DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Betriebssysteme

Kapitel 2 Prozesse und Threads

Ziele der Vorlesung

- Einführung
  - -Historischer Überblick
  - -Betriebssystemkonzepte
- Prozesse und Threads
  - -Einführung in das Konzept der Prozesse
  - -Prozesskommunikation
  - -Scheduling von Prozessen
  - -Threads
- Speicherverwaltung
  - -Einfache Speicherverwaltung
  - -Virtueller Speicher
  - -Segmentierter Speicher
- Dateien und Dateisysteme
  - -Dateien
  - -Verzeichnisse
  - -Implementierung von Dateisystemen
- Grundlegende Eigenschaften der I/O-Hardware
  - -Festplatten
  - -Terminals
  - -Die I/O-Software
- Deadlocks/Verklemmungen
- Virtualisierung und die Cloud
- Multiprozessor-Systeme
- IT-Sicherheit
- Fallstudien

Aufgabe/Frage

Fragen

- Was ist ein Prozess?
- Was ist der Unterschied zwischen Programm und Prozess?
- Kennen Sie ein Analogon!

  Hilfe: Was kann das Betriebssystem durch Prozesse realisieren?

Bedingungen:

→ im Team?

→ Zeit: 5 min

Lösung

Antwort

- Das Betriebssystem ist als Menge von (sequenzielle) Prozessen organisiert.
- Ein Prozess ist ein Programm in Ausführung.
- Bei Multiprogrammierung wird zwischen Prozessen hin hergeschaltet.
- Jeder Prozess umfasst eine virtuelle CPU mit eigenem Adressraum

- Ein Analogon zu Prozessen ist das Kuchen backen:
  - das Programm ist das Kuchenrezept
  - Der Prozess ist die Aktivität des Backens

Prozesse

- Moderne Computer erledigen viele Dinge gleichzeitig
- Wie funktioniert z.B. ein Webserver?
  - Es kommen viele Anfragen nach Webseiten
  - Bei Eintreffen der Anfrage überprüft der Server den Cache-Inhalt
  - Seite gefunden, angefragte Seite wird zurückgeschickt
  - Seite nicht gefunden, Starten einer Plattenanfrage, um die entsprechende Seite zu holen.
  - Plattenanfrage dauert; CPU arbeitet andere Anfragen ab.

    → Nebenläufigkeit wird durch Prozesse (und Threads) modelliert und gesteuert

Prozesse

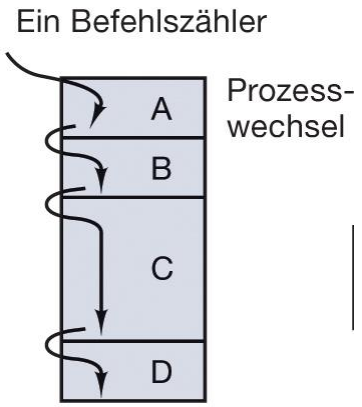
- Beim Hochfahren werden viele Prozesse heimlich gestartet (Benutzer merkt das nicht)
- All diese Aktivitäten müssen verwaltet werden
- Multiprogrammiersystem unterstützt mehrere Prozesse
- CPU wechselt schnell von Programm zu Programm (jedes Programm rechnet ca. 10 bis 100 ms)
- Es läuft immer nur ein Programm auf der CPU
- Im Zeitraum von 1 sec können mehrere Programme rechnen
  - → Quasiparallelität bzw. Pseudoparallelität
  - → Gegensatz zu Hardwareparallelität von Multiprozessorsystemen
- Konzeptionelles Modell der sequentiellen Prozesse

Prozesse

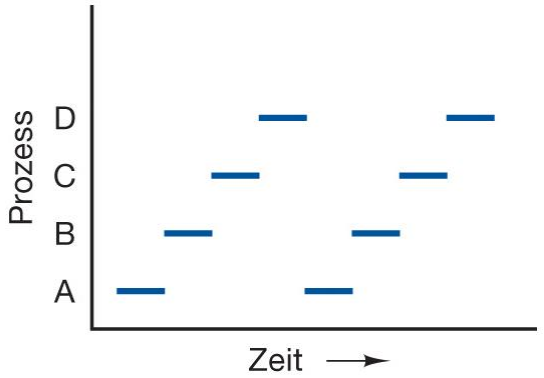
Das Prozessmodell

- **Das Betriebssystem ist als Menge von (sequenziellen) Prozessen organisiert**
- ein Prozess ist die Instanz eines Programms in Ausführung, inkl. Des aktuellen Wertes des Befehlszählers, der Registerinhalte und der Belegungen der Variablen
- Konzeptionell besitzt jeder Prozess seine eigene virtuelle CPU
- Es gibt eine Menge von (quasi-)parallel laufenden Prozessen
- Das schnelle Hin- und Herschalten zwischen den Prozessen wird als Multiprogrammierung bezeichnet (ein Umladen der Register ist notwendig).

Prozesse



Speicher eines Computers mit 4 parallelen Programmen



Alle Prozesse sind über ein längeres Zeitintervall in ihrere Ausführung fortgeschritten, obwohl zu einem beliebigen Zeitpunkt nur ein Prozess tatsächlich läuft

- Prozesse haben einen Adressraum
  - Inklusive logischen Befehlszähler
  - Konsequenz:
    - Laufzeit eines Programms ist nicht mehr reproduzierbar!
    - Programme dürfen keine Annahmen über den Zeitablauf enthalten

Prozesse: Weitere Betrachtung

Wir gehen im weiteren davon aus, daß es im betrachteten System nur eine CPU mit einem Core gibt

Prozesse

- Analogon Programm und Prozess
  - Kuchen backen:
    - Das Programm ist das Kuchenrezept mit allen Anweisungen
    - Der Informatiker ist der Prozessor (CPU)
    - Die Zutaten sind die Eingabedaten
    - Der Prozess ist die Aktivität des Backen (Rezept lessen, Zutaten herbeiholen, backen)
  - Der Informatiker bekommt einen Anruf während er das Rezept abarbeitet
  - Der Informatiker notiert sich die Stelle des Rezepts, wo er sich gerade befindet
  - Der Informatiker telefoniert (weil höhere Prio)
  - Nach Ende des Anrufs fährt der Informatiker an der Stelle fort, wo er unterbrochen hatte
  - Beachte: Backen von 2 Kuchen heißt auch 2 mal Backen
    → Ein Programm 2x starten, ergibt auch 2 Prozesse!

Prozesse

Prozesserzeugung

- Einfache Systeme: alle benötigten Prozesse werden nach dem Systemstart vorhanden sein.
- Allgemein: Prozesse müssen je nach Bedarf im laufenden Betrieb erzeugt und beendet werden
- Prozesse können im Vordergrund (z.B. Interaktion mit Benutzer) und im Hintergrund (keine Zuordnung zu bestimmten Personen, z.B. Daemon) laufen.
  Unix = ps (process status=Anzeige der im Hintergrund laufenden Prozesse)
  Windows = Taskmanager

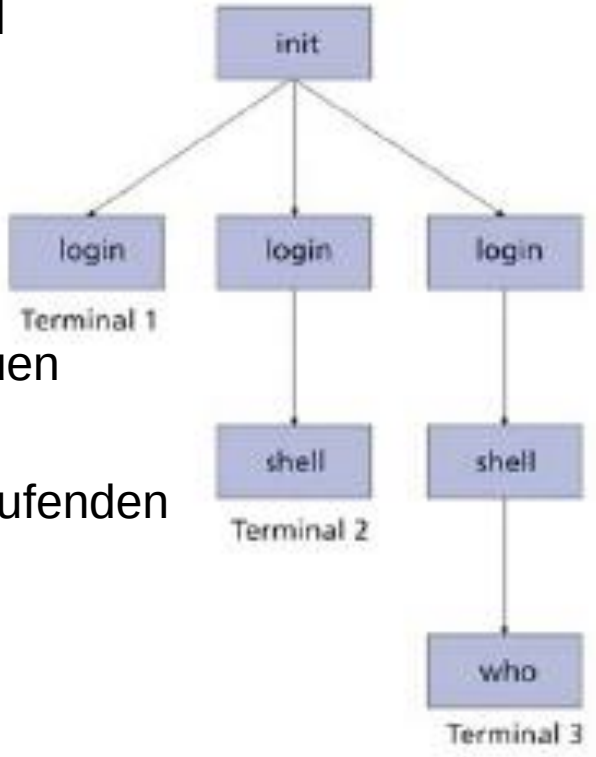
Prozesse

Prozesserzeugung

- Ereignisse, die die Erzeugung eines Prozess verursachen
  - System-Initialisierung
    - Initialisierung, meist als Hintergrundprozess (durch Betriebssystem)
  - Durch einen anderen Prozess
    - Systemaufruf
  - Benutzeranfrage, einen neuen Prozess zu erzeugen
    - in interaktiven Systemen wird durch (Doppel-)Klicken eines Icons ein neuer Prozess gestartet; darin läuft das gewählte Programm ab
  - Initiierung einer Stapelverarbeitung (Stapeljob)
    - Stapeljobs werden abgearbeitet; bei genügend Betriebsmittel erzeugt Betriebssystem neuen Prozess und verarbeitet die nächste Aufgabe, z.B. Bestandsverwaltung
- Technisch: es wird immer ein neuer Prozess erzeugt, indem ein bestehender Prozess einen Systemaufruf zur Prozesserzeugung ausführt.

Prozesse

- Prozesserzeugung
  - Beispiel: Unix starten
    - init liest Datei /etc/init/tty?.conf
    - Startet dann ein login für jedes Terminal
    - Nach der Anmeldung startet eine Shell, von der aus die nächsten Prozesse gestartet werden können.



  - Unix-Systemaufruf zur Erzeugung eines neuen Prozesses: fork
    - fork erzeugt eine exakte Kopie des aufrufenden Prozesses

Prozesse

Prozessbeendingung

- Bedingungen:
  - Normales Beenden (freiwillig)
    - Wenn Aufgabe erledigt ist! (exit bzw. exitProcess)
  - Beenden aufgrund eines Fehlers (freiwillig)
    - Der Prozess stellt einen Fehler fest, der nicht vom Programm verursacht wurde (z.b. Datendatei nicht vorhanden)
  - Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig)
    - Der Prozess selbst verursacht einen Fehler
    - Ausführen eines unzulässigen Befehl
    - Zugriff auf ungültige Speicheradresse
  - Beenden durch anderen Prozess (unfreiwillig)
    - kill bzw. TerminateProcess
    - Berechtigung notwendig

Aufgabe/Frage

Fragen

- Wie beenden Sie z.B. ein Textverarbeitungsprogramme, Internetbrowser oder ähnliche Programme?

- Was passiert bei interaktiven Programmen, wenn falsche Parameter angegeben wurden?

Bedingungen:

→ im Team?

→ Zeit: 3 min

Lösung

Antwort

- Durch ein Symbol oder Menüpunkt

- Bei interaktiven Programmen wird ein Dialogfenster geöffnet, in dem der Benutzer den Parameter eingeben kann.

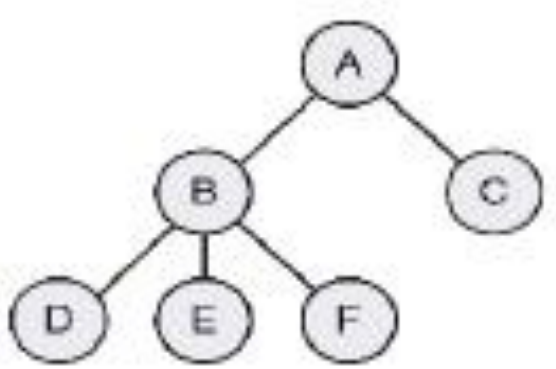
Prozesse

Prozesserzeugung Unix

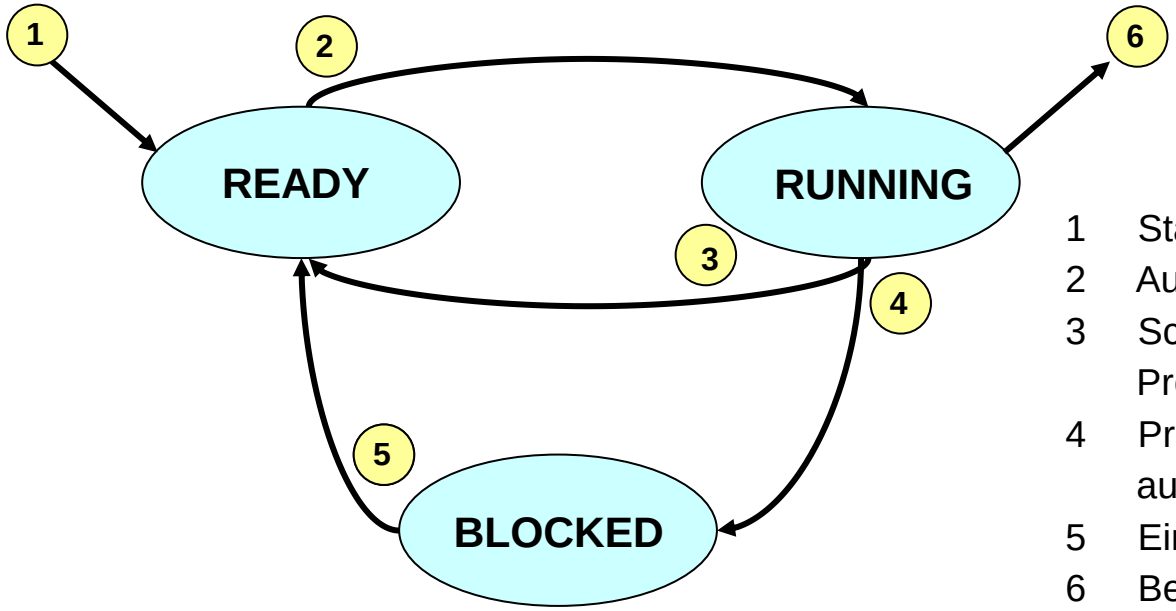
- Systemaufruf zur Erzeugung eines neuen Prozesses: fork
- fork erzeugt eine exakte Kopie des aufrufenden Prozesses mit dem gleichen Speicherabbild, die gleichen Umgebungsvariablen und die gleichen geöffneten Dateien

Prozesshierarchien

- Beziehungen zwischen Prozessen, z.B. Eltern-Kind
- Erzeugt Kindprozess weitere Prozesse: Prozesshierarchie
- Kindprozess hat ‘nur’ ein Elternteil
- Unix:
  - Prozess bildet mit all seinen Kindern und weitere Nachkommen eine Prozessfamilie
  - Signale werden an alle Mitglieder dieser Familie verschickt
    - Jeder Prozess entscheidet, wie auf Signal reagiert wird!
- Windows
  - Kein Konzept der Prozesshierarchie
  - Alle Prozesse sind gleichwertig
  - Beziehungen zwischen Eltern und Kind-Prozess werden durch spezielle Tokens (Handle) gesteuert!



- Prozess-Zustände
  - Wichtig für die Verwaltung der Ressourcen durch Betriebssysteme
    - **Running** (rechnend, die Befehle werden im Moment auf der CPU ausgeführt)
    - **Ready** (rechenbereit, kurzzeitig gestoppt, um einen anderen Prozess rechnen zu lassen)
    - **Blocked** (blockiert, nicht lauffähig bis best. externes Ereignis eintritt)

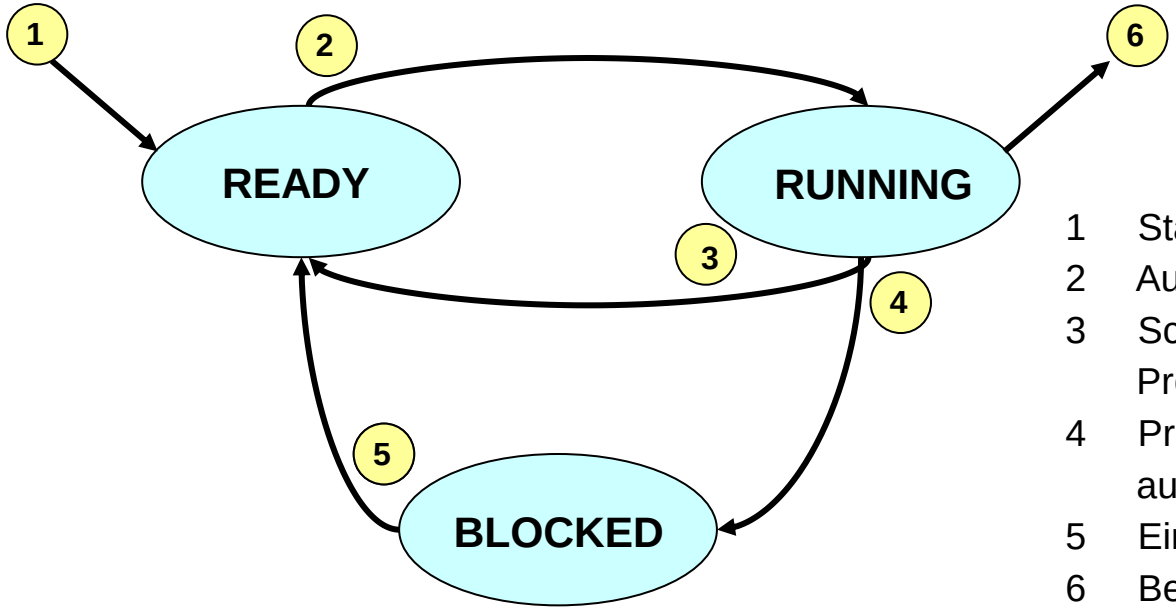


1 Start nach Erzeugung
2 Auswahl durch Scheduler
3 Scheduler wählt anderen Prozess
4 Prozess blockiert, weil er auf Eingabe wartet
5 Eingabe vorhanden
6 Beendigung des Prozesses

Verwaltung der Übergänge durch Scheduler

Prozesse

- Prozess-Zustände
  - 4= Prozess blockiert, wenn das BS entdeckt, dass ein Prozess im Augenblick nicht fortfahren kann
  - 3= Scheduler entscheidet, dass ein Prozess lange genug gelaufen ist und nun ein anderer Prozess Rechenzeit bekommen soll.
  - 2= alle anderen Prozesse haben gerechten Rechenanteil an Rechenzeit erhalten, erstem Prozess wird die CPU wieder zugeteilt
  - 5= externes Ereignis tritt ein, auf das ein Prozess gewartet hat (z.B. Eingabedaten)

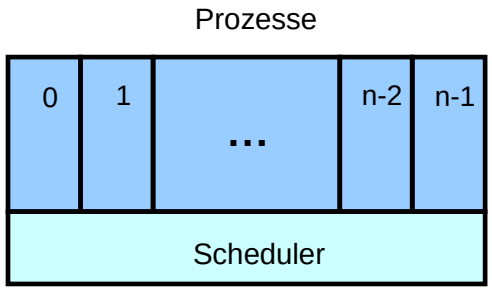


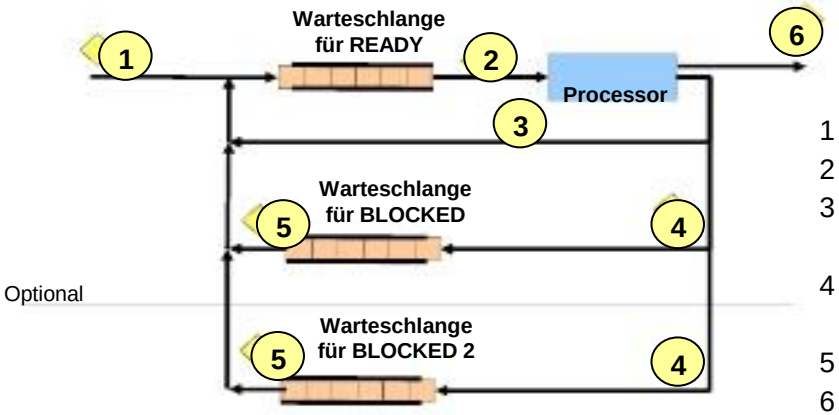
1 Start nach Erzeugung
2 Auswahl durch Scheduler
3 Scheduler wählt anderen Prozess
4 Prozess blockiert, weil er auf Eingabe wartet
5 Eingabe vorhanden
6 Beendigung des Prozesses

→ **Verwaltung der Übergänge durch Scheduler**

Prozesse

- Scheduler
  - Verwaltet die Prozesse
  - Über dem Scheduler liegen eine Vielzahl unterschiedlicher Prozesse
  - Unterste Schicht des Betriebssystems
  - Es wird behandelt
    - Unterbrechungen (Interrupts)
    - Starten und Stoppen von Prozessen
    - Warteschlangen für Prozesse



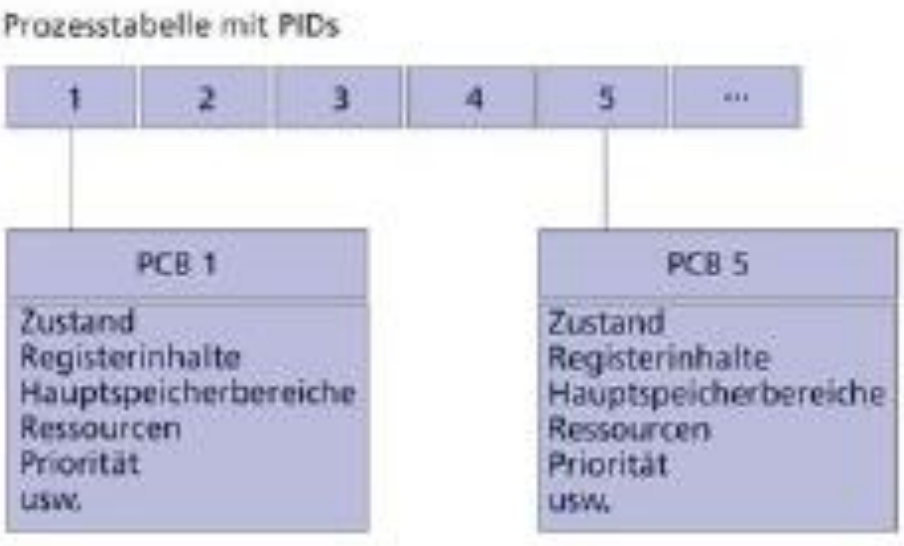


1 Start nach Erzeugung
2 Auswahl durch Scheduler
3 Scheduler wählt anderen Prozess
4 Prozess blockiert, weil er auf Eingabe wartet
5 Eingabe vorhanden
6 Beendigung des Prozesses

Prozesse

- Datenmodell zur Verwaltung von Prozessen
  - Betriebssystem pflegt eine **Prozesstabelle**
    - Ein Eintrag pro Prozess (ProcessControlBlock (PCB))
    - PCB enthält alle Informationem über Prozess
      - Alle Informationen in einer Datenstruktur/Datenmodell
      - Zur **Prozessverwaltung**
        - Register, Befehlszähler, PSW, Stack
        - Prozessidentifikation
          - Kennung des Prozesses (P-ID) und des Elternprozesses
          - Benutzerkennung
        - Zustandsinformation
          - Priorität, verbrauchte CPU-Zeit, Signale,…
      - Zur **Speicherverwaltung**
        - Zeiger auf Textsegment, Datensegment, Stacksegment
      - Zur **Dateiverwaltung**
        - Wurzelverzeichnis, Arbeitsverzeichnis, offene Dateien, Benutzer-Id, Gruppen-ID, …

- Prozesswechsel
  - mit Hilfe der Prozesstabelle
    - Beispiel: Interrupt von E/A-Gerät („Daten stehen bereit“)

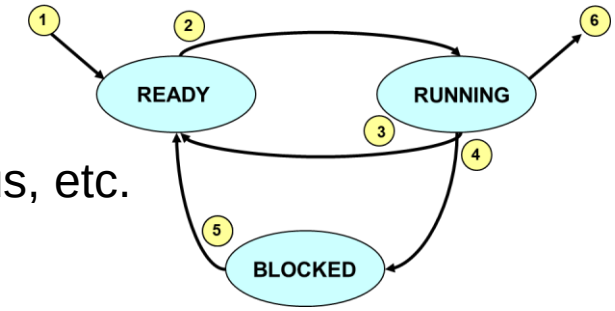


Beispiel

- Prozess 1 läuft, es kommt ein Interrupt (z.B. Daten stehen bereit)
- Alles von Prozess 1 wird auf dem aktuellen Stack gesichert
- Sprung zu Prozess 5

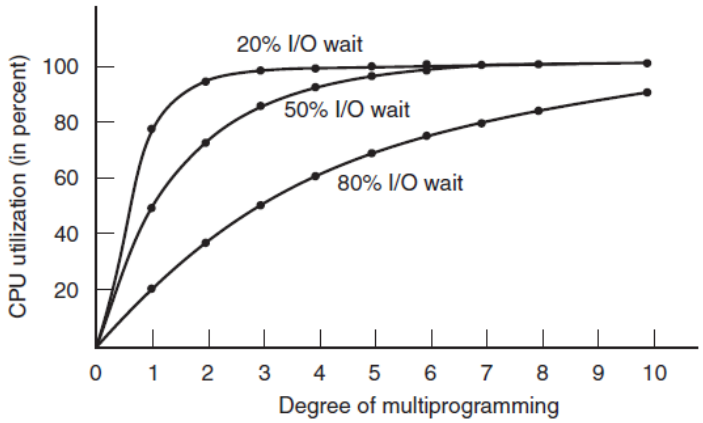
    - Wechseln zwischen zwei PCB

- Prozesswechsel
  - Ablauf
    - Sichern des Befehlszählers, Prozessorstatus, etc.
      - ➢ Prozess auf „Blocked“ setzen
    - Ursache der Unterbrechung ermitteln
    - Ereignis (z.B. Ende der E/A) entsprechend behandeln
      - ➢ Blockierte Threads auf „READY“ setzen
    - Sprung zum Scheduler
    - Scheduler entscheidet welcher Prozess als nächstes läuft



Aufgabe/Frage

- N=Prozesse, die gleichzeitig im Speicher gehalten werden
- P=E/A-Anteil eines Prozesse
- CPU-Ausnutzung= $1 - P^N$



- Computer hat 8 GB Speicher
- Betriebssystem und Verwaltungstabellen beanspruchen 2 GB
- Jedes Benutzerprogramm nimmt 2 GB ein
- Durchschnittliche Ein/Ausgabewartezeit ist 80%

Fragen:

- Wieviel Benutzerprogramme können gleichzeitig im Speicher gehalten werden?
- Wie hoch ist die CPU-Ausnutzung in Prozent?
- Was passiert, wenn man dem System weitere 8 GB spendiert?

Bedingungen:

→ im Team?,

→ Zeit: 10 min

10 min

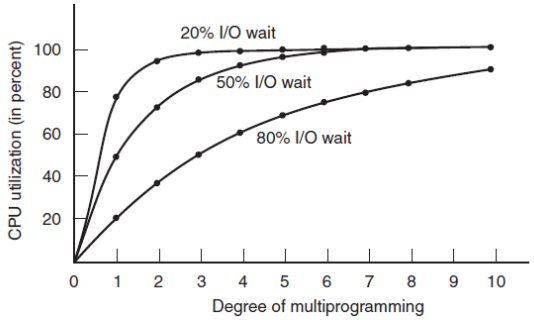
Lösung

Antwort

- Drei (3*2=6+2GB=8 GB) Benutzerprogramme
- 1-0,8 hoch 3=0,512, 0,512 entspricht 49% CPU-Ausnutzung (Betriebssystemaufwand wird ignoriert)

- Es können 7 Benutzerprogramme gleichzeitig laufen (7*2=14+2GB=16GB)
- 1-0,8 hoch 7=0,2097152 entspricht 79% CPU-Ausnutzung (Betriebssystemaufwand wird ignoriert)

- Computer hat 8 GB Speicher
- Betriebssystem und Verwaltungstabellen beanspruchen 2 GB
- Jedes Benutzerprogramm nimmt 2 GB ein
- Durchschnittliche Ein/Ausgabewartezeit ist 80%



Prozesse

- In traditionellen Betriebssystemen
  - Jeder Prozess hat eigenen Adressraum
  - mit einem Ausführungsfaden (thread of control)

- Wünschenswert: Ablauf mehrerer Ausführungsfäden in ein und demselben Adressraum quasiparallel, als ob es einzelne Prozesse wären (‘ein Prozess innerhalb eines Prozesses?’)

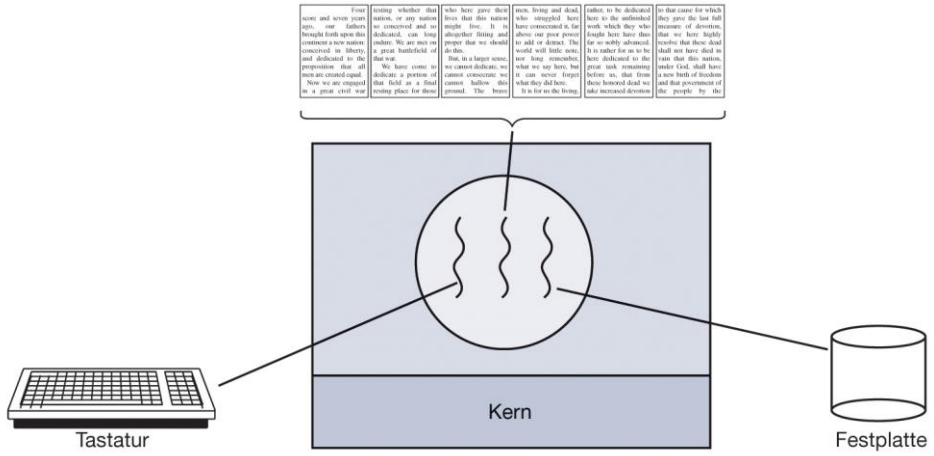


Abbildung 2.7: Ein Textverarbeitungsprogramm mit drei Threads

Prozesse und Threads

- Threads
  - Miniprozess
    - in vielen Anwendungen sind mehrere Aktivitäten auf einmal zu erledigen;
    - einige der Aktivitäten könnten blockieren; Zerlegung der Anwendung in mehrere quasiparallel laufende sequentielle Threads
    - Mehrere Ausführungsfäden in ein und demselben Adressraum
      - ➢ Daten werden gemeinsam benutzt
    - Ausführung im Benutzermodus möglich
    - Erstellung ist 10-100 mal schneller als Prozesse

Prozesse und Threads

- Prozessmodell basiert auf zwei unabhängigen Konzepten:
  - **Konzept der Ressourcen-Bündelung für Prozesse**
    - Prozesse gruppieren zusammengehörende Ressourcen z.B. Adressraum, geöffnete Dateien, Signale, Kindprozesse, …
    - die Zusammenfassung zu einem Prozess bedeutet auch leichtere Verwaltung
  - **Konzept der Ausführung von Prozessen mittels Threads**
    - Threads sind die Einheiten, die für die Ausführung auf der CPU verwaltet werden
    - Thread besitzt
      - Befehlszähler (sagt aus, welcher Befehl als nächstes ausgeführt wird),
      - Register für lokale Variablen,
      - Stack.
    - Mehrere Ausführungsfäden sind nun möglich (Multi-Threading)
      - Hardware-Unterstützung (schnelles Umschalten in nsec)

Aufgabe/Frage

- Erklären Sie die Nützlichkeit von Threads anhand des Beispiels eines Textverarbeitungsprogramm
- Welche Threads könnte es aus Ihrer Sicht geben?
- Warum würde die Verwendung von drei getrennten Prozessen hier nicht funktionieren?

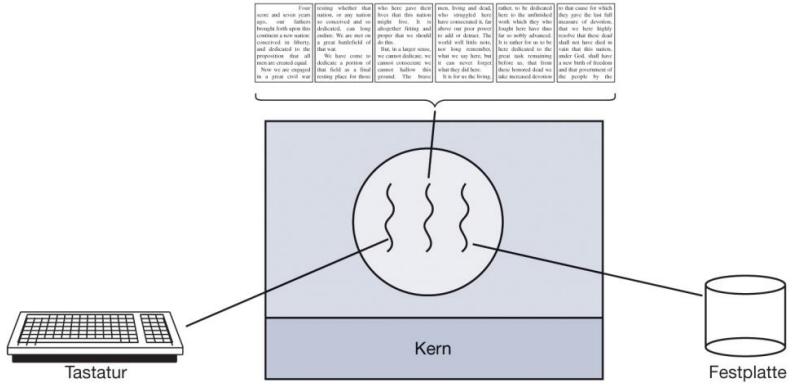


Abbildung 2.7: Ein Textverarbeitungsprogramm mit drei Threads

Bedingungen:
→ im Team?,
→ Zeit: 5 min

5min

Lösung

Antwort

- Das Textverarbeitungsprogramm empfängt Eingabe von der Tastatur
- Das Textverarbeitungsprogramm zeigt das erzeugte Dokument auf dem Bildschirm genau so formatiert an, wie es im Ausdruck erscheinen wird.
- Das Textverarbeitungsprogramm speichert alle paar Minuten die gesamte Datei
- Das Textverarbeitungsprogramm ist in 3 Threads geschrieben.

- Alle drei Threads müssen auf dem gleichen Dokument operieren.
- Die drei Threads nutzen im Gegensatz zu drei Prozessen den gemeinsamen Speicher und haben alle Zugriff auf das zu bearbeitende Dokument.

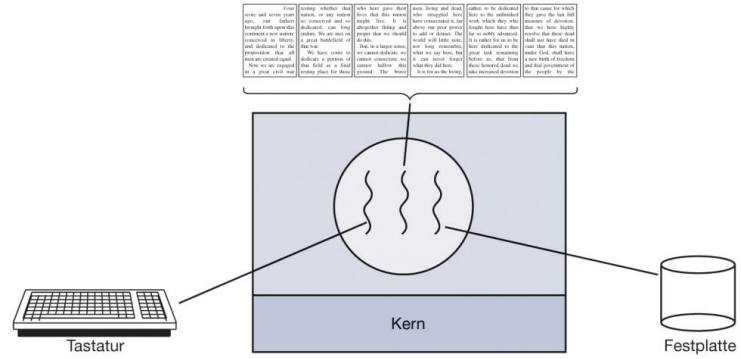


Abbildung 2.7: Ein Textverarbeitungsprogramm mit drei Threads

- Threads

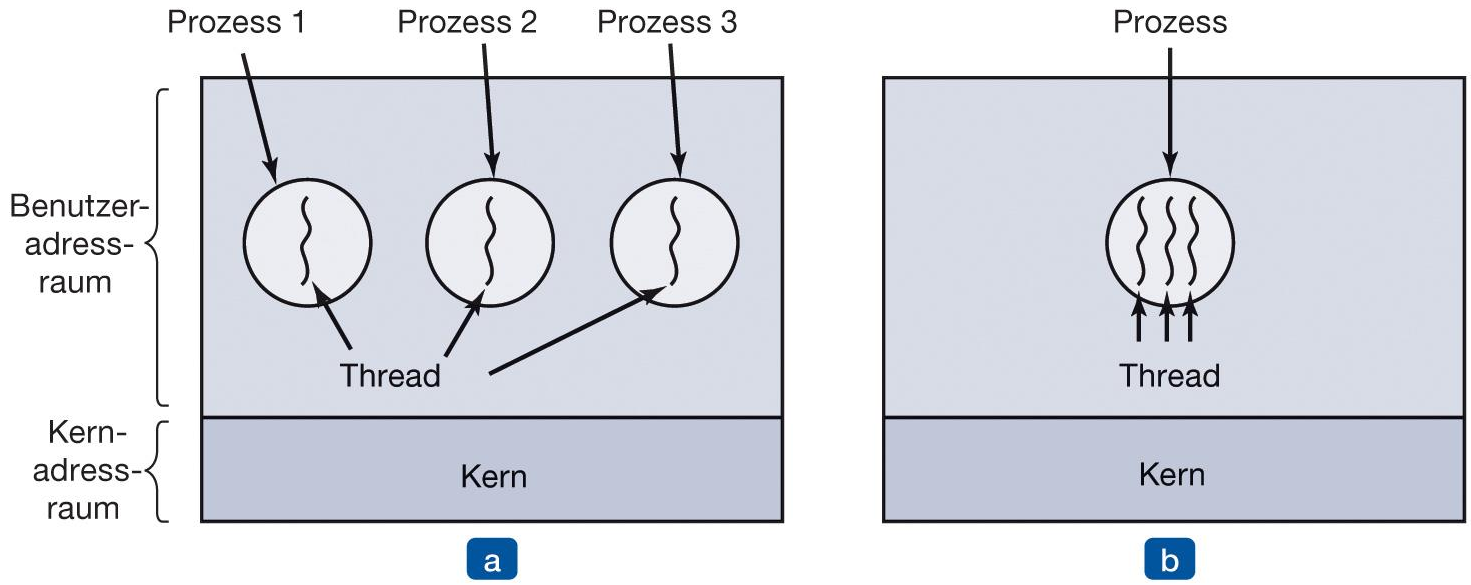


Abbildung 2.11: (a) Drei Prozesse mit je einem Thread (b) Ein Prozess mit drei Threads

In beiden Bildern existieren 3 Threads, Abb. 2.11 (a) jeder Thread in einem Adressraum, Abb. 2.11(b) alle drei Threads benutzen denselben Adressraum und somit auch die globalen Variablen

Prozesse und Threads

- In einem Prozess mit mehreren Threads auf einem 1 Core-System wechseln sich die Threads in der Ausführung ab.
- Durch das Hin- und Herwechseln zwischen mehreren Prozessen gibt es die Illusion von unabhängigen parallel laufenden sequenziellen Prozessen.
- Multithreading:
  - CPU wechselt schnell zwischen Threads hin und her und erzeugt die Illusion, dass alle Threads parallel laufen, wenn auch auf einer langsameren CPU.
  - Beispiel: Drei rechenintensive Threads laufen in einem Prozess; scheinbar laufen sie parallel auf je einem Core (mit einem Drittel der Geschwindigkeit der realen CPU)
- Verschiedene Threads in einem Prozess sind nicht so unabhängig voneinander wie verschiedene Prozesse. Sie benutzen den selben Adressraum und haben somit auch die gleichen globalen Variablen.
- Jeder Thread darf auf jede Speicheradresse innerhalb des Adressraums des Prozesses zugreifen.
- Ein Thread kann den Stack eines anderen Threads lesen, schreiben oder ihn löschen. Es gibt keinen Schutz.
- Die Threads teilen sich einen Adressraum, einen Satz an geöffneten Dateien, Kindprozessen, Alarme, Signale usw.

Prozesse und Threads

- Threads

Prozesseigenschaften	Threadeigenschaften
Elemente pro Prozess	**Elemente pro Thread**
Adressraum	Befehlszähler
Globale Variable	Register
Geöffnete Dateien	Stack
Kindprozesse	Zustand
Ausstehende Signale	
Signale und Signalroutinen	
Verwaltungsinformationen	

Abbildung 2.12: Die erste Spalte führt Elemente auf, die alle Threads des Prozesses teilen. Die zweite Spalte zeigt Elemente, die zu einem individuellen Thread gehören.

Beispiel: Thread öffnet Datei, Datei ist auch für die anderen Threads des Prozesses sichtbar.
Wichtig: Der Prozess ist die Einheit der Ressourcenverwaltung und nicht der Thread.

Prozesse und Threads

- Threads
  - Jeder Thread hat seinen eigenen Stack.
  - Der Stack eines jeden Threads enthält einen Rahmen für jede aufgerufene Prozedur (lokale Variablen der Prozedur, Rücksprungadresse bei Beendigung der Prozedur)
- Thread-Zustände
  - Gleiche Zustände wie Prozesse
  - RUNNING, BLOCKED, READY
- Thread-Verwaltung
  - Thread-Tabelle
  - Befehlszähler, Register,
  - Stack und Zustand

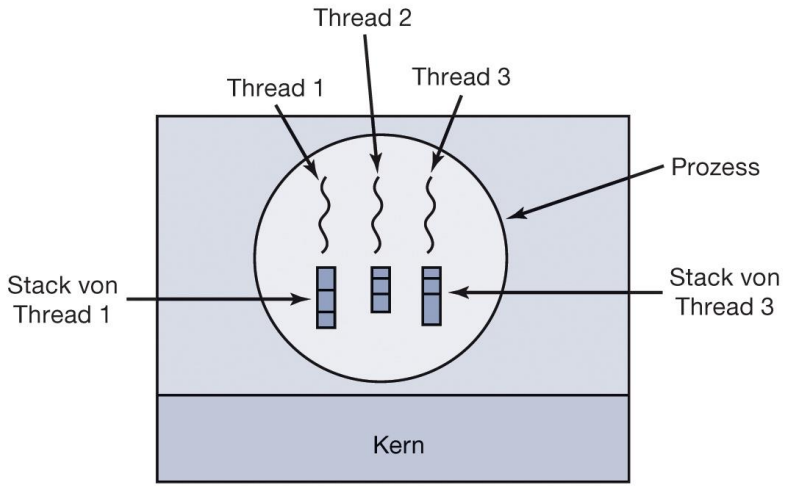


Abbildung 2.13: Jeder Thread hat seinen eigenen Stack.

- Threads
  - Lebenszyklus
    - Erzeugung von Threads
      - Prozess startet einen Thread, der neuen Thread erzeugt (create)
        - Argument: Name der Prozedur, die ausgeführt werden soll
        - Keine Angabe des Adressraums notwendig
      - Neuer Thread läuft im Adressraum des erzeugenden Threads
    - Beendigung
      - Nach Beendigung der Aufgabe durch den Thread selbst (exit)
      - Synchronisation
        - Warten auf die Beendigung eines bestimmten Threads (join)
      - Freiwilliger Verzicht
        - Ressourcen an die CPU zurückgeben (yield)
        - Scheduler wird beauftragt anderen Thread zu starten

POSIX Threads

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

- Threads
  - Threads können im Benutzer- oder im Kernadressraum implementiert sein
  - **Thread im Benutzeradressraum**
    - Kern weiß nichts von den Threads
    - Kern meint, er verwalte gewöhnliche Prozesse mit einem einzigen Thread

      → Vorteil: ein Thread-Paket auf Benutzerebene kann auch auf einem Betriebssystem realisiert werden, das keine Threads unterstützt (Implementierung durch Bibliothek)

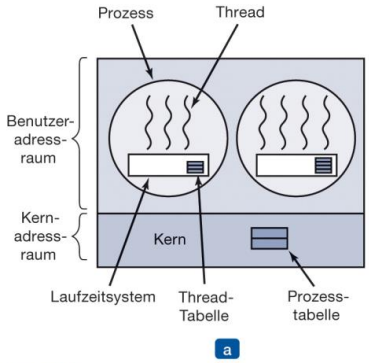


Abbildung 2.16: (a) Ein Thread-Paket auf Benutzerebene

- Threads
  - Threads können im Benutzer- oder im Kernadressraum implementiert sein
  - **Thread im Kernadressraum**
    - Kern kennt und verwaltet Threads
    - Eine Thread-Tabelle
    - Kein Laufzeitsystem
    - Alle Aufrufe, die einen Thread blockieren, sind als Systemaufrufe realisiert

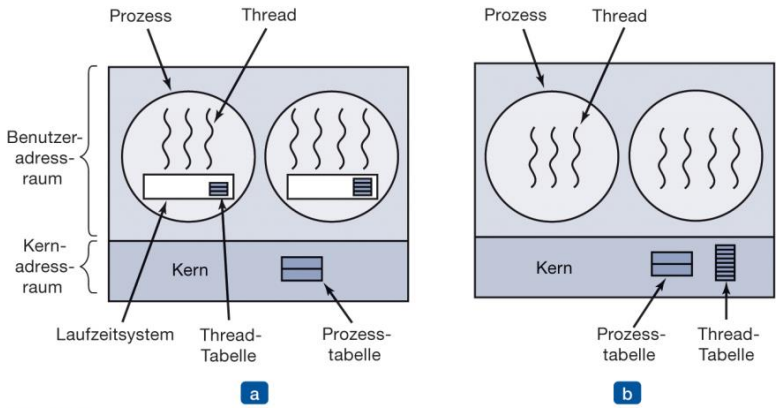


Abbildung 2.16: (a) Ein Thread-Paket auf Benutzerebene (b) Ein Thread-Paket, verwaltet vom Kern

Prozesse und Threads

- Threads (Struktur)
  - Threads laufen auf der Basis eines Laufzeitsystems, das aus einer Sammlung von Prozeduren besteht.
  - Werden die Threads im Benutzeradressraum verwaltet, braucht jeder Prozess seine eigene private Thread-Tabelle, um den Überblick zu behalten.
  - Thread-Tabelle analog Prozesstabelle, es werden nur die Eigenschaften einzelner Threads verwaltet, z.B. Befehlszähler, Stackpointer, Register, Zustand usw.
  - Beim Übergang in rechenbereit oder blockiert, werden DIE Informationen in der Thread-Tabelle gespeichert, die es ermöglichen, den Thread später wieder zu starten.

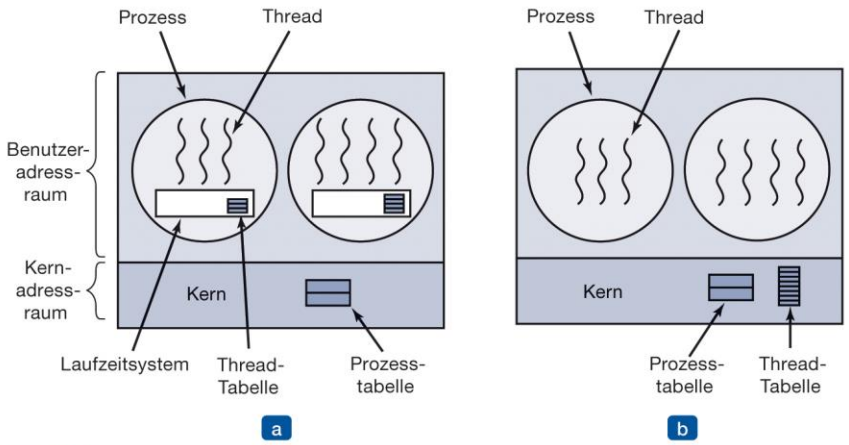


Abbildung 2.16: (a) Ein Thread-Paket auf Benutzerebene (b) Ein Thread-Paket, verwaltet vom Kern

- Threads
  - Thread-Verwaltung – Gegenüberstellung User vs. Kernmodus
    - **Benutzermodus**
      - Pro
        - Schneller Wechsel, da Speicher-prozedur und Scheduler lokale Proze-duren sind, (es ist kein Sprung in den Kern notwendig)
        - Jeder Prozess kann seine eigene angepasste Schedulingstrategie haben
      - Con
        - Umgang mit blockierenden System-aufrufen
        - Ein Thread kann den gesamten Prozess blockieren; z.B. beim Aufruf eines Befehls/Sprungs, der nicht im Speicher ist, blockiert Prozess → Seitenfehler
    - **Kernmodus**
      - Pro
        - Kein Laufzeitsystem;
        - es gibt keine Thread-Tabelle in jedem Prozess; Kern hat Thread-Tabelle, die alle Threads im System verwaltet
        - Effizienter Umgang mit blockierenden Systemaufrufen; alle Aufrufe, die einen Thread blockieren könnten, sind als Systemaufrufe realisiert → hohe Kosten
      - Con
        - Höhere Kosten bei Thread-Wechsel durch Systemaufrufe
        - Umgang mit Signalen

- Threads
  - Threadwechsel
    - erfolgt immer dann, wenn Betriebssystem die Kontrolle erhält
      - bei Systemaufrufen
        - Thread gibt Kontrolle freiwillig ab!
      - Bei Ausnahmen
      - Bei Interrupt
        - Periodischer Timer-Interrupt stellt sicher, daß kein Thread die CPU monopolisiert
    - Scheduler des Betriebssystems entscheidet, welcher Thread als nächster rechnen soll
      - falls nötig, wird dann auch der Prozess gewechselt
      - Scheduler kann die Kosten für den Threadwechsel berücksichtigen

Zusammenfassung

- Prozessmodell
  - Prozess: **Einheit der Ressourcenverwaltung**
    - Adressraum, geöffnete Dateien, Signale, Prioritäten, ...
  - Thread: **Einheit der Prozessorzuteilung**
    - Befehlszähler, CPU-Register, Stack, Thread-Zustand, …
      → mehrere Threads pro Prozess möglich
  - Prozess/Thread-Zustände
    - RUNNING, READY, BLOCKED
    - Warteschlangen

Zusammenfassung

- Realisierungsvarianten für Threads
  - Im Kernmodus (gängig), im Benutzermodus
  - Threadwechsel
    - Bei Systemaufrufen, Ausnahmen oder Interrupts
    - Umladen des Prozessorkontext
      - ➢ beim Prozesswechsel auch Wechsel des Speicherabbilds
- Programmierschnittstellen
  - Posix-Threads
  - Java-Threads

Aufgabe/Frage

Durch das Prozessmodell ist es möglich in einem System Multi-programmierung zu realisieren!

Fragen:

- Welche neuen Herausforderungen(/Probleme) können im Gegensatz zu Systemen mit „einem Programm in Ausführung“ nun auftreten?

Bedingungen:

→ im Team?

→ Zeit: 3 min

Lösung

- Interaktion zwischen Prozessen
  - Prozess nutzen gemeinsame Ressourcen
  - Aber: Prozesse haben getrennte Adressräume
    - Geräte, Dateien, …
  - Unbewußt (Wettstreit)
    - „ICH schreib in die Datei A!“
  - Bewußt (Kooperation durch Teilen)
    - „Ich verwende die Datei X.dat, um die Ressource X zu verwalten!“
  - Prozesse können sich auch kennen (Kooperation durch Kommunikation)
    - Über Prozess ID
- Strukturierte Art und Weise der Kommunikation notwendig

  → **Interprozess--Kommunikation (z.B. mit P-ID)**

Interprozess-Kommunikation

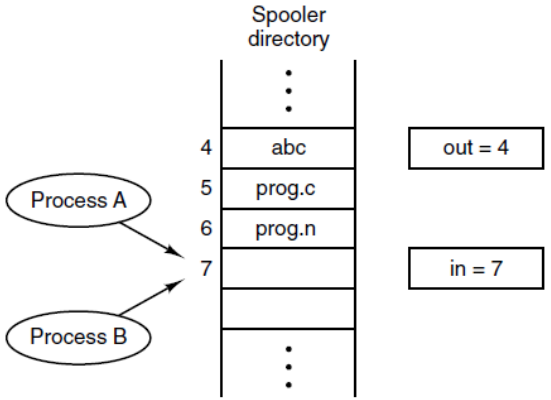
- Prozesse kommunizieren ständig mit anderen Prozessen
  - Beispiel: Ausgabe des ersten Prozesses wird an den zweiten Prozess weitergereicht und dann zum nächsten, usw.
  - Drei Problemkreise:
    1. Wie werden Informationen von einem Prozess an einen anderen (strukturiert) weitergeleitet (2 Adressräume)
    2. Vermeidung des gleichzeitigen Zugriffs auf gemeinsame Ressourcen
    3. Sauberer Ablauf, wenn Abhängigkeiten vorliegen (Datenausgabe und Weiterverarbeitung)

  - Zwei der obigen Themen gelten auch für Threads
    - Weiterreichen von Infos ist **kein Problem**, da Threads den gleichen Adressraum benutzen (1.)
    - Sich in die Quere zu kommen (2.) und einen sauberen Ablauf zu gewährleisten (3.) gelten auch für Threads
- Strukturierte Art und Weise der Kommunikation notwendig
  → Interprozess--Kommunikation (z.B. P-ID)

→ Im Folgenden werden nur noch Prozesse betrachtet!!

Race Conditions – Beispiel Druckerspooler

- Prozess trägt den Dateinamen der Datei, die er drucken möchte, in Druckerspooler (Spoolerordner) ein.
- Ein anderer Prozess (Druckerdaemon) prüft zyklisch, ob irgendwelche Dateien zu drucken sind.
- Gibt es eine Datei, druckt er diese aus und entfernt den Namen
- Druckerspooler (Spoolerordner) enthält große Anzahl von Einträgen (1,2,3,4,5…) mit Dateinamen



- Es gibt 2 gemeinsame Variablen (unter allen Prozessen verfügbar)
  - *out*, zeigt auf die nächste zu druckende Datei
  - *in*, zeigt auf den nächsten freien Eintrag im Ordner

- Prozess A und B entscheiden fast gleichzeitig, dass sie eine weitere Datei in den Druckerspooler schreiben wollen
- Prozess A liest *in*=7 aus und speichert 7 in seiner lokalen Variablen *next_free_slot*
- Zu diesem Zeitpunkt gibts einen Interrupt und es kommt zu einem Wechsel zu Prozess B

- Prozess B liest *in*=7 aus und speichert 7 in seiner lokalen Variablen *next_free_slot*
- Prozess B läuft weiter und speichert den Namen der Datei im Eintrag 7; Prozess B endet
- Prozess A läuft wieder an, schaut in *next_free_slot*, findet 7, löscht den Eintrag von Prozess B und schreibt seinen Dateinamen im Eintrag 7, setzt danach *in* auf 8
- Der Druckerspooler ist in sich konsistent, aber Prozess B wird nie einen Ausdruck bekommen.

Interprozess-Kommunikation

Race Conditions

- Prozesse kommunizieren ständig mit anderen Prozessen
  - Prozesse, die miteinander arbeiten, benutzen gemeinsamen Speicher, den jeder beschreiben und lesen kann z.B. Festplatte, Arbeitsspeicher, Drucker, …
  - Endergebnis hängt davon ab, welcher Prozess wann genau läuft

→ Race Conditions sind **Unbedingt** zu vermeiden!

→ **Problem: Sie sind sehr schwer zu finden, da die Umgebung nicht reproduziert werden kann**

Race Conditions

- Vermeidung von Race Conditions
  - Zu einem Zeitpunkt darf nur jeweils einem Prozess der Zugriff erlaubt werden (z.B. zum Lesen oder Beschreiben)
  - Es muss sichergestellt werden, daß kein anderer Prozess auf einen gemeinsam genutzten Speicher oder Dateien zugreift oder andere kritische Operationen ausführt.
  - Es muss primitive Operationen geben, um **wechselseitigen Ausschluss** zu erzielen (Synchronisation ist notwendig)
    - Sperrsynchronisation (Ausführung in beliebiger Reihenfolge)
    - Reihenfolgesynchronisation (in fester Reihenfolge), z.B. Datei erzeugen und dann lesen

Wechselseitiger Ausschluss

- Prozesse greifen manchmal auf gemeinsam genutzten Speicher oder Dateien zu oder führen andere kritische Operationen durch.
- **Kritische Regionen/Abschnitte sind Teile des Programms**, in denen auf gemeinsam genutzte Ressourcen (kritische Ressourcen) zugegriffen wird.

- **4 Bedingungen für wechselseitigen Ausschluss**
  - Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein
  - Keine Annahmen über Geschwindigkeit und Anzahl der CPU
  - Kein Prozess, der außerhalb der kritischen Regionen läuft, darf andere Prozesse blockieren
  - Kein Prozess soll ewig warten, um in seine kritische Region zu gelangen

Wechselseitiger Ausschluss (Erklärung)

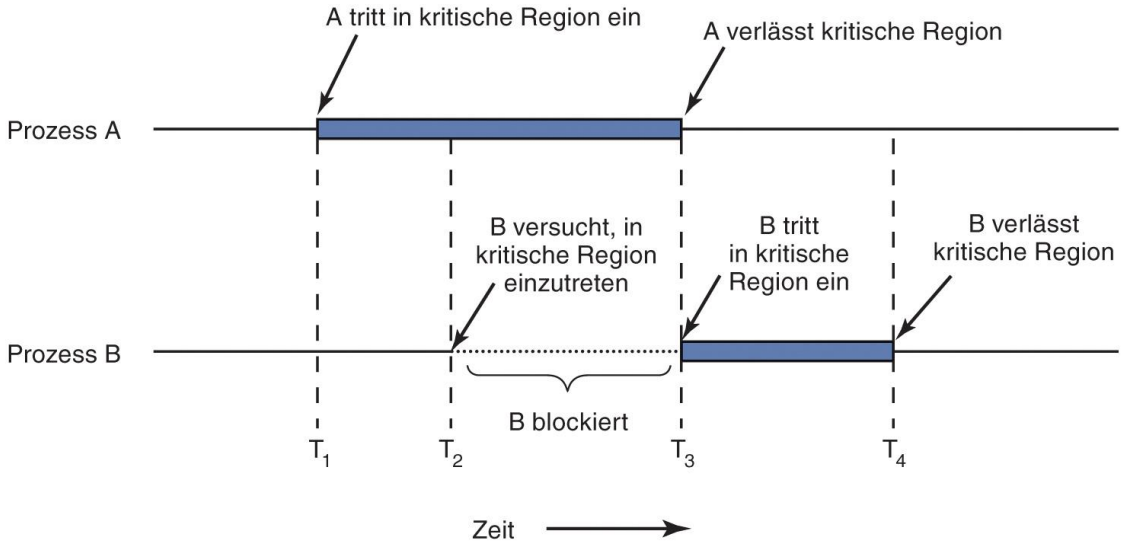


Abbildung 2.22: Wechselseitiger Ausschluss unter Verwendung von kritischen Regionen

- A tritt in kritische Region zum Zeitpunkt T1
- Zum Zeitpunkt T2 versucht B in seine kritische Region einzutreten.
- B scheitert, da A schon in seiner kritischen Region ist
- B wird schlafen gelegt, bis zum Zeitpunkt T3, an dem A seine kritische Region verläßt.
- B tritt sofort in seine kritische Region ein
- B verläßt seine kritische Region zum Zeitpunkt T4

Prozesse

Es gibt folgende **Verfahren** einen wechselseitigen Ausschluss zu realisieren:

- Interrupts ausschalten
- Sperrvariable
- Strikter Wechsel
- Lösung von Peterson

- Wechselseitiger Ausschluss
  - **Ansatz 1: Interrupts ausschalten**
    - Prozesse schalten alle Interrupts aus, sofort nachdem sie in die kritische region eingetreten sind (auch kein Timerinterrupt möglich)
    - CPU wird nicht von einem Taktgeber oder anderen Unterbrechungen weitergeschaltet; nur bei freiwilliger Abgabe der CPU
    - Kein anderer Prozess kommt dazwischen
    - begin_region(): Sperren des Interrupts
    - end_region(): Freigabe der Interrupts

Interprozess-Kommunikation

- Wechselseitiger Ausschluss
  - **Ansatz 1: Interrupt ausschalten**
  - Probleme
    - funktioniert nur bei Ein-Prozessor-Rechnern
    - E/A ist blockiert
    - wenn die Interrupts nicht mehr eingeschaltet werden, würde der Prozess, der die Interrupts ausgeschaltet hat, im Benutzerraum die Kontrolle über das Betriebssystem übernehmen
    - Das Betriebssystem selbst nutzt den Ansatz des Ausschaltens der Interrupts um spezielle Aktionendurchzuführen; aber nur im Kern-Modus.

  → Der wechselseitige Ausschluss durch Abschalten von Interrupts wird immer geringer (Grund Mehrkernsysteme).

  → Bei einem Multiprozessorsystem betrifft die Abschaltung der Interrupts nur diejenige CPU, die den disable interrupt-Befehl ausgeführt hat.

- Wechselseitiger Ausschluss
  - **Ansatz 2: Sperrvariablen**
    - Verwendung einer einzelnen gemeinsam genutzten (Sperr)-Variablen, die mit 0 initialisiert ist
    - Prozess fragt vor Eintritt in kritische Region die Sperre ab
    - Wenn Sperre 0 ist, dann setzt der Prozess sie auf 1 und betritt die kritische Region; ist Sperre 1 wartet der Prozess bis die Sperre 0 wird
    - 0 = kein Prozess in der kritischen Region; 1 = ein Prozess in kritischer Region

Prozess 0

```
{
  while(belegt);
  belegt = true;
  // kritischer abschnitt
  belegt = false;
}
```

begin_region()

end_region()

Prozess 1

```
{
  while(belegt);
  belegt = true;
  // kritischer abschnitt
  belegt = false;
}
```

Interprozess-Kommunikation

- Wechselseitiger Ausschluss
  - **Ansatz 2: Sperrvariablen**
  - Problem: Sperrvariablen verhindern race condition nicht sicher
    - Beispiel:
      - 1. und 2. Prozess lesen gleichzeitig Sperre 0
      - Sie setzen beide die Sperre auf 1
      - Zwei Prozesse betreten gleichzeitig kritische Region

Interprozess-Kommunikation

- Wechselseitiger Ausschluss
  - **Ansatz 3: Strikter Wechsel**
    - Variable turn gibt an, wer an der Reihe ist; turn ist mit 0 initialisiert
    - 1.Prozess stellt fest, daß turn 0 ist, betritt die kritische Region
    - 2.Prozess stellt auch fest, daß turn 0 ist und sitzt deshalb in einer Schleife fest, die laufend turn untersucht (busy waiting)
    - Verläßt 1.Prozess die kritische Region, setzt er turn auf 1.
    - 2.Prozess tritt in kritische Region ein
    - Verläßt 2.Prozess die kritische Region, setzt er turn auf 0, usw.

Prozess 0

```
{
 while(turn != 0);

 // kritischer abschnitt
 turn = 1;
}
```

begin_region()

end_region()

Prozess 1

```
{
 while(turn != 1);

 // kritischer abschnitt
 turn = 0;
}
```

Interprozess-Kommunikation

- Wechselseitiger Ausschluss
  - **Ansatz 3: Strikter Wechsel**
  - Problem: Prozesse sind unterschiedlich langsam
    - Variable turn gibt an, wer an der Reihe ist
    - 1.Prozess stellt fest, daß turn 0 ist, betritt die kritische Region
    - 2.Prozess stellt auch fest, daß turn 0 ist und sitzt deshalb in einer Schleife fest, die laufend turn untersucht (busy waiting)
    - Verläßt 1.Prozess die kritische Region, setzt er turn auf 1.
    - 2.Prozess tritt in kritische Region ein
    - Annahme: 2.Prozess beendet seine kritische Region sehr schnell; beide Prozesse befinden sich nicht in ihren kritischen Regionen, turn=0
    - 1.Prozess führt seine gesamte Schleife schnell aus, verläßt seine kritische Region und setzt turn auf 1; beide Prozesse sind nicht im kritischen Bereich
    - 1.Prozess beendet seinen nicht kritischen Bereich
    - 1.Prozess kann nicht in kritischen Bereich eintreten, da turn = 1 und 2.Prozess mit seiner nicht kritischen Region beschäftigt ist (langsam)
    - Prozess 1 hängt in Schleife fest, bis 2.Prozess turn auf 0 setzt.
  - → Verletzung der Bedingung 3: Kein Prozess, der außerhalb der kritischen Region läuft, darf andere Prozesse blockieren
  - → Prozesse müssen abwechselnd in kritischen Abschnitt eintreten

- Wechselseitiger Ausschluss
  - **Ansatz 4: Lösung von Peterson**
    - Verbindung von Sich-Abwechseln und Sperrvariable (T. Dekker, G.L. Peterson)
    - Es ist kein Prozess in seiner kritischen Region
    - 1.Prozess ruft *enter_region* auf und bekundet sein Interesse, indem er sein Feldelement auf *true* setzt und die Variable turn auf 0.
    - 2.Prozess ist nicht interessiert, *enter_region* kehrt sofort zurück
    - Falls 2.Prozess enter_region aufruft, wird er solange warten, bis interested(0) auf False wechselt, was nur eintritt wenn 1.Prozess leave_region aufruft, um die kritische Region zu verlassen.

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

- Wechselseitiger Ausschluss
  - **Ansatz 4: Lösung von Peterson**
  - Problem: Abfragen und Ändern sind 2 Schritte
    - Die Lösung ist eine Atomare ReadModifyWrite Operation der CPU
    - Heutige Prozessoren arbeiten mit dem Befehl Test and Set Lock (TSL)
      - Beispiel TSL RX, Lock
        - Inhalt des Speicherworts lock wird ins Register RX eingelesen und ein Wert ungleich 0 an der Speicheradresse lock abgelegt.
      - Das Lesen und Schreiben dieses Worts sind unteilbare Operationen
      - kein anderer Prozessor kann auf das gemeinsame Speicherwort zugreifen
      - Bei der Ausführung wird der Speicherbus gesperrt;

Interprozess-Kommunikation – Beispiel 1

Erzeuger-Verbraucher-Problem

- ein weiteres klassisches Problem der Interprozess-kommunikation
- Reales Beispiel: Getränkeautomat
  - Erzeuger: Kantinenpersonal füllt regelmäßig den Automaten auf
  - Verbraucher: Studenten und Professoren holen sich Getränke
  - Der Automat kann eine bestimmte Anzahl Flaschen aufnehmen

Interprozess-Kommunikation – Beispiel 1

Erzeuger-Verbraucher-Problem

- Verallgemeinerung
  - Erzeuger legen Dinge in Puffer
  - Verbraucher nehmen Dinge aus dem Puffer

  - Synchronisation zwischen Erzeuger und Verbraucher notwendig
    - Erzeuger wartet wenn der Puffer voll ist
    - Consumer wartet wenn der Puffer leer ist

Interprozess-Kommunikation

Erzeuger-Verbraucher-Problem

- Verallgemeinerung
  - Kommunikation zwischen zwei Prozessen
    - Erzeuger: legt Informationen in gemeinsamen Puffer ab
    - Verbraucher: liest Informationen aus gemeinsamen Puffer
  - Vermeidung von „busy wait“ durch blockierende Systemaufrufe
    - Sleep(): veranlasst den Aufrufer zu blockieren
    - Wakeup(pid): hebt die Blockierung von Prozess pid auf
  - Erzeuger:
    - wird blockiert, falls Puffer voll ist (Geht schlafen)
    - Wird vom Verbraucher geweckt, wenn der Puffer leer ist
  - Verbraucher:
    - Wird blockiert, wenn der Puffer leer ist
    - Wird vom Erzeuger geweckt, wenn der Puffer voll ist

Erzeuger-Verbraucher-Problem

- Reales Beispiel
  - Mensa ist immer überfüllt
    - Lösung:
      - Es gibt so viele Teller wie Sitzplätze
      - Jeder Student nimmt sich einen Teller vom Stapel am Eingang
      - Beim Verlassen der Mensa spült der Student den Teller und legt ihn auf den Stapel zurück
      - Ist kein Teller auf dem Stapel mehr verfügbar, muss der Student warten, bis wieder ein Teller verfügbar ist
    - Auswirkung: Keine überfüllte Mensa mehr!
    - Aber: aktives Warten der Studenten am Eingang!
      → Es gäbe kein aktives Warten, wenn sleep() und wakeup() benutzt würde (Verfahren von Dijkstra (1965): Semaphor)

Semaphor

- Allgemeines Synchronisationskonstrukt
  - für wechselseitigen Ausschluss und für Reihenfolgesynchronisation
- Semaphor = ganzzahlige Variable (Anzahl der Weckrufe)
  - 0 keine Weckrufe
  - >0 ein oder mehrere Weckrufe

- 2 atomare Funktionen notwendig
  - down() (Verallgemeinerung von sleep())
    - Falls Semaphor >0
      - Verringere Semaphor um 1
  - up() (Verallgemeinerung von wakeup())
    - Erhöhe Semaphor um 1
    - Falls Semaphor <=0 einen blockierten Prozess wecken

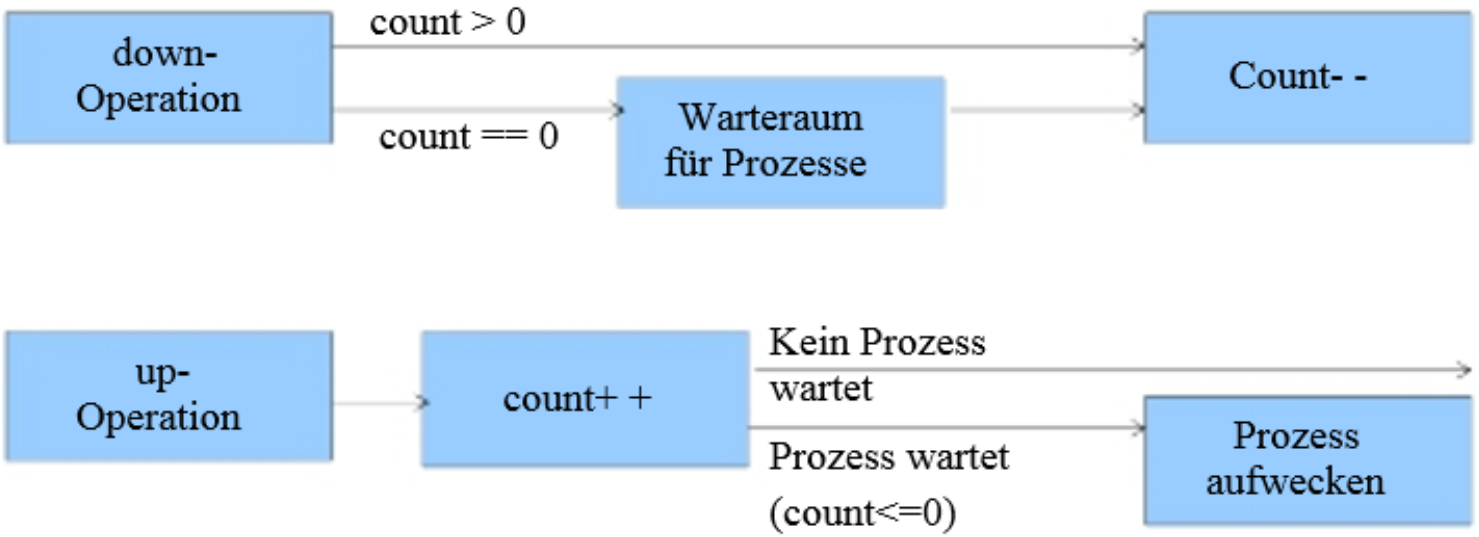
Semaphor

- Semaphor kommt aus der Eisenbahn

Semaphor
- Operationen

down()
- Falls Semaphor >0 → Verringere Semaphor

up()
- Erhöhe Semaphor um 1
- falls Semaphor <=0 → wecken eines blockierten Prozess



- Funktionsnamen unterscheiden sich in der Literatur:
  - down(): P() (proberen), aquire(), (sleep())
  - up(): V() (verhogen), release(), (wakeup())

Interprozess-Kommunikation

Semaphor

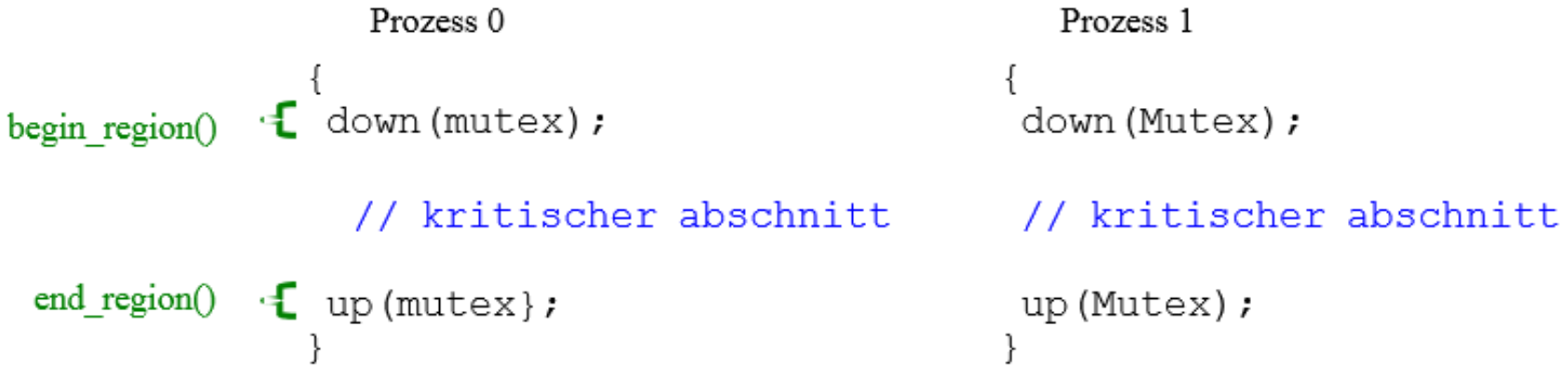
- Bedeutung des Zählers?
  - Zähler >= 0
    - Gibt Anzahl freier Ressourcen an
    - z.B. Anzahl der Flaschen im Getränkeautomaten
  - Zähler < 0
    - Anzahl der wartenden Prozesse
    - z.B. Anzahl der durstigen Studenten und Professoren

- Anwendung der Semaphoren
  - Bei Wechselseitigem Auschluss
  - Bei Reihenfolgesynchronisation (Erzeuger-Verbraucher)

Interprozess-Kommunikation

Semaphor

- Mutex - Anwendung auf wechselseitigen Ausschluss
  - Kritischer Abschnitt
    - Gemeinsame Ressource oder Codestück
  - Semaphor „Mutex“ setzt den Zähler auf 1
    - Binärer Semaphor



Semaphor

- Anwendung auf Erzeuger-Verbraucher-Problem
  - Anforderungen
    1. Nur ein Prozess darf auf Puffer zugreifen
    2. Aus dem leeren Puffer darf nichts entfernt werden
       - ➢ Verbraucher muss warten
    3. In den vollen Puffer darf nichts eingefügt werden
       - ➢ Erzeuger muss warten

  - Lösung:

1. Wechselseitiger Ausschluss (Mutex) — *Wechselseitiger Ausschluss*
2. Semaphore full — *Reihenfolgesynchronisation*
   » wird mit 0 initialisiert
3. Semaphore empty — *Reihenfolgesynchronisation*
   » Wird mit N=Puffergröße initialisiert

Semaphor

- Anwendung auf Erzeuger-Verbraucher-Problem

Semaphore

```
Semaphor  mutex = 1; // für wechselseitigen Ausschluss
Semaphor  full  = 0; // verhindert Entfernen aus leerem Puffer
Semaphor  empty = N; // verhindert Schreiben in vollen Puffer
```

Producer

```
void producer()
{
 while(TRUE) {
  item = produce_item();
  down(&empty);
  down(&mutex);
  insert_item(item);// k.A
  up(&mutex);
  up(&full);

 }
}
```

begin_region()

end_region()

Consumer

```
void consumer()
{
 while(TRUE) {

 down(&full);
 down(&mutex);
 item=remove_item(); // k.A.
 up(&mutex);
 up(&empty);
 consume_item(item);
 }
}
```

Semaphor

- Anwendung auf Erzeuger-Verbraucher-Problem

Semaphore

```
Semaphor  mutex = 1; // für wechselseitigen Ausschluss
Semaphor  full  = 0; // verhindert Entfernen aus leerem Puffer
Semaphor  empty = N; // verhindert Schreiben in vollen Puffer
```

- full bzw. empty sind Scheduling-Randbedingungen
  - Erzeuger: warte, wenn Puffer voll
  - Verbraucher: warte, wenn Puffer leer
- Daumenregel:
  - Eine Semaphore für jede Randbedingung (Constraint)

Semaphor

- Wenn zwei Prozesse abwechselnd aufgerufen werden, benutzt man zwei Semaphoren
  - Prozess 1 kommt nach Prozess 2
  - Prozess 2 kommt nach Prozess 1

```
Semaphor  full  = 0;
Semaphor  empty = 1;
```

```
while(TRUE)  {
   down(&empty);
   print(„1“);
   up(&full);
 }
```

```
while(TRUE)  {
   down(&full);
   print(„2“);
   up(&empty);
}
```

Monitor

- Programmierung der Semaphoren ist schwierig
  - Reihenfolge der down/up-Operationen ist wichtig
    - Falsche Reihenfolge kann zu Deadlock Situationen führen
  - Synchronisation über das gesamte Programm verteilt
- Monitore
  - Basisoperation zur Synchronisation (Brinch Hansen (1973), Hoare (1974))
  - Sammlung von Prozeduren, Variablen und Datenstrukturen, die alle in einer Art von Modul oder Paket zusammengefasst werden
  - Prozesse können jederzeit die Prozeduren in einem Monitor aufrufen
  - Prozesse haben aus Prozeduren, die ausserhalb des Monitors definiert sind, keinen direkten Zugriff auf die internen Variablen und Datenstrukturen des Monitors
  - Zugriff auf die Daten nur über Monitor-Prozeduren
  - Nur jeweils ein Prozess kann den Monitor benutzen
  - Nur eine Prozedur des Monitors kann ausgeführt werden!
  - Spezielles Programmiersprachenkonstrukt, das der Compiler erkennt

Monitor

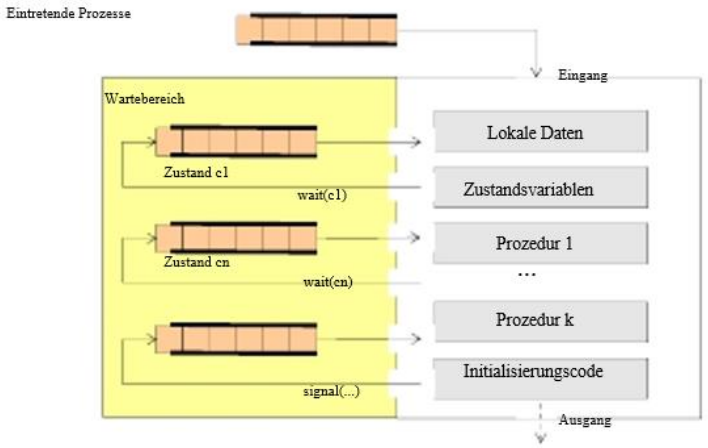
- Realisierung
  - Compiler realisiert den wechselseitigen Ausschluss von Monitoreintritten
  - Person, die den Monitor programmiert, muss nicht wissen, wie der Compiler den wechselseitigen Ausschluss realisiert

  - Für wechselseitigen Ausschluss mit Mutex

  - Für die Reihenfolgesynchronisation zwischen Monitorprozeduren
    - Einführung Zustandsvariable (condition variable)
    - Zwei Operationen wait() und signal()

Monitor

- Realisierung (Reihenfolgesynchronisation)
  - Zustandsvariable mit Zwei Operationen
    - **wait():** Blockieren des aufrufenden Prozesses
      - Aufrufender Prozess wird blockiert
      - Aufrufender Prozess wird in die Warteschlange der Zustandsvariablen eingetragen
      - Monitor steht bis zum Ende der Blockierung anderen Prozessen zur Verfügung
    - **signal():** Aufwecken blockierter Prozesse
      - Falls Warteschlange der Zustands-variablen nicht leer
      - Mindestens einen Prozess wecken
        - Aus Warteschlange entfernen
        - Blockierung aufheben



Monitor

- Realisierung für Erzeuger-Verbraucher

```
monitor ProducerConsumer
 condition full, empty
 integer count
 procedure insert(item)
 begin
  if count = N then wait(full);
  insert_item(item)
  count = count +1 ;
  if count=1 then signal(empty);
 end
 procedure remove()
 begin
  if count = 0 then wait(empty);
  remove_item(item)
  count = count - 1 ;
  if count=N-1 then signal(full);
 end
 count=0;
end monitor;
```

```
Procedure producer
Begin
 While true do
 Begin
   Item = produce_item;
   ProducerConsumer.insert(item);
 end;
end;
```

```
Procedure consumer
Begin
  While true do
  Begin
   item=ProducerConsumer.remove();
   consume_item(item);
  end;
End;
```

Monitor

- Eigenschaften von Monitoren
  - Es kann immer nur ein Prozess im Monitor aktiv sein!
    - Falls schon ein Prozess im Monitor ist, wird der aufrufende Monitor stillgelegt!
    - Zentrale Idee: Prozess kann auch innerhalb der kritischen Region schlafen gehen, da der Lock mit dem Schlafengehen frei gegeben wird
      - ➢ Unterschied zu Semaphor: kann nicht in kritischer Region warten!
    - Monitor-Realisierung braucht
      - ➢ ein **Mutex** für gegenseitigen Ausschluss
      - ➢ **Zustandsvariablen,** für Scheduling-Randbedingungen
        - › Prozess kann nicht mehr weitermachen, weil Puffer voll ist!

Monitor

- Beispiel Java
  - Threads werden im Benutzermodus unterstützt
  - Mit „synchronized“ wird garantiert, dass nur eine Methode des Objekts ausgeführt wird!
  - Anwenden auf Erzeuger-Verbraucher-Problem
    - 4 Klassen
      - Producer
      - Consumer
      - Monitor
        - Implementiert den Monitor
        - Verwendet synchronized für insert() und remove()
      - ProducerConsumer
        - Erzeugt die Threads und startet diese!

Monitor

- Für wechselseitigen Ausschluss in der parallelen Programmierung sind Monitore weniger fehleranfällig als Semaphore (+)

- Monitore sind ein Pattern
  - sehr oft als Konzept innerhalb einer Programmiersprache
    - Compiler muss sie erkennen und damit umgehen
    - C kann das nicht
      - kann Semaphore über atomare Assemblerroutine in den Bibliotheken leicht realisieren

- Monitore und Semaphore eignen sich nicht für verteilte Systeme
  - mit getrenntem privaten Speicher nicht realisierbar
  - Semaphore sind sehr maschinennah!
  - Monitore nur in wenigen Programmiersprachen anwendbar!

→ **Austausch von Nachrichten**

Nachrichtenaustausch (message passing)

- Bisher: Speicherbasierte Kommunikation
  - Über gemeinsamen Speicher
    - (s. Erzeuger/Verbraucher-Problem)
    - i.d.R. zwischen Threads
  - Synchronisation muss explizit programmiert werden

- Neu: Nachrichtenbasierte Kommunikation
  - Senden/Empfangen von Nachrichten (über das BS)
    - ➢ i.d.R. zwischen Prozessen
  - auch über Rechnergrenzen hinweg möglich
  - Implizite Synchronisation

Nachrichtenaustausch

- Methode der Interprozesskommunikation, die 2 Befehle benutzt
  - **send**(Ziel, Nachricht)
    - Versenden einer Nachricht
  - **receive** (Quelle, Nachricht)
    - Empfang einer Nachricht
      - Wenn keine Nachricht vorhanden, blockiert der Empfänger so lange
  - Besonderheiten:
    - Bestätigungen (Acknowledgements), um sich gegen den Verlust von Nachrichten abzusichern
    - Authentifizierung in verteilten Systemen
  - Auf lokalen Ressourcen ist Nachrichtenaustausch immer als Semaphor-Operation realisiert

Nachrichtenaustausch

- Erzeuger-Verbraucher-Problem
  - Ansatz:
    - Nachrichten haben die gleiche Größe
    - Gesendete und nicht empfangene Nachrichten werden vom BS zwischengespeichert
    - Zu Beginn schickt der Consumer N leere Nachrichten an den Producer
    - Jedes Mal wenn Erzeuger eine Nachricht an den Verbraucher geben kann, nimmt er eine leere und schickt eine Gefüllte zurück
    - Gesamtzahl der Nachrichten bleibt gleich
    - Falls der Erzeuger schneller arbeitet als der Consumer, sind alle Nachrichten gefüllt und warten auf den Verbraucher; Erzeuger wird blockiert und wartet auf eine leere Nachricht
    - Falls der Verbraucher schneller arbeitet, passiert das Gegenteil

Producer

```
void producer(){
 int item;
 message m;
 while(true){
  item=produce_item();
  receive(consumer, &m);
  build_message(&m, item);
  send(consumer, &m);

 }
}
```

Consumer

```
void consumer(){
 int item;
 message m;
 while(true){
  receive(producer, &m);
  item=extract_message(&m);
  send(producer, &m);
  consume_item(&item);

 }
}
```