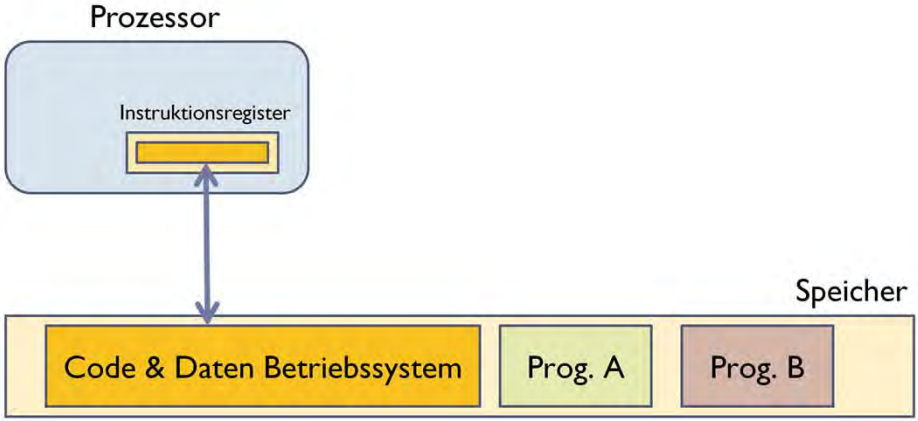
Betriebssystemaspekte

1. Einprozessor-Einkern-Systeme (†)
2. Mehrprozessor/Mehrkern-Systeme
3. SMP-Hardware
4. SMP-Betriebssystem
5. Synchronisationsmechanismen
6. Erweiterte Betriebssystemfunktionalität

1. Einprozessor-Einkern-Systeme

Gibt es seit ca. 2005 nicht mehr in Rechnern



Erinnerung zu Betriebssystemen

- (Ausführbares) Programm
  - Binärcode, der auf Platte steht und geladen wird
- Prozess
  - Objekt des Betriebssystems zur Verwaltung eines Programms in der Ausführung
  - Prozesse haben geschützte Adressräume – kein anderer Prozess hat Zugriff
    - Kommunikation zwischen Prozessen über Nachrichtenaustausch
  - Ressourcen wie z.B. Adressräume, Dateien, Unterbrechungen usw. werden prozessbezogen verwaltet
  - Verwaltet vom BS mittels Prozesskontrollblock
- Thread (in einem Prozess)
  - So eine Art Unterprozess im Prozess
  - Threads eines Prozesses teilen sich den Adressraum dieses Prozesses und können darin allen möglichen Unsinn anstellen
    - Threads von verschiedenen Prozessen sind natürlich gegeneinander geschützt – haben aber natürlich auch keine gemeinsamen Variablen
  - Verwaltet vom BS mittels Threadkontrollblock
- Betriebssystem
  - Bodensatz an Code, der alles Obige organisiert und am Laufen hält
  - NICHT als Prozesse oder Threads organisiert ! (zumindest nicht in Linux)

Moduswechsel

Der Prozessor arbeitet

- **entweder** im Betriebssystemmodus
  - und bearbeitet Code des Betriebssystems
- **oder** im Benutzermodus
  - und bearbeitet Code des Benutzerprozesses

Wechsel des Modus

- Systemaufruf im Programm (synchron)
- Unterbrechung (asynchron)

Synchronisation im traditionellen Unix-Kern

Feststellung: der UNIX-Kern ist wiedereintrittsfähig (reentrant)

- Mehrere Prozesse arbeiten im Kern zur selben Zeit und evtl. im selben Code-Bereich
- Natürlich nicht echt gleichzeitig sondern verschränkt !
  - Es gibt ja nur einen Prozessor

Frage: Könnte es zu inkonsistenten Daten kommen? Wenn ja, wie vermeidet man es?

- Die Situation könnte auftreten, wenn zwei Prozesse dieselben Daten, z.B. Puffer, benutzen
- Natürlich darf Dateninkonsistenz nicht auftreten

Beispiel einer Problemsituation (1)

- Prozess A will etwas von Datei 1 lesen
- A ruft read() und geht in Systemmodus
- BS legt Puffer 1 an, sendet Befehl an Platte und legt A schlafen
- Neuer lauffähiger Prozess wird ausgesucht: B
- Prozess B will etwas von Datei 2 lesen
- B ruft read() und geht in Systemmodus
- BS legt Puffer 2 an, sendet Befehl an Platte und legt B schlafen
- Neuer lauffähiger Prozess wird ausgesucht: C
- C läuft
- Platte erzeugt Unterbrechung, weil Daten von 1 vorliegen
- C wird unterbrochen und geht in Systemmodus
- BS nimmt Daten entgegen, schreibt sie in Puffer 1
- Neuer lauffähiger Prozess wird ausgesucht: noch immer C

Synchronisation im traditionellen Unix-Kern...

Vermeidung von Inkonsistenzen

- Das Betriebssystem ist zunächst einmal nicht unterbrechbar (non-preemptive)
- D.h. eine BS-Aktivität wird zu Ende geführt, auch wenn dadurch die Zeitscheibe des zugehörigen Prozesses überschritten wird
- Nach dem Ende sind alle Datenstrukturen konsistent und ein anderer Prozess kann unbedenklich damit arbeiten

Aber ...

Synchronisation im traditionellen UNIX-Kern...

Problem: Unterbrechungen

- Während der Aktivität im BS-Kern könnte eine Unterbrechung erfolgen
- Die Unterbrechungsbehandlungsroutine könnte zufällig dieselben Datenstrukturen bearbeiten

Lösung:

- Vorübergehendes Verbieten von Unterbrechungen durch Erhöhung des *ipl* (interrupt priority level)
- Kritischer Bereich mit Erhöhung/Verringerung eingerahmt

Beispiel einer Problemsituation (2)

- Prozess A will etwas von Datei 1 lesen
- A ruft read() und geht in Systemmodus
- BS legt Puffer 1 an, sendet Befehl an Platte und legt A schlafen
- Neuer lauffähiger Prozess wird ausgesucht: B
- Prozess B will etwas von Datei 2 lesen
- B ruft read() und geht in Systemmodus
- BS legt Puffer 2 an, sendet Befehl an Platte und legt B schlafen
- Neuer lauffähiger Prozess wird ausgesucht: C
- C läuft
- Platte erzeugt Unterbrechung, weil Daten von 1 vorliegen
- C wird unterbrochen und geht in Systemmodus
- BS nimmt Daten entgegen, schreibt sie in Puffer 1
- Hier könnte weitere Unterbrechung kommen, weil Daten von 2 vorliegen – an dieser Stelle aber unerwünscht und deshalb unterdrückt
- Danach: BS nimmt Daten von Datei 2 entgegen, schreibt sie in Puffer 2
- Neuer lauffähiger Prozess wird ausgesucht: noch immer C

Synchronisation im traditionellen Unix-Kern...

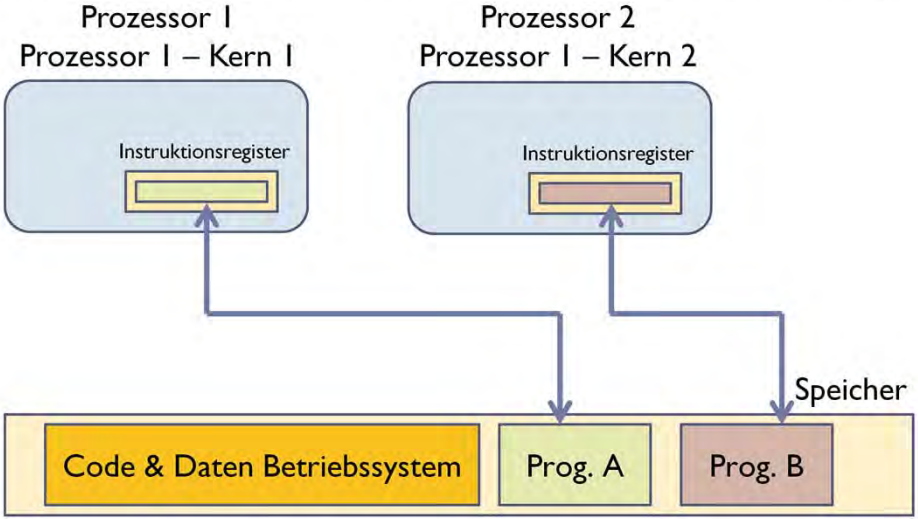
Beim Einprozessorsystem

- Synchronisation problemlos möglich
- Nur kleinere Probleme
- Ununterbrechbarkeit des Kerns ist starker Schutz

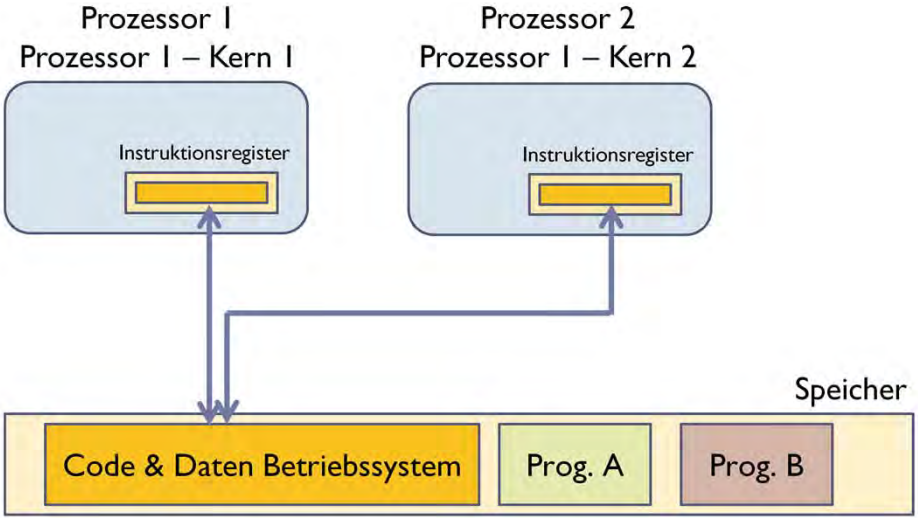
(Bei Echtzeitbetriebssystemen ist die Ununterbrechbarkeit des Kern nicht mehr gegeben – damit neue Probleme)

2. Mehrprozessor/Mehrkern-Systeme

Mehrkern-Systeme sind seit ca. 2005 Standard



Mehrprozessor/Mehrkern-Systeme...



Mehrprozessor/Mehrkern-Systeme...

Unkritisch

- Beide Prozessoren bearbeiten Code verschiedener Programme/Prozesse

Beginn aller Probleme

- Ein Prozessor wechselt in den Systemmodus
  - Wegen Systemaufruf oder Unterbrechung
- Frage: Was macht anderer Prozessor?
  - Systemmodus verbieten? Ineffizient!
  - Systemmodus erlauben? Gefährlich!

Beispiel einer Problemsituation (3)

- Prozess A auf Prozessorkern 1 will etwas von Datei 1 lesen
- A ruft read() und geht in Systemmodus
- Echt gleichzeitig: Prozess B auf Prozessorkern 2 will etwas von Datei 2 lesen
- B ruft read() und geht in Systemmodus
- Ab hier Gefahr, dass beide Prozessorkerne identischen Code echt gleichzeitig bearbeiten, z.B. den zur Pufferverwaltung
- Vermutlich stürzt hier das System ab...

Mehrprozessor/Mehrkern-Systeme...

Was benötige ich alles?

- Spezial-Hardware zur Unterbrechungssteuerung an jedem einzelnen Prozessor
- Cache-Kohärenz-Mechanismen
- **Nebenläufig ausführbares Betriebssystem**
  - **„Threads" im Betriebssystem sind etwas anderes, als Threads in Prozessen**

3. SMP-Hardware

SMP Symmetric Multiprocessing

- Variante des Betriebssystems, die mehrere Prozessoren/Kerne unterstützen kann

Intels Multiprocessor Specification MPS 1.4 (1995)

- Ursprünglich eingeführte Referenz
- Beschreibt Intel-SMP-Systeme
- Festlegung der BIOS-Eigenschaften
- Beschreibung des APIC
  - Advanced Programmable Interrupt Controller
  - Jetzt: x2APIC
- Cache-Kohärenz, MESI-Protokoll
- In der Praxis damals typisch: 2- und 4-Wege-Systeme

Heute enthalten in Advanced Configuration and Power Interface (ACPI)

4. SMP-Betriebssystem

Zunächst eine Begriffsbestimmung

- In einem SMP-System sind alle Prozessoren gleichberechtigt
- Sie greifen gemeinsam auf denselben Code und dieselben Daten des Betriebssystemkerns zu und stehen im Wettbewerb um Systemressourcen
- Jeder Benutzerprozess kann auf jedem Prozessor zur Ausführung gebracht werden

SMP-Betriebssystem...

Was bedeutet das?

- Tatsächlich ist das SMP-Betriebssystem ein paralleles Programm auf einer Maschine mit gemeinsamem Speicher
- Der Code wird nebenläufig ausgeführt
  - (vgl. nebenläufig vs. parallel)
- Die Daten können beliebig manipuliert werden
- Inkonsistenzen vermeidet man durch Sperren
  - Sperren von Code-Abschnitten
  - Sperren von Datenbereichen

SMP-Betriebssystem...

Alles dreht sich um die Sperren

- Eine große Sperre (sog. giant lock):
  Nur ein Prozessor kann in das Betriebssystem
  Daten bzgl. Der Konsistenz geschützt
  - Problem gelöst; Effizienz vernichtet
- Viele kurze Sperren:
  Daten bzgl. der Konsistenz geschützt
  - Gute Nebenläufigkeit bei geringen Kosten durch die Sperren
- Ganz viele sehr kurze Sperren:
  Daten bzgl. der Konsistenz geschützt
  - Bessere Nebenläufigkeit bei höheren Kosten durch die Sperren

SMP-Betriebssystem...

Wie mache ich mein Einprozessor-Einkern-Betriebssystem SMP-fähig?

- Analysiere alle Datenstrukturen und Abläufe im BS-Code
  - Zunächst Subsysteme wie Speicherverwaltung, Scheduling, Ein-/Ausgabe etc.
- Schütze kritische Bereich vor gleichzeitigem Zugriff
- Verfeinere den Schutz (mehr Nebenläufigkeit)
  - Inkrementelle Parallelisierung

SMP-Betriebssystem... Linux

- Bei Linux dauert(e) dieser Vorgang Jahre!
  - Erstmals gut SMP-fähig in Kernel 2.2 (Jan. 1999)
- Im Kernel 2.4 (Jan. 2001)
  - Alle Subsysteme durch feingranulare Sperren abgesichert
  - „Kernel subsystems are fully threaded"
- Skalierbarkeit bzgl. Anzahl der Prozessoren ist problematisch
  - Bisherige Grenze: 4 (?)

5. Synchronisationsmechanismen

- Es müssen jetzt verschiedenste Datenstrukturen geschützt werden, die bei einem einzelnen Prozessor unkritisch waren
- Einfache Flags reichen nicht aus, da diese gleichzeitig manipuliert werden könnten
- Unterbrechungssperren müssen global gesetzt werden können
- Weitere BS-Details
  - Traditioneller Sleep/wakeup-Mechanismus auf Mehrprozessor-Systemen unbrauchbar
  - Wiederaufwecken von Threads kritisch
    - Thundering-herd-problem

Synchronisationsmechanismen...

Essentiell: Hardware-Unterstützung

- **Atomares** Testen-und-Setzen
  - Testet Bit, setzt Wert auf ´1´, gibt alten Wert zurück
  - Nach Abschluß ist das Bit ´1´
  - Rückgabewert ´0´: man hat jetzt Zugriff, Ressource war frei
  - Rückgabewert ´1´: Ressource von anderen belegt
- Anweisung kann nicht einmal durch eine Unterbrechung unterbrochen werden
- In vielen Systemen sogar atomare Nutzung des Speicherbusses

Synchronisation mittels Semaphor

Standardverfahren:

- P() dekrementiert Semaphor und blockiert, wenn Wert kleiner 0 wird
- V() inkrementiert Semaphor und weckt Thread auf, wenn Wert kleiner oder gleich 0 ist

BS-Kern garantiert Atomarität der Aktion

- Einprozessor-Einkern-System:
  durch Ununterbrechbarkeit der BS-Kerns
- Mehrprozessor/Mehrkern-System:
  durch tieferliegende unteilbare Aktion

Synchronisation mittels Semaphor...

Problem

- Blockieren und Aufwecken erfordert Kontextwechsel im Betriebssystem: langsam

- Nicht akzeptabel für kurze Blockierungen
- Weniger aufwendiger Mechanismus benötigt

Synchronisation mittels Spinlock

Einfachster Sperrenmechanismus: Spinlock

- engl: *spin lock, simple lock, simple mutex*
- Skalare Variable
  - ´0´ bedeutet verfügbar
  - ´1´ bedeutet belegt
- Manipulation mittels aktivem Warten (busy-wait) und atomarem Testen-und-Setzen

Synchronisation mittels Spinlock...

muss atomar sein!

```
void spin_lock (spinlock_t *s)  {
  while (test_and_set (s) != 0)  /* belegt */
    ;                            /* warte auf Freigabe */
}


void spin_unlock (spinlock_t *s)  {
  *s = 0;
}
```

Möglicher Nachteil: Busblockierung

Synchronisation mittels Spinlock...

```
void spin_lock (spinlock_t *s)  {
  while (test_and_set (s) != 0) /*belegt*/
    while (*s != 0);
          /* warte auf Freigabe          */
          /* hier nur lesender Zugriff ! */
}

void spin_unlock (spinlock_t *s)  {
  *s = 0;
}
```

Synchronisation mittels Spinlock...

Verwendung von Spinlocks

- Kurzzeitige Sperre kritischer Bereiche im Betriebssystem-Code
- Niemals für blockierenden Code einsetzen

Analyse

- Aktives Warten (normalerweise unerwünscht)
- Sehr billig, wenn Ressource nicht belegt ist
- Insgesamt billig bei geringem Belegt-Grad

6. Erweiterte BS-Funktionalität

Was brauchen wir sonst noch?

- Feingranulare Ausführungsobjekte
  - Kernel-Threads

    Werden im BS-Code erzeugt und teilen sich mit ihm den Adressraum
- Speicherverwaltung für Mehrprozessor-Systeme
  - Verwaltung mehrerer Speicherbänke
- Scheduling für Mehrprozessor-Systeme

Erweiterte BS-Funktionalität

Mehrprozessor-Scheduling

- Prozessor-Affinität

  Ein Prozeß oder ein Thread sollte auf dem Prozessorkern fortgesetzt werden, auf dem er zuletzt lief

  Grund: Gültiger Inhalt im Cache des Prozessorkerns

- Gang-Scheduling

  Alle Threads eines Prozesses sollten zusammen zugeteilt werden

  Grund: Verzögerungen an gemeinsam genutzten Sperren vermeiden

Betriebssystemaspekte
Zusammenfassung

- Synchronisation in Einprozessor-Einkern-Systemen fast ohne Probleme
- Bei SMP-Systemen läuft das Betriebssystem auf allen Prozessoren
- Hauptproblem: innere Synchronisation und Schutz der Datenstrukturen
- Betriebssystem-Code wird durch den Einbau von Sperren parallelisiert
- Feinere Sperren bedeuten mehr Nebenläufigkeit
- Hardware-Unterstützung für die Sperren notwendig
- Semaphor ist zu kostspielig
- Spinlock ist das wichtigste Synchronisationskonzept
- Scheduling von Threads ist sehr komplex

Betriebssystemaspekte
Die wichtigsten Fragen

- Wie funktioniert Synchronisation im traditionellen UNIX-Kern?
- Was versteht man unter Wiedereintrittsfähigkeit?
- Wie werden Maschinenbefehle in Mehrprozessor/Mehrkern-Systemen abgearbeitet?
- Was benötige ich für ein Mehrprozessor-System?
- Was kennzeichnet ein SMP-Betriebssystem?
- Welche Varianten von Sperren finden wir im SMP-Betriebssystem?
- Wie wird ein Betriebssystem SMP-fähig?
- Wie funktioniert Synchronisation mittels Semaphor?
- Wie funktioniert Synchronisation mittels Spinlock?
- Welche weiteren Funktionen muss ein SMP-Betriebssystem aufweisen?