <= 0 ) /* warten */;
    *s= *s-1;
}
```

atomare Funktion

- ◆ V-Operation (*verhogen; vrijgeven; signal; up*)

- macht Zugang für anderen Prozess frei

```
void V( int *s )
{
    *s= *s+1;
}
```

atomare Funktion

- ähnlich zu lock/unlock, aber mächtiger

4 Gegenseitiges Warten

■ Implementierung mit Hilfe eines Semaphors

```
int barrier = 0;  
int result;
```

```
... Thread 1  
P(&barrier);  
f1(result);  
...
```

```
... Thread 2  
result = f2(...);  
V(&barrier);  
...
```

5 Unterstützung durch spezielle Maschinenbefehle

- Problem: Implementierung der lock- oder der P-Operation

```
void P ( int *s )
{
    if ( *s <= 0 ) {
        sleep();
    }
    *s= *s-1;
}
```

hier darf keine V-Operation
dazwischen kommen!
(lost-wakeup-Problem)

- Lösungsmöglichkeiten

- ◆ Implementierung durch das Betriebssystem und keine Threadumschaltung während der Operation
 - hilft nicht bei parallelem Ablauf auf Multiprozessor
- ◆ Atomarität kurzer kritische Abschnitte durch spezielle Maschinenbefehle, die den Zugriff anderer Prozessoren auf den Speicher kurzzeitig sperren
 - *Test-and-Set* Instruktion
 - *Compare-and-Swap* Instruktion

5 Unterstützung durch spezielle Maschinenbefehle (2)

■ Test-and-set

◆ Maschinenbefehl mit folgender Wirkung

```
bool test_and_set( bool *plock )
{
    bool initial= *plock;
    *plock= TRUE;
    return initial;
}
```

◆ Ausführung ist atomar

➡ wenn zwei Threads den Befehl gleichzeitig ausführen wollen sorgt die Hardware dafür, dass ein Thread den Befehl vollständig zuerst ausführt

◆ Ausgangssituation: ***plock == FALSE**

◆ Ergebnis von **test_and_set**:

- der Thread, der den Befehl zuerst ausführt, erhält **FALSE**,
der andere **TRUE**

5 Unterstützung durch spezielle Maschinenbefehle (3)

■ Kritische Abschnitte mit Test-and-Set Befehlen

```
bool lock= FALSE;
int counter = 0;
```

Prozess 0

```
while( 1 ) {
    while(
        test_and_set(&lock) ){};
    /* critical */
    counter ++;

    lock= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    while(
        test_and_set(&lock) ){};
    /* critical */
    printf("%d",counter);
    counter --;

    lock= FALSE;

    ... /* uncritical */
}
```

★ Code ist identisch und für mehr als zwei Prozesse geeignet

5 Unterstützung durch spezielle Maschinenbefehle (4)

- Koordination durch atomare Maschinenbefehle ist **nicht-blockierend!**
 - andere Prozessoren werden nur durch die kurzzeitige Bus-Sperre während des Test-and-Set Befehls aufgehalten
 - durch geschickte Wahl der Datenstrukturen und Zugriffsalgorithmen in der Anwendung kann man kritische Abschnitten, die durch locks gesichert werden müssen, minimieren oder ganz vermeiden
- ↳ extrem wichtig bei hochgradiger Parallelität
 - ◆ Sperre kritischer Abschnitte durch Locks verhindert parallele Abläufe (noch harmlos bei zwei Threads, katastrophal bei 10 oder 100 Threads)
 - Problem: Nichtblockierende Zugriffsalgorithmen für komplexere Datenstrukturen(z. B. verkettete Listen) sind sehr kompliziert