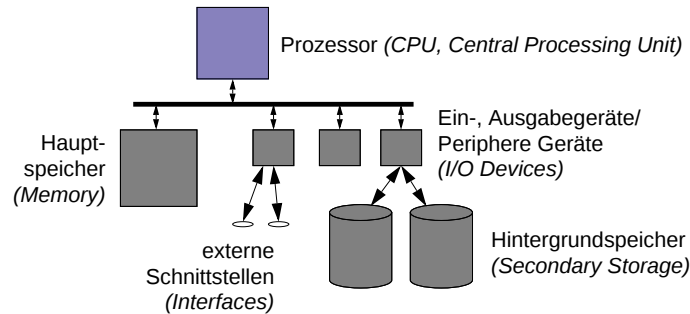


■ Einordnung



J.1 Prozessor (2)

■ Beispiel für Instruktionen

```
...
0010 5510000000 movl DS:$10, %ebx
0015 5614000000 movl DS:$14, %eax
001a 8a          addl %eax, %ebx
001b 5a18000000 movl %ebx, DS:$18
...
```

■ Prozessor arbeitet in einem bestimmten Modus

- ◆ Benutzermodus: eingeschränkter Befehlssatz
- ◆ privilegierter Modus: erlaubt Ausführung privilegierter Befehle
 - Konfigurationsänderungen des Prozessors
 - Moduswechsel
 - spezielle Ein-, Ausgabebefehle

J.1 Prozessor

■ Register

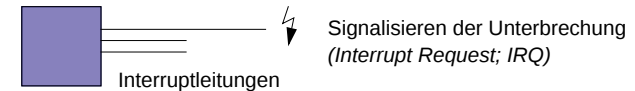
- ◆ Prozessor besitzt Steuer- und Vielzweckregister
- ◆ Steuerregister:
 - Programmzähler (*Instruction Pointer*)
 - Stapelregister (*Stack Pointer*)
 - Statusregister
 - etc.

■ Programmzähler enthält Speicherstelle der nächsten Instruktion

- ◆ Instruktion wird geladen und
- ◆ ausgeführt
- ◆ Programmzähler wird inkrementiert
- ◆ dieser Vorgang wird ständig wiederholt

J.1 Prozessor (3)

■ Unterbrechungen (*Interrupts*)



◆ ausgelöst durch ein Signal eines externen Geräts

- asynchron zur Programmausführung
 - Prozessor unterbricht laufende Bearbeitung und führt eine definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
 - vorher werden alle Register einschließlich Programmzähler gesichert (z.B. auf dem Stack)
 - nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden
 - Unterbrechungen werden im privilegierten Modus bearbeitet

J.1 Prozessor (4)

- **Ausnahmesituationen, Systemaufrufe (*Traps*)**
 - ◆ ausgelöst durch eine Aktivität des gerade ausgeführten Programms
 - fehlerhaftes Verhalten
(Zugriff auf ungültige Speicheradresse, ungültiger Maschinenbefehl, Division durch Null)
 - kontrollierter Eintritt in den privilegierten Modus
(spezieller Maschinenbefehl - *Trap* oder *Supervisor Call*)
– Implementierung der Betriebssystemschnittstelle
 - ↳ synchron zur Programmausführung
- ◆ Prozessor schaltet in privilegierten Modus und führt definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
 - Ausnahmesituation wird geeignet bearbeitet (z. B. durch Abbruch der Programmausführung)
 - Systemaufruf wird durch Funktionen des Betriebssystems im privilegierten Modus ausgeführt (partielle Interpretation)
 - Parameter werden nach einer Konvention übergeben (z.B. auf dem Stack)

J.2 Prozesse (2)

- **Mehrprogrammbetrieb**
 - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
 - steht nur ein echter Prozessor zur Verfügung, werden Zeitanteile der Rechenzeit an die Prozesse vergeben (***Time Sharing System***)
 - die Entscheidung, welcher Prozess zu welchem Zeitpunkt wieviel Rechenzeit zugeteilt bekommt, trifft das Betriebssystem (***Scheduler***)
 - die Umschaltung zwischen Prozessen erfolgt durch das Betriebssystem (***Dispatcher***)
 - Prozesse laufen nebenläufig
(das ausgeführte Programm weiß nicht, an welchen Stellen auf einen anderen Prozess umgeschaltet wird)

J.2 Prozesse

- ▲ **Bisherige Definition:**
 - ◆ Programm, das sich in Ausführung befindet, und seine Daten
(Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
 - eine konkrete Ausführungsumgebung für ein Programm mit den dazu erforderlichen Betriebsmitteln:
Speicher, Rechte, Verwaltungsinformation (verbrauchte Rechenzeit,...),...
- **eine etwas andere Sicht:**
 - ◆ ein virtueller Prozessor, der ein Programm ausführt
 - Speicher → virtueller Adressraum
 - Prozessor → Zeitanteile am echten Prozessor
 - Interrupts → Signale
 - I/O-Schnittstellen → Dateisystem, Kommunikationsmechanismen
 - Maschinenbefehle → direkte Ausführung durch echten Prozessor
oder
partielle Interpretation von Trap-Befehlen
durch Betriebssystemcode

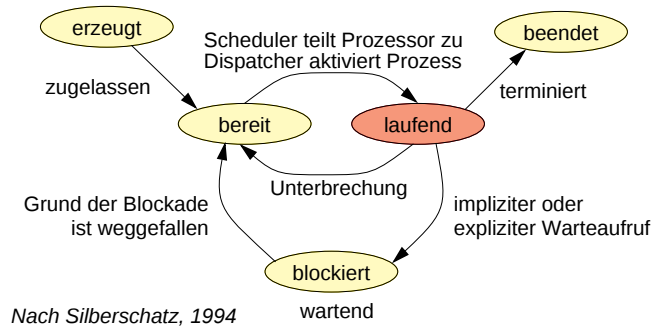
J.3 Prozesszustände

- **Ein Prozess befindet sich in einem der folgenden Zustände:**
 - ◆ **Erzeugt (*New*)**
Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
 - ◆ **Bereit (*Ready*)**
Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
 - ◆ **Laufend (*Running*)**
Prozess wird vom realen Prozessor ausgeführt
 - ◆ **Blockiert (*Blocked/Waiting*)**
Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
 - ◆ **Beendet (*Terminated*)**
Prozess ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

J.3 Prozesszustände (2)

J.3 Prozesszustände

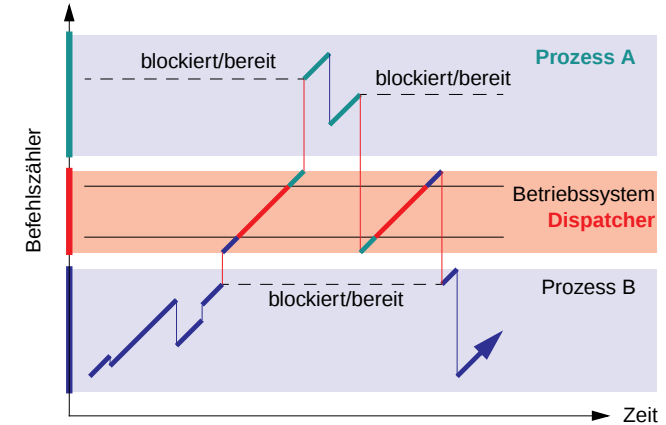
■ Zustandsdiagramm



J.4 Prozesswechsel (2)

J.4 Prozesswechsel

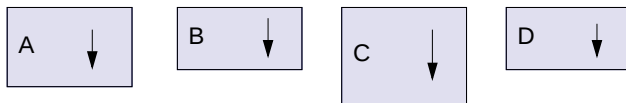
■ Umschaltung



J.4 Prozesswechsel

J.4 Prozesswechsel

■ Konzeptionelles Modell



vier Prozesse mit eigenständigen Befehlszählern

■ Umschaltung (Context Switch)

- ◆ Sichern der Register des laufenden Prozesses inkl. Programmzähler (Kontext),
- ◆ Auswahl des neuen Prozesses,
- ◆ Ablaufumgebung des neuen Prozesses herstellen (z.B. Speicherabbildung, etc.),
- ◆ gesicherte Register des neuen Prozesses laden und
- ◆ Prozessor aufsetzen.

J.4 Prozesswechsel (3)

J.4 Prozesswechsel

■ Prozesskontrollblock (Process Control Block; PCB)

- ◆ Datenstruktur des Betriebssystems, die alle nötigen Daten für einen Prozess hält. Beispielsweise in UNIX:

- Prozessnummer (PID)
- verbrauchte Rechenzeit
- Erzeugungszeitpunkt
- Kontext (Register etc.)
- Speicherabbildung
- Eigentümer (UID, GID)
- Wurzelkatalog, aktueller Katalog
- offene Dateien
- ...

J.5 Prozesserzeugung (UNIX)

J.5 Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
- ◆ Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

```
pid_t p;          Vater
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

J.5 Prozesserzeugung (2)

J.5 Prozesserzeugung (UNIX)

- ◆ Der Kind-Prozess ist eine perfekte **Kopie** des Vaters
 - gleiches Programm
 - gleiche Daten (gleiche Werte in Variablen)
 - gleicher Programmzähler (nach der Kopie)
 - gleicher Eigentümer
 - gleiches aktuelles Verzeichnis
 - gleiche Dateien geöffnet (selbst Schreib-, Lesezeiger ist gemeinsam)
 - ...
- ◆ Unterschiede:
 - Verschiedene PIDs
 - **fork()** liefert verschiedene Werte als Ergebnis für Vater und Kind

J.5 Prozesserzeugung (UNIX)

J.5 Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
- ◆ Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

```
pid_t p;          Vater
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

```
pid_t p;          Kind
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

J.6 Ausführen eines Programms (UNIX)

J.6 Ausführen eines Programms (UNIX)

- Das von einem Prozess gerade ausgeführte Programm kann durch ein neues Programm ersetzt werden

```
int execv( const char *path, char *const argv[] );
```

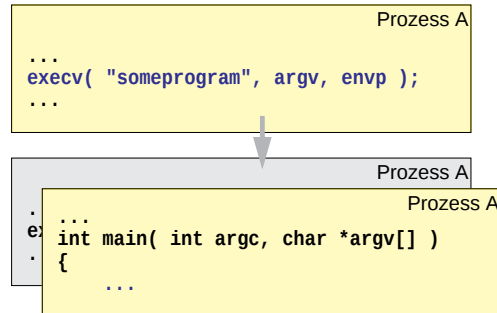
```
Prozess A
...
execv( "someprogram", argv, envp );
...
```

J.6 Ausführen eines Programms (UNIX)

J.6 Ausführen eines Programms (UNIX)

- Das von einem Prozess gerade ausgeführte Programm kann durch ein neues Programm ersetzt werden

```
int execv( const char *path, char *const argv[]);
```



das zuvor ausgeführte Programm wird dadurch beendet.

J.7 Signale

J.7 Signale

1 Signalisierung des Systemkerns an einen Prozess

- Software-Implementierung der Hardware-Konzepte

- ◆ **Interrupt:** asynchrones Signal aufgrund eines "externen" Ereignisses
 - CTRL-C auf der Tastatur gedrückt (Interrupt-Signal)
 - Timer abgelaufen
 - Kind-Prozess terminiert
 - ...
- ◆ **Trap:** synchrones Signal, ausgelöst durch die Aktivität des Prozesses
 - Zugriff auf ungültige Speicheradresse
 - Illegalen Maschinenbefehl
 - Division durch NULL
 - Schreiben auf eine geschlossene Kommunikationsverbindung
 - ...

J.6 Operationen auf Prozessen (UNIX)

J.6 Ausführen eines Programms (UNIX)

- ◆ Prozess beenden

```
void _exit( int status );  
[ void exit( int status ); ]
```

- ◆ Prozessidentifikator

```
pid_t getpid( void );           /* eigene PID */  
pid_t getppid( void );        /* PID des Vaterprozesses */
```

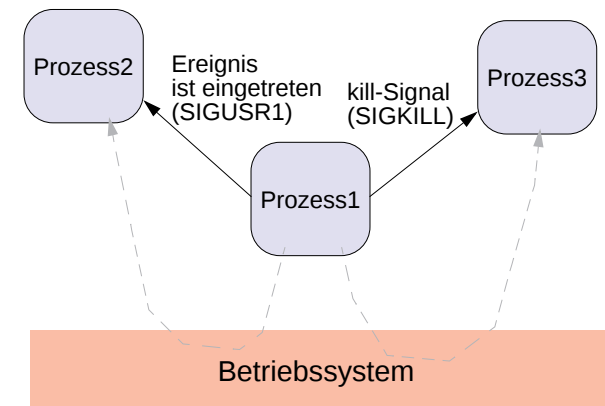
- ◆ Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```

2 Kommunikation zwischen Prozessen

J.7 Signale

- ein Prozess will einem anderen ein Ereignis signalisieren

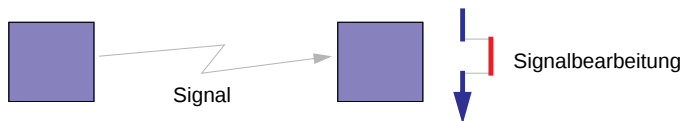


3 Reaktion auf Signale

- abort
 - ◆ erzeugt Core-Dump (Segmente + Registercontext) und beendet Prozess
- exit
 - ◆ beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
 - ◆ ignoriert Signal
- stop
 - ◆ stoppt Prozess
- continue
 - ◆ setzt gestoppten Prozess fort
- signal handler
 - ◆ Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

4 POSIX Signalbehandlung

- Betriebssystemschnittstelle zum Umgang mit Signalen
- Signal bewirkt Aufruf einer Funktion (analog ISR)



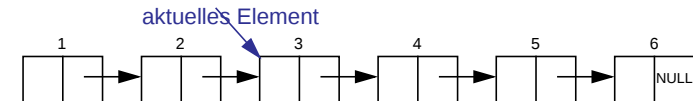
- ◆ nach der Behandlung läuft der Prozess an der unterbrochenen Stelle weiter

- Systemschnittstelle
 - ◆ sigaction – Anmelden einer Funktion = Einrichten der ISR-Tabelle
 - ◆ sigprocmask – Blockieren/Freigeben von Signalen $\approx$ cli() / sei()
 - ◆ sigsuspend – Freigeben + passives Warten auf Signal + wieder Blockieren $\approx$ sei() + sleep_cpu() + cli()
 - ◆ kill – Signal an anderen Prozess verschicken

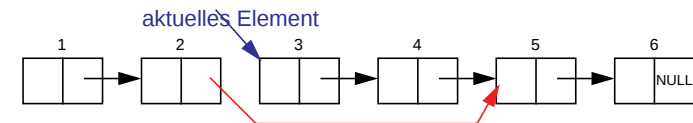
5 Signale und Nebenläufigkeit → Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller
- Beispiel:

- ◆ main-Funktion läuft durch eine verkettete Liste



- ◆ Prozess erhält Signal; Signalhandler entfernt Elemente 3 und 4 aus der Liste und gibt den Speicher dieser Elemente frei



5 Signale und Nebenläufigkeit → Race Conditions (2)

- zusätzliche Problem:
 - ◆ Signale können die Behandlung anderer Signale unterbrechen
 - ◆ Signale können Bibliotheksfunktionen unterbrechen, die nicht dafür eingerichtet sind
 - Funktionen printf() oder getchar()
 - siehe Funktion readdir im nächsten Kapitel

- Lösung:
 - Signal während Ausführung von kritischen Programmabschnitten blockieren!
 - kritische Bibliotheksfunktionen aus Signalbehandlungsfunktionen möglichst nicht aufrufen

- grundlegendes Problem
 - man muss wissen, welche Funktion(en) in Bezug auf Nebenläufigkeit problematisch sind (**nicht reentrant**)

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** (Lightweight 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

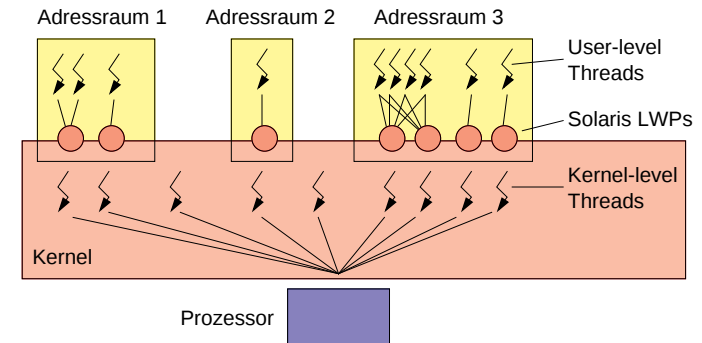
- Implementierung
 - ◆ Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
 - ◆ Realisierung durch Bibliotheksfunktionen
 - ◆ Betriebssystem sieht nur einen Kontrollfaden
- ★ Vorteile
 - ◆ keine Systemaufrufe zum Umschalten erforderlich
 - ◆ 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: LWP's 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

3 Semaphore

■ Ein Semaphore (griech. Zeichenträger) ist eine Datenstruktur des Systems mit zwei Operationen (nach Dijkstra)

◆ P-Operation (*proberen; passeren; 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

2 Gegenseitiger Ausschluss (*mutual exclusion*)

■ Einfache Implementierung durch *mutex*-Variablen

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

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

Thread 1

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

Thread 2

■ Realisierung

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

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

4 Gegenseitiges Warten

■ Implementierung mit Hilfe eines Semaphors

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

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

Thread 1

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

Thread 2

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 (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 (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 (4)

- Koordinierung 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

J.10 Thread-Konzepte in UNIX/Linux

J.10 Thread-Konzepte in UNIX/Linux

- verschiedene Implementierungen von Thread-Paketen verfügbar
 - reine User-Level-Threads
eine beliebige Zahl von User-Level-Threads wird auf einem Kernel Thread "gemultiplexed" (*many:1*)
 - reine Kernel-Level-Threads
jedem auf User-Level sichtbaren Thread ist 1:1 ein Kernel-Level-Thread zugeordnet (*1:1*)
 - Mischungen: eine große Zahl von User-Level Threads wird auf eine kleinere Zahl von Kernel Threads abgebildet (*many:many*)
 - + User-Level Threads sind billig
 - + die Kernel Threads ermöglichen echte Parallelität auf einem Multiprozessor
 - + wenn sich ein User-Level-Thread blockiert, dann ist mit ihm der Kernel-Level-Thread blockiert in dem er gerade abgewickelt wird — aber andere Kernel-Level-Threads können verwendet werden um andere, lauffähige User-Level-Threads weiter auszuführen

Systemnahe Programmierung in C

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2009

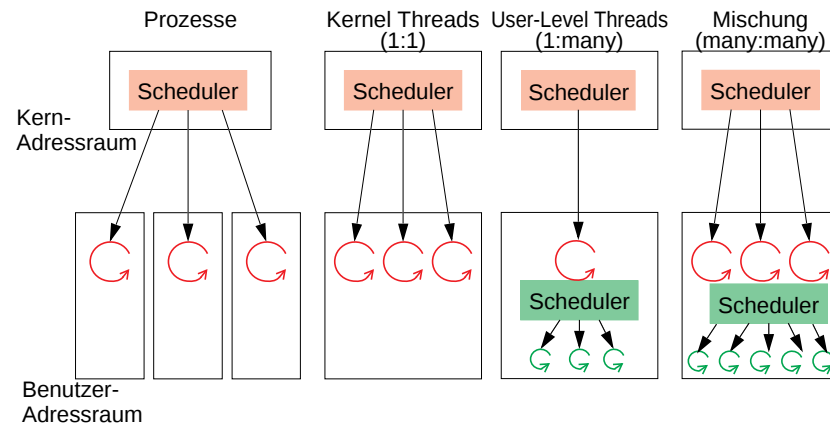
J-Prozesse.fm 2009-07-08 11.22

J.41

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

J.10 Thread-Konzepte in UNIX/Linux (2)

J.10 Thread-Konzepte in UNIX/Linux



- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
 - ➔ IEEE-POSIX-Standard P1003.4a

Systemnahe Programmierung in C

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2009

J-Prozesse.fm 2009-07-08 11.22

J.42

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 pthread-Benutzerschnittstelle

J.10 Thread-Konzepte in UNIX/Linux

■ Pthreads-Schnittstelle (Basisfunktionen):

pthread_create	Thread erzeugen & Startfunktion angeben
pthread_exit	Thread beendet sich selbst
pthread_join	Auf Ende eines anderen Threads warten
pthread_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

- Funktionen in Pthreads-Bibliothek zusammengefasst
gcc ... -pthread

Systemnahe Programmierung in C

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2009

J-Prozesse.fm 2009-07-08 11.22

J.43

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 pthread-Benutzerschnittstelle (2)

J.10 Thread-Konzepte in UNIX/Linux

■ Thread-Erzeugung

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine)(void *),
                  void *arg)
```

thread Thread-Id
attr Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.

Thread wird erzeugt und ruft Funktion **start_routine** mit Parameter **arg** auf.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

Systemnahe Programmierung in C

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2009

J-Prozesse.fm 2009-07-08 11.22

J.44

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 pthread-Benutzerschnittstelle (3)

- Thread beenden (bei return aus **start_routine** oder):

```
void pthread_exit(void *retval)
```

Der Thread wird beendet und **retval** wird als Rückgabewert zurück geliefert (siehe pthread_join)

- Auf Thread warten und exit-Status abfragen:

```
int pthread_join(pthread_t thread, void **retvalp)
```

Wartet auf den Thread mit der Thread-ID **thread** und liefert dessen Rückgabewert über **retvalp** zurück.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

3 Pthreads-Koordinierung

- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
 - ◆ Implementierung durch den Systemkern
 - ◆ komplexe Datenstrukturen, aufwändig zu programmieren
 - ◆ für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache **Mutex**-Variablen
 - ◆ gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

2 Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];
int main(int argc, char* argv[]) {
    pthread_t tids[100];
    ...
    for (i = 0; i < 100; i++)
        pthread_create(&tids[i], NULL, mult,
                      (void *)(&c[i]));
    for (i = 0; i < 100; i++)
        pthread_join(&tids[i], NULL);
    ...
}

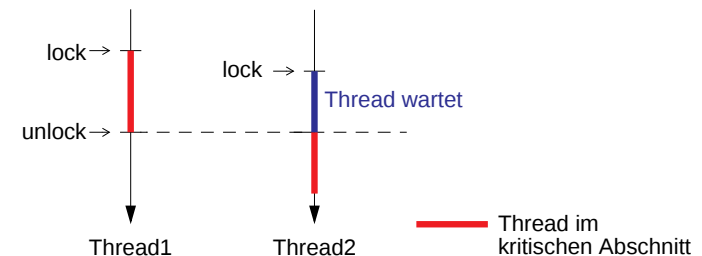
void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```

3 Pthreads-Koordinierung (2)

★ Mutexes

- Koordinierung von kritischen Abschnitten



3 Pthreads-Koordinierung (3)

... Mutexes (2)

■ Schnittstelle

◆ Mutex erzeugen

```
pthread_mutex_t m1;
s = pthread_mutex_init(&m1, NULL);
```

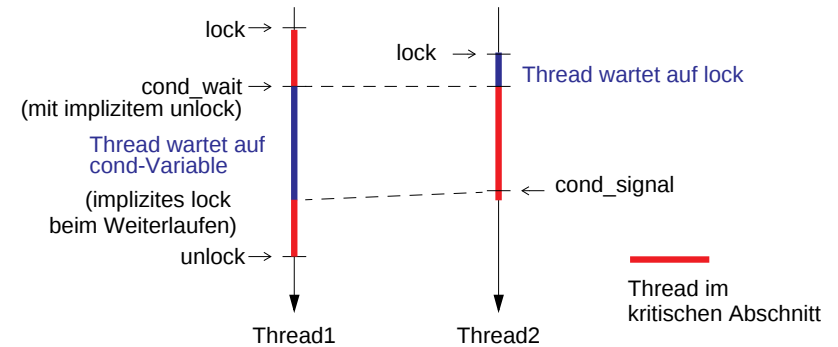
◆ Lock & unlock

```
s = pthread_mutex_lock(&m1);
... kritischer Abschnitt
s = pthread_mutex_unlock(&m1);
```

3 Pthreads-Koordinierung (5)

★ Condition Variables

- Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



3 Pthreads-Koordinierung (4)

... Mutexes (3)

- Komplexere Koordinierungsprobleme können alleine mit Mutexes nicht implementiert werden

- ➔ Problem:
 - Ein Mutex sperrt die eine komplexere Datenstruktur
 - Der Zustand der Datenstruktur erlaubt die Operation nicht
 - Thread muss warten, bis die Situation durch anderen Thread behoben wurde
 - Blockieren des Threads an einem weiteren Mutex kann zu Verklemmungen führen

- ➔ Lösung: Mutex in Verbindung mit sleep/wakeup-Mechanismus

- ➔ **Condition Variables**

3 Pthreads-Koordinierung (6)

... Condition Variables (2)

- Realisierung

- ◆ Thread reiht sich in Warteschlange der Condition Variablen ein
- ◆ Thread gibt Mutex frei
- ◆ Thread gibt Prozessor auf
- ◆ Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
- ◆ Deblocierter Thread muss als erstes den kritischen Abschnitt neu betreten (lock)
- ◆ Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden

3 Pthreads-Koordinierung (7)

... Condition Variables (3)

■ Schnittstelle

◆ Condition Variable erzeugen

```
pthread_cond_t c1;
s = pthread_cond_init(&c1, NULL);
```

◆ Beispiel: zählende Semaphore

P-Operation

```
void P(int *sem) {
    pthread_mutex_lock(&m1);
    while ( *sem == 0 )
        pthread_cond_wait
            (&c1, &m1);
    (*sem)--;
    pthread_mutex_unlock(&m1);
}
```

V-Operation

```
void V(int *sem) {
    pthread_mutex_lock(&m1);
    (*sem)++;
    pthread_cond_broadcast(&c1);
    pthread_mutex_unlock(&m1);
}
```

3 Pthreads-Koordinierung (8)

... Condition Variables (4)

■ Bei **pthread_cond_signal** wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher

- evtl. Prioritätsverletzung wenn nicht der höchstpriorisierte gewählt wird
- Verklemmungsgefahr wenn die Threads unterschiedliche Wartebedingungen haben

■ Mit **pthread_cond_broadcast** werden alle wartenden Threads aufgeweckt

■ Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert

J.11 Threads und Koordinierung in Java

■ Thread-Konzept und Koordinierungsmechanismen sind in Java integriert

■ Erzeugung von Threads über Thread-Klassen

➤ Beispiel

```
class MyClass implements Runnable {
    public void run() {
        System.out.println("Hello\n");
    }
}

....
MyClass o1 = new MyClass(); // create object
Thread t1 = new Thread(o1); // create thread to run in o1

t1.start(); // start thread

Thread t2 = new Thread(o1); // create second thread to run in o1
t2.start(); // start second thread
```

J.11 Threads und Koordinierung in Java (2)

★ Koordinierungsmechanismen

■ Monitore: exklusive Ausführung von Methoden eines Objekts

- es darf nur jeweils ein Thread die **synchronized**-Methoden betreten

➤ Beispiel:

```
class Bankkonto {
    int value;
    public synchronized void AddAmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmount(int v) {
        value=value-v;
    }
}

...
Bankkonto b=...
b.AddAmount(100);
```

◆ Conditions: gezieltes Freigeben des Monitors und Warten auf ein Ereignis

K letzte Infos / Organisatorisches

- bis 24.07. Übungsbetrieb mit Klausurvorbereitung
- Rechnerübungstermine finden weiter statt -
wenn nach 15 min. niemand kommt, geht der Übungsleiter
- Boards können gegen Ausweis jederzeit bei Michael Stilkerich zur Arbeit
im CIP-Pool ausgeliehen werden (max. jeweils ein Tag)
- alte Klausuren sind über die Web-Seite verfügbar
- Klausurtermin:
 - Freitag 31.07.2009, Uhrzeit steht noch nicht fest (-> Webseite)
- Fragestunde am Mi. 22.07.09 12:15
 - außerdem Besprechung der Evaluationsergebnisse
 - bei Bedarf zusätzliche Fragestunde in der Klausurwoche
- weitere Details und Termine auf der Webseite zur Vorlesung