Angewandte Computer- und Biowissenschaften

HOCHSCHULE MITTWEIDA
University of Applied Sciences

Fraunhofer SIT

Bundeskriminalamt

Betriebssysteme

Betriebssystemarchitektur

Autor: Ronny Bodach, Felix Fischer

Stand: 27.04.2022

Agenda

1. Grundlegender Aufbau von Rechnern
2. Von Neumann Rechnerarchitektur
3. Harvard-Architektur
4. Computerarchitektur nach Tanenbaum
    1. Transistorebene
    2. Logische Ebene
    3. Microarchitektur
    4. Instruction Set Architecture
    5. Betriebssystem
5. Betriebssystem-klassifikation
6. Interrupts
7. **Prozess, Task und Thread**
8. **Scheduling**
9. **Parallelität und Nebenläufigkeit**
10. Speicherverwaltung

Prozess, Task und Thread

Teil 1

Wiederholung Prozessor

- Register
  - ein Prozessor besitzt Steuer- und Mehrzweckregister
  - das Steuerregister enthält:
    - Programmzähler (Instruction Pointer)
    - Stapelregister (Stack Pointer)
    - Statusregister
    - usw.
- Programmzähler enthält Speicherstelle der nächsten Instruktion
  - die Instruktion wird geladen und
  - sie wird ausgeführt
  - der Programmzähler wird inkrementiert
  - dieser Vorgang wird ständig wiederholt

Wiederholung Prozessor Modi

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

Vom Programm zum Prozess

Ein Prozess ist ein in Ausführung befindliches (nicht zwingend aktives) Programm (Maschinenprogramm, Anwendung).

Das heißt: Während ein Maschinenprogramm nur eine Folge von Befehlen ist, wird dieses Programm zum Prozess, indem es in den Speicher geladen wird, einen eigenen Prozess-Kontext erhält und gestartet wird (Der Programmzähler wird auf die Anfangsadresse des Maschinenprogramms gesetzt).

Insbesondere beinhaltet der Prozess:

- den aktuellen Wert des Programmzählers
- aktuelle Werte der Register und der Variablen

Prozess

- ein Programm, das sich in Ausführung befindet, und seine aktuell genutzten Daten
- ein Programm kann sich mehrfach in Ausführung befinden, dann gibt es auch mehrere Prozesse!
- eine konkrete Ausführungsumgebung für ein Programm, mit den dazu notwendigen Betriebsmittel: Speicher, Rechte, Verwaltungsinformation (verbrauchte Rechenzeit,...),...

Vom Programm zum Prozess

Prozess stellt Ausführungsumgebung bereit wie ein virtueller Prozessor:

- ein virtueller Prozessor, der ein Programm ausführt
- Speicher → virtueller Adressraum
- Prozessor → Zeitanteile am echten Prozessor
- Interrupts → Signale zur Unterbrechung
- I/O-Schnittstellen → Dateisystem, Kommunikationsmechanismen
- Maschinenbefehle → direkte Ausführung durch echten Prozessor

Mehrprozessbetrieb (Multitasking)

- mehrere Prozesse können quasi gleichzeitig ausgeführt werden
- für jeden Prozessor werden Zeitanteile der Rechenzeit an die Prozesse im ***Time Sharing System*** vergeben
- die Entscheidung, der Umschaltung zwischen Prozessen und die Zuteilung von Rechenzeit trifft das Betriebssystem mit dem ***Scheduler***
- die Umschaltung zwischen Prozessen erfolgt durch das Betriebssystem mit Hilfe eines ***Dispatcher***
- Prozesse laufen nebenläufig (das gerade in Ausführung befindliche Programm weiß nicht, wann und wo auf einen anderen Prozess umgeschaltet wird)
- Grundlage für die Arbeit im Multitasking/Multiprocessing sind Prozesse mit eigenständigen Befehlszählern je Prozess

Mehrbenutzerbetrieb (Multi-User)

- mehrere Benutzer führen mehrere Programme in einzelnen Prozessen aus
- ähnlich dem Multitasking-Betrieb
- Aber: die Unterscheidung von Benutzern und evtl. Zugriffsberechtigungen wird durch das Betriebssystem realisiert

Kooperatives Multitasking

Kooperatives Multitasking bezeichnet den Prozesswechsel unter Kontrolle der Prozesse. Der gerade laufende Prozess bestimmt den Zeitpunkt des Prozess-Kontextwechsels.

Nachteile:

- Wartzeiten beeinflussen die Arbeit anderer Prozesse (keine Transparenz)
- Keine Fairness zwischen Prozessen möglich
- Möglichkeit des Hängens beim Warten aus Ressourcen

Dennoch ist kooperatives Multitasking keineswegs überholt oder schlecht. Gerade im Bereich der Mikrocontroller und Echtzeitanwendungen gibt es viele gute Argumente dafür: Kooperatives Multitasking ist deterministischer, also besser zeitlich und logisch vorhersagbar und es ist besser simulierbar.

Beispiele für Betriebssysteme mit kooperativem Multitasking sind Windows 3.x und MacOS vor Version 10

Präemptives Multitasking

Der Prozesswechsel unterliegt dem Betriebssystem

Grundlage für Prozesswechsel:

- Wechsel nach/in Systemaufrufen und Unterbrechungen/Interrupts
- Warten auf Ereignisse (z.B. Zeitpunkt, Nachricht, E/A-Operationen)
  - Beenden von Prozessen
  - Ende der Zeitzuteilung im Time Sharing Betrieb
  - Prozess erhält den Status Abarbeitungsfähig

Prozesskontextwechsel (Process Context Switch)

Eine Prozess Umschaltung erfolgt durch:

1. Sichern der vom Prozess genutzten Register und Befehlszähler
2. Prozessauswahl
3. Prozessumgebung generieren
4. gesicherte Registerinformationen und Befehlszäher laden
5. Befehl des Befehlszähler abarbeiten

Die notwendigen Prozessinformationen erhält das Betriebssystem dabei aus einem Prozesskontrollblock (PCB), welcher dem Prozess im Hauptspeicher vorgelagert ist.

Prozesskontrollblock (Process Control Block)

- Datenstruktur des Betriebssystems, die alle nötigen Daten für einen Prozess hält.
- Der PCB in Windows Umgebungen
  - Da Windows zur Gattung der "Closed-Source"-Software gehört, ist der Prozesskontrollblock unter Windows nicht offiziell veröffentlicht.
  - Russinovich et.al. 2012 (P1) beschreiben, dass jeder unter Windows erzeugter Prozess durch eine Instanz der Datenstruktur EPROCESS (engl.: executive process structure) repräsentiert wird.

Prozesskontrollblock (Process Control Block)

Datenstruktur des Betriebssystems, die alle nötigen Daten für einen Prozess hält.

Der PCB in UNIX Umgebungen enthält:

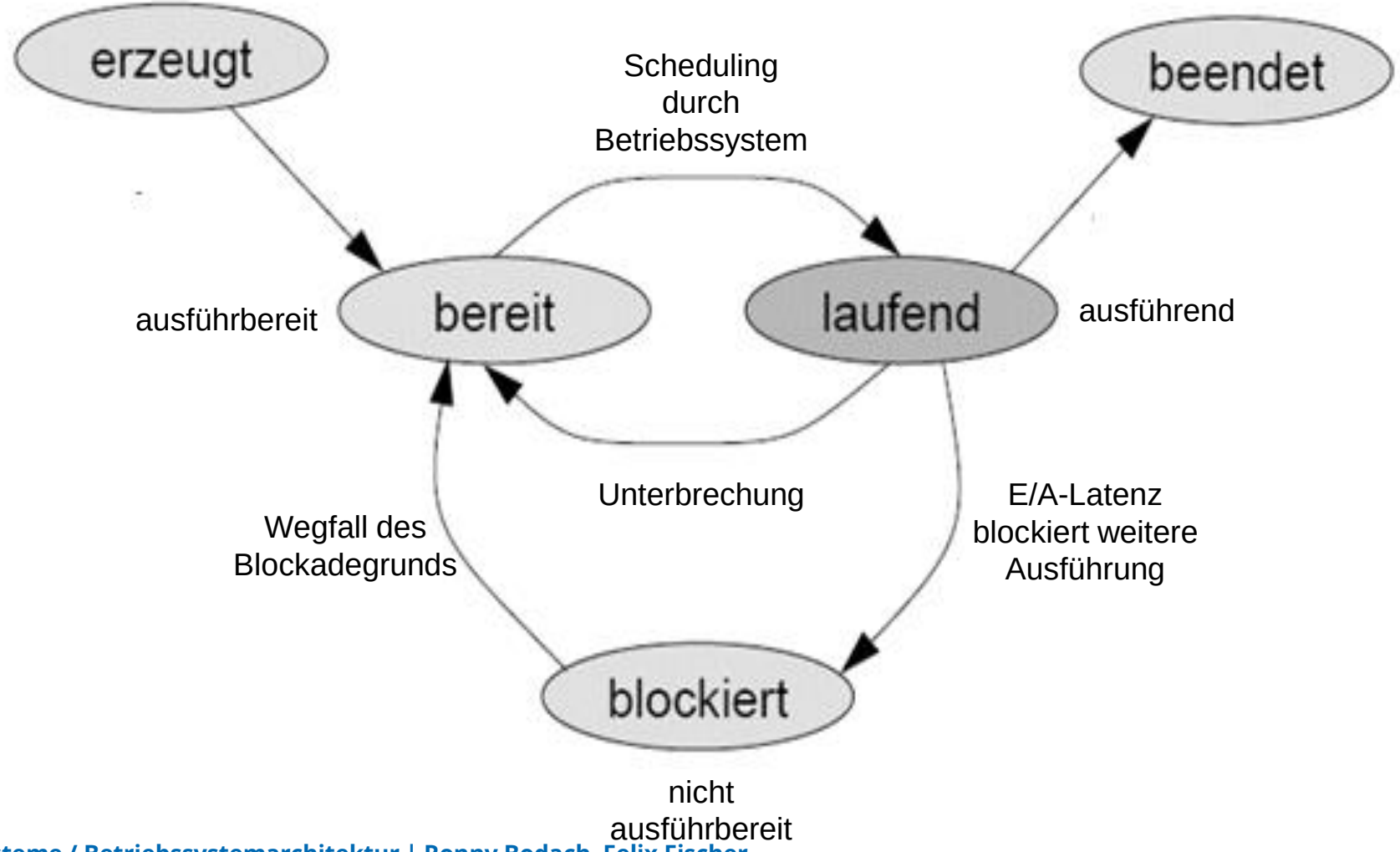
- Prozessnummer (PID)
- verbrauchte Rechenzeit
- Erzeugungszeitpunkt
- Kontext (Register etc.)
- Speicherabbildung
- Eigentümer (UID, GID)
- ...

Prozesszustände

Ein Prozess befindet sich in einem der folgenden Zustände:

- **Erzeugt** *(New)*
  Prozess ist erzeugt, ist noch nicht im Besitz aller 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 Ausgabe-operation, 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

Zustandsdiagramm der Prozesszustände



Scheduling

Scheduling Auswahlstrategien

Kriterien für die Auswahl durch den Scheduler:

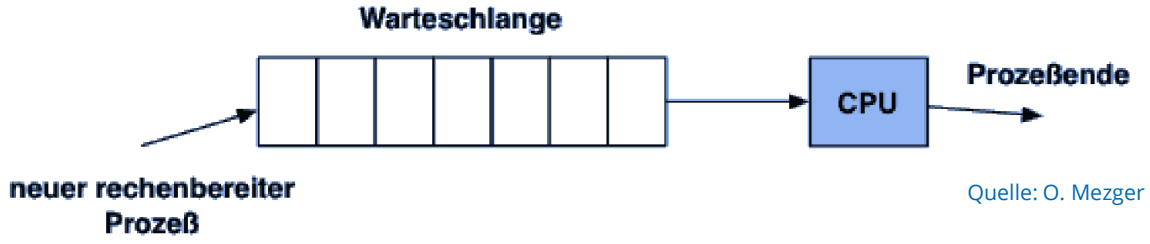
- CPU-Auslastung: gewünscht ist eine 100% ausgelastete CPU
- Durchsatz: pro Zeiteinteilung Optimum an bearbeiteten Prozessen
- Verweilzeit: geringe Gesamtabarbeitungszeit eines Prozesses
- Wartezeit: geringe Verweildauer eines Prozesses im Zustand „bereit“ ohne Abarbeitung
- Antwortzeit: im interaktivem Betrieb schnelle Reaktion auf Eingaben

Scheduling Auswahlstrategien

- First Come First Served
- Shortest Job First
- Prioritäts-Scheduling (non präemptiv)
- Round Robin
- Prioritäts-Scheduling (präemptiv)
- Multilevel-Queue Scheduling
- Multilevel-Feedback-Queue Scheduling

First Come First Served (FCFS)

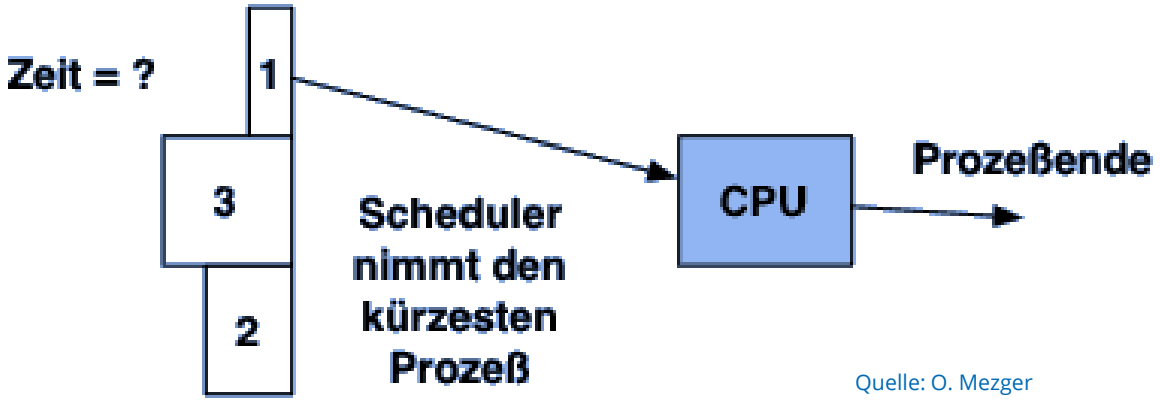
- Auch First In First Out (FIFO)
- Eingangsreihenfolge (Warteschlange)
- Ein Prozess verfügt über den Prozessor, bis er beendet ist, erst dann kann ein neuer Prozess gestartet werden.
- Vorteile:
  - Einfach
- Nachteil:
  - Antwortzeit wächst mit Prozessanzahl



Quelle: O. Mezger

Shortest Job First (SJF)

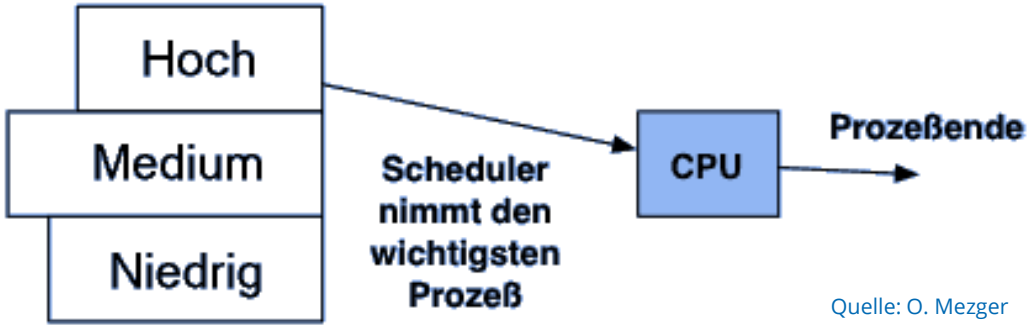
- Ziel: Gesamtwartezeit aller Prozesse minimieren,
- Voraussetzung:
  - Restrechenzeit der Prozesse lässt sich abschätzen



Quelle: O. Mezger

Highest Priority First (nicht präemptiv)

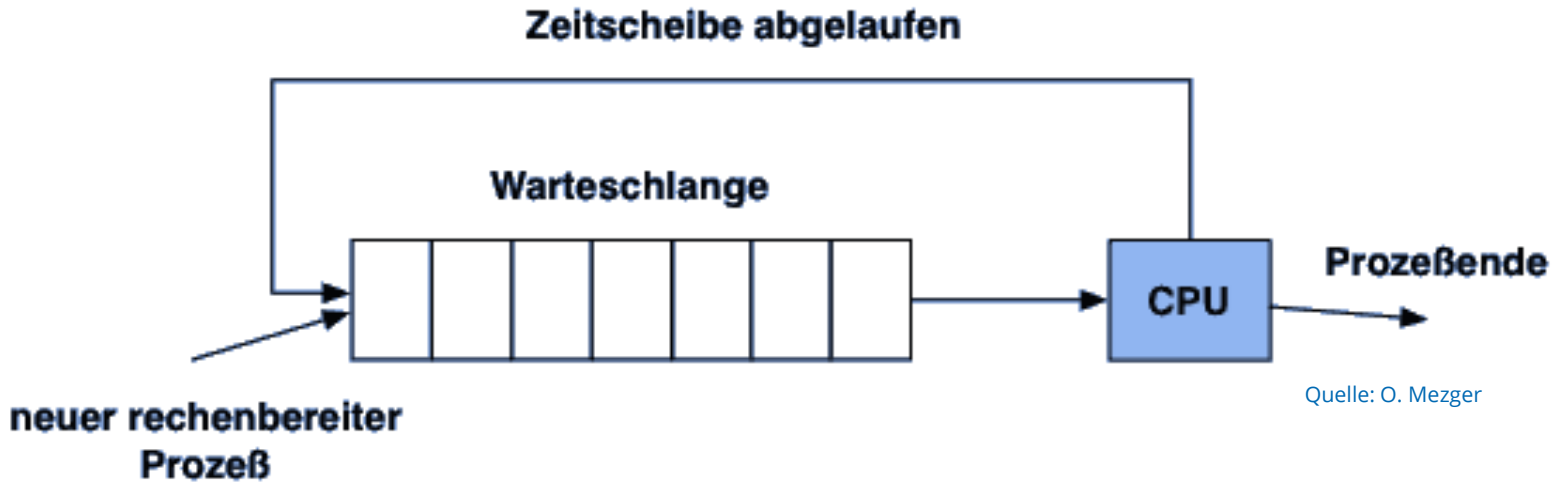
- Nach dem Ende eines Prozesses wird aus den wartenden Prozessen der mit der höchsten Priorität ausgewählt und gestartet.
- Probleme
  - Prioritätsumkehr (Priority Inversion)
  - Aushungerung (Starvation)



Quelle: O. Mezger

Round Robin

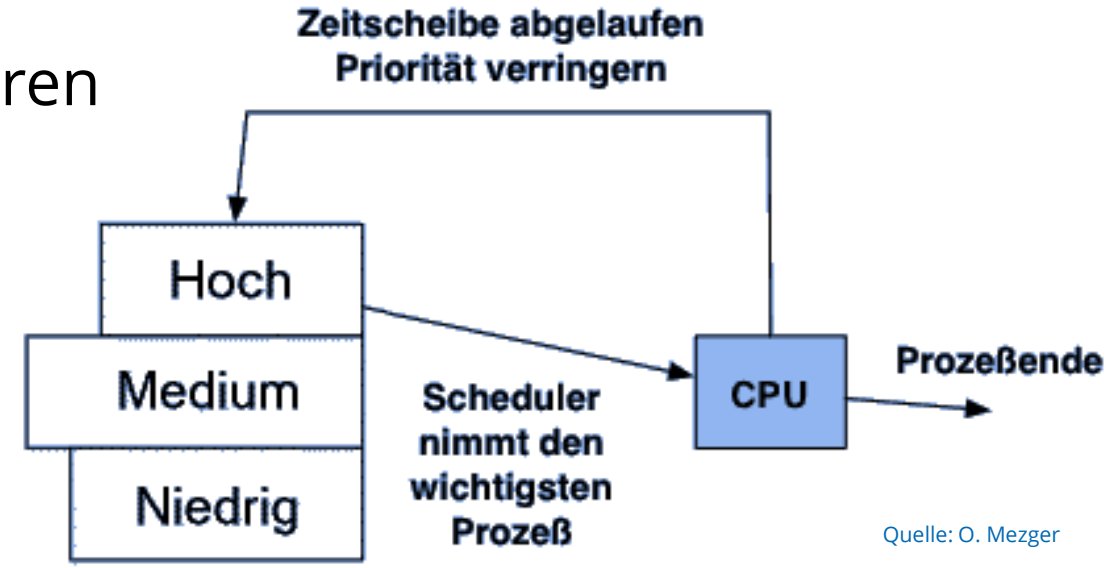
- Zeitscheibenverfahren
- Warteschlange mit erneuten Anstellen nach Zeitablauf
- neue Prozesse hinten angestellt
- Implizite Annahme:
  - alle Prozesse sind gleich wichtig



Quelle: O. Mezger

Prioritäts-Scheduling (präemptiv)

- Prozesse werden nach Priorität ausgewählt
- Nach Ablauf der Zeitscheibe wird die Priorität verringert
- Prozesswechsel wird durch neue Prioritäten veranlasst.
- Bei Fällen gleicher Priorität?
  Round-Robin als Standard-Verfahren
- Hinweis: Nach einer "Epoche" (relativ lange Zeiteinheit bzgl. Zeitscheibe) werden die Prioritäten wieder erhöht. Dies nennt man Aging.



Quelle: O. Mezger

Multilevel-Queue Scheduling

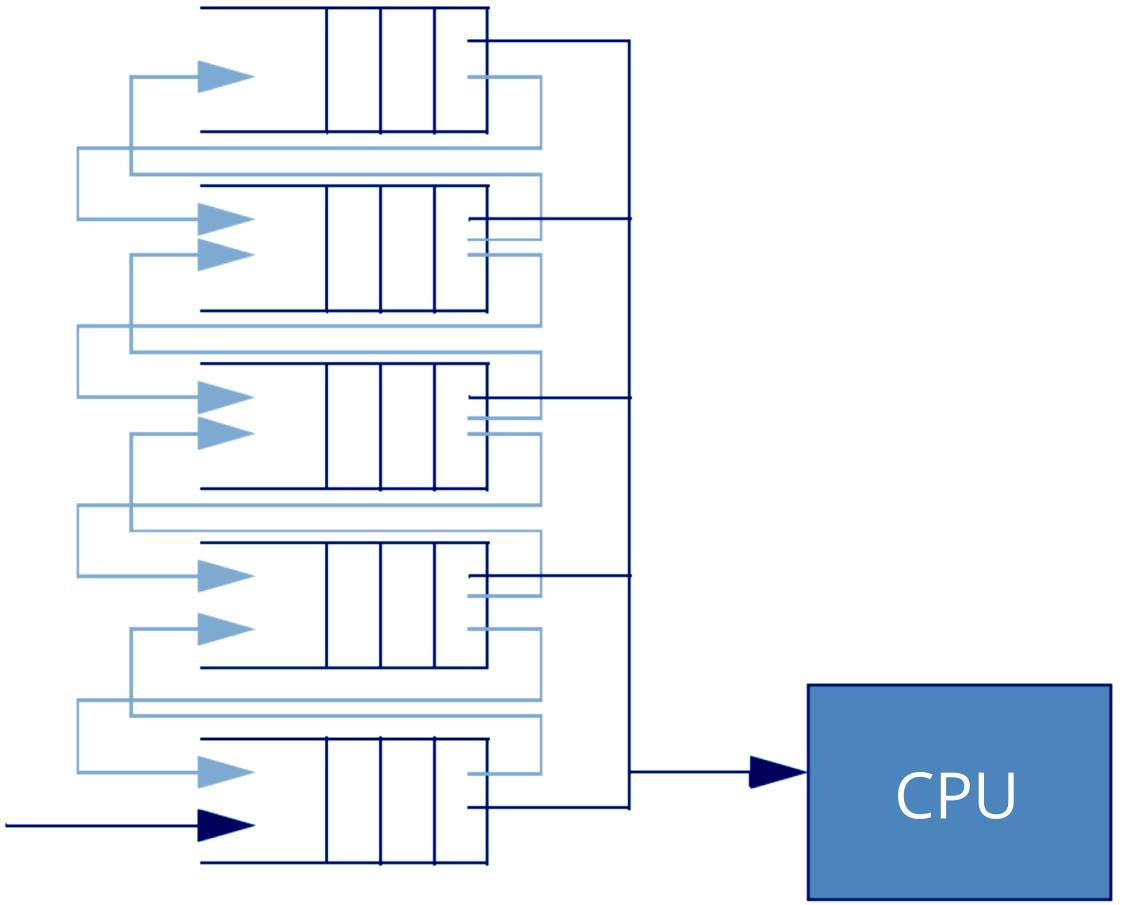
- Scheduling durch verschiedene Scheduling-Klassen
  - Hintergrundprozesse (Batch)
  - Vordergrundprozesse (interaktive Prozesse)
- Zwischen den Klassen gibt es als Scheduling-Strategie feste Prioritäten (Vordergrundprozesse immer vor Hintergrundprozessen)
- Jede Klasse besitzt ihre eigenen Warteschlangen (Queues) und verwaltet diese nach einer eigenen Scheduling-Strategie (Round Robin)

Multilevel-Feedback-Queue Scheduling

- Ähnlich Multilevel-Queue Scheduling
  - Einteilung in Scheduling-Klassen mit gleicher Scheduling Strategie innerhalb der Klassen (z.B. Round Robin)
  - Auswahl der Scheduling-Klassen mittels Prioritäten (z.B. 1. Klasse vor 2. Klasse)
- Zusätzlich bei Multilevel-Feedback-Queue Scheduling (MLFB)
  - Transfer von Prozessen von einer Klasse in die andere (Feedback)

Multilevel-Feedback-Queue Scheduling

- Beispiel
  - fünf Scheduling-Klassen mit eigener Priorität, Queue und Strategie
  - Transfer von Prozessen zwischen den fünf Scheduling-Klassen möglich



Quelle: H. Falk

Prozess, Task und Thread

Teil 2

Multitasking für mehrere Prozesse einer Anwendung

- Die Aufgabenverteilung in modernen Betriebssystemen wird in aller Regel auf mehrere Prozesse verteilt. Diese Prozesse nutzen gemeinsame Betriebsmittel und kooperieren dabei untereinander.
- Die Aufteilung einer Anwendung in Tasks kann ebenfalls mittels mehrerer Prozesse realisiert werden:
  - Bei geeigneter Hardware ist die parallele Ausführung von Prozessen durchführbar. Solche Multiprozessor-Systeme werden etwa in Hochleistungsrechnen (Simulationen, Wettermodelle etc.) genutzt.
  - Ein weiteres Beispiel sind Client-Server Anwendungen unter UNIX/LINUX die pro Verbindung einen eigenen Server-Prozess starten.
  - Auch die Implementation von Echtzeitfähigkeiten in bestehende Betriebssysteme wird über geeignete Prioritätsgesteuerte Prozesse realisiert.

Multitasking für mehrere Prozesse einer Anwendung

- Kommunikation zwischen kooperierenden Prozessen erfolgt über
  - über gemeinsame Speicherbereichnutzung
  - Inter-Prozess-Kommunikation (IPC)

- Aufteilung einer Anwendung in unabhängige Tasks
  - einfache Implementation
  - parallele Abarbeitung von Tasks
  - Priorisierung von Tasks
  - Prozessverwaltungsstrukturen bedingen einen gewissen Overhead
  - Prozesswechsel sind aufwändig und nur in mehreren Schritten realisierbar

Alternativen zu Tasks

- Threads (Aktivitätsträger)
- User-level Threads
- Kernel-level Threads
- Lightweight Processes (LWP)

Threads

- gemeinsame Nutzung von Ressourcen:
  - Instruktionen
  - Datenbereiche
  - E/A Ressourcen
- separate Nutzung von:
  - Programmzählern
  - Registersätzen
  - Stackbereichen

Threads

- Die Umschalten zwischen zwei Threads einer Prozess-Gruppe kann einfacher erfolgen als ein Prozess-Kontextwechsel.
- Programmzähler, Registersätze und Stackbereiche müssen gewechselt werden
- die Datenspeicherinhalte bleiben unberührt
- alle E/A Ressourcen bleiben weiterhin verfügbar

User-Level Threads

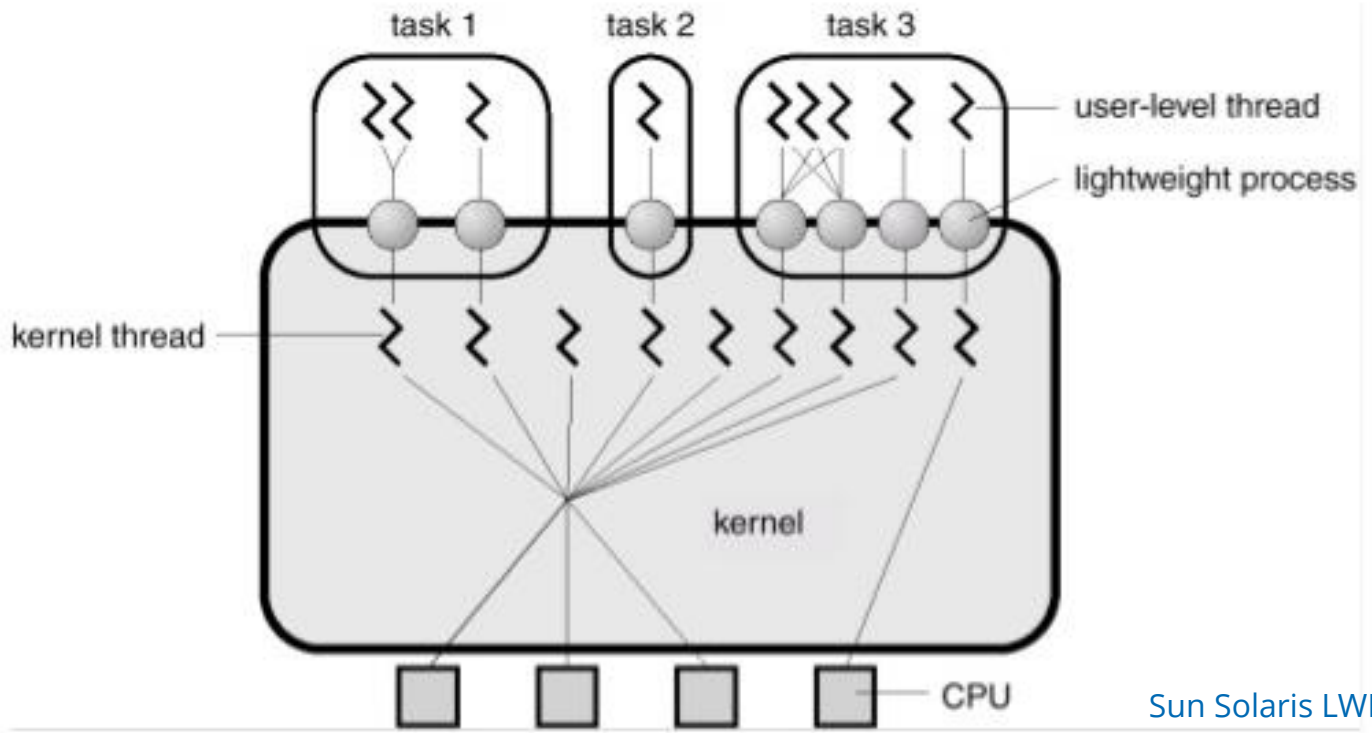
- Implementierung
  - Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
  - Betriebssystem sieht nur einen Single-Thread-Prozess
- Vorteile
  - keine Systemaufrufe zum Umschalten von Threads erforderlich
  - können auf jedem Betriebssystem ausgeführt werden.
  - für Thread-Wechseln sind keine Kernelmodus-Berechtigungen erforderlich.
  - Thread-Wechsel obliegen dem Anwender und dessen Implementation (Scheduling-Strategie)
- Nachteile
  - Multithread-Anwendungen können Multiprocessing nicht nutzen.
  - der gesamte Prozess wird blockiert, wenn ein Thread einen Blockierungsvorgang ausführt

Kernel-Level Threads

- Implementierung
  - Threads auf Kernelebene werden direkt vom Betriebssystem verarbeitet, und die Threadverwaltung erfolgt durch den Kernel.
- Vorteile
  - kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
  - Multiprozessor Hardware wird mit echten parallelen Abläufen wird unterstützt
  - Betriebssystem kann ebenfalls Multithreading unterstützen
- Nachteile
  - Moduswechsel in den Kernelmodus erforderlich für Threadumschaltung
  - Fairnessverhalten nötig (zwischen Prozessen mit vielen und solchen mit wenigen Threads)
  - Scheduling-Strategie meist vom Betriebssystem vorgegeben

Lightweight Processes (LWP)

LWP ist eine Implementation wo das Scheduling und die Ausführung durch den Kernel erfolgt, jedoch die sonstige Implementation denen, der User Level Threads entspricht.



Sun Solaris LWP nach Silberschatz 1999

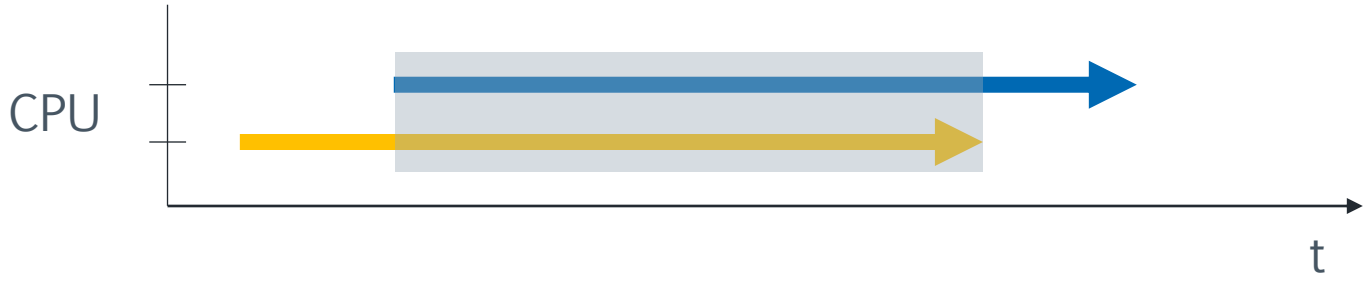
Parallelität und Nebenläufigkeit

Parallelität und Nebenläufigkeit

- Durch die Anwendung folgender Technologien lässt sich die Rechenleistung Signifikant erhöhen und es können echte parallele Aufgaben bzw. quasi-parallele Aufgaben in Nebenläufigkeit ausgeführt werden:
  - durch Multiprocessing (Multitasking)
  - durch Multithreading
  - Spezialfall: Hyperthreading

Parallelität von Prozessen oder Threads

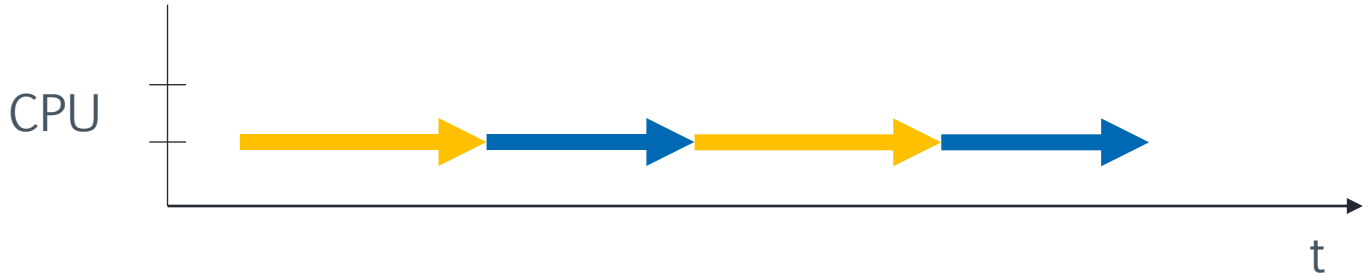
- die Anweisungen zweier Prozesse/Threads werden parallel bearbeitet, wenn die Anweisungen unabhängig voneinander zur gleichen Zeit ausgeführt werden
- Eine echte parallele Bearbeitung erfolgt nur auf geeigneter Hardware wie Multiprozessoren/Mehrkernprozessoren
- keine parallele Abarbeitung von Befehlen auf Singleprozessoren möglich



Quelle: H. Falk

Nebenläufigkeit

- Zwei Prozesse heißen nebenläufig, wenn ihre Anweisungen unabhängig voneinander abgearbeitet werden
- eine parallele Bearbeitung ist immer nebenläufig
- nebenläufige Bearbeitung ist auch auf Singleprozessoren möglich durch eine zeitliche Begrenzung einzelner Prozesse oder andere geeignete Scheduling Techniken, sozusagen quasi-parallel



Quelle: H. Falk

Probleme nebenläufiger Abarbeitung

- die in einer höheren Programmiersprache aufgeführten Befehle werden bei Abarbeitung nicht atomar (unteilbar) abgearbeitet, da die einzelnen Befehle in der Regel in mehrerer Maschinenbefehle aufgeteilt werden
- bei einem Prozesswechsel innerhalb einer Befehlsausführung in der höheren Programmstruktur oder zwischen zwei aufeinanderfolgenden Befehlen können Inkonsistenzen auftreten, die im schlimmsten Fall zu einem Absturz führen

Ausführungsinkonsistenzen

- Grund der Inkonsistenzen ist in so einem Fall, die gemeinsame Nutzung von Daten und Ressourcen
- Dieses Problem kann man mit der Einführung sogenannter kritischer Abschnitte (Critical Sections) beseitigen
  - Man nutz dabei die Methode des wechselseitigen Ausschluss (Mutual Exclusion, Mutex)
  - ein Prozess/Thread erhält Zugang zu Daten/Ressourcen nur im wechselseitigen Einverständnis
  - damit erreicht man das Kritische Abschnitte zeitlich unteilbar (atomar) werden
  - Notwendig dafür sind Implementation die verhindern, dass mehrere Prozesse gleichzeitig im kritischen Abschnitt sind

Kritische Abschnitte

- Damit dienen kritische Abschnitte der Beschränkung und Koordinierung der Nebenläufigkeit durch die Methode des wechselseitigen Ausschluss in dem sie:
  - die zeitliche teilbare Ausführung von Befehlen höherwertiger Programmiersprachen einschränken
  - die Nebenläufigkeit für den Zeitraum der Abarbeitung dieser kritischen Abschnitte außer Kraft setzen
  - die Unabhängigkeit bei der Befehlsausführung unterbinden

Beispiel Inkonsistenzen

Inkonsistenzen durch gleichzeitige Nutzung der Variablen counter:

```
int counter = 0;
```

Main Init

```
counter = counter +1;
```

Prozess1

```
printf counter;
counter = 0;
```

Prozess2

Hierbei kann es passieren das der Zähler noch zählt bevor er auf Null gesetzt wird, da im Prozess 2 zwischen den beiden Befehlsausführungen Prozess 1 bereits wieder aktiv sein kann.

Wert von counter hängt vom Scheduling der Prozesse ab

Wechselseitiger Ausschluss

Zwei Prozesse wollen regelmäßig den kritischen Abschnitt aus höherer Programmierung betreten. Mit der Grundannahme, dass Maschinenbefehle unteilbar (atomar) sind, wäre eine Möglichkeit die Nutzung einer gemeinsamen Variablen:

```
int turn = 1;
```

Main Init

```
do {
   while (turn == 2);

   /* critical section */

   turn = 2;

}
while (true);
```

Prozess1

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

   /* critical section */

   turn = 1;

}
while (true);
```

Prozess2

Wechselseitiger Ausschluss

- Problem dieser Lösung ist, dass den kritischen Abschnitts jeweils der nächste Prozess nur dann betreten kann, wenn der andere gegenläufige Prozess diesen durchlaufen hat.
- Die Implementierung ist somit unvollständig und kann durch das Ersetzen von turn durch zwei unabhängige Variablen (Thread-Zustand-Flags) behoben werden:
  - prepare1 zeigt an, dass Prozess 1 bereit für den kritischen Abschnitt ist, er kann diesen aber nur betreten wenn Prozess 2 nicht ebenfalls diesen Zustand schon besitzt
  - prepare2 zeigt an, dass Prozess 2 bereit für den kritischen Abschnitt ist, er kann diesen aber nur betreten wenn Prozess 1 nicht ebenfalls diesen Zustand schon besitzt

Wechselseitiger Ausschluss

Implementation mit prepare1 und prepare2:

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Wechselseitiger Ausschluss

- In dieser Implementation wird ein wechselseitiger Ausschluss erreicht und die Prozesse können unabhängig von der vorhergehenden Ausführung auch mehrfach abgearbeitet werden.
- Großes Problem dieser Lösung ist aber die Möglichkeit der sogenannten Prozess Verklemmung (Deadlock).
- Die Prozess Verklemmung (Deadlock) erfolgt durch ein wechselseitiges Setzen der Zustandsvariablen.

Wechselseitiger Ausschluss

Implementation mit prepare1 und prepare2:

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Wechselseitiger Ausschluss

Implementation mit prepare1 und prepare2:

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Wechselseitiger Ausschluss

Implementation mit prepare1 und prepare2:

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Wechselseitiger Ausschluss

Implementation mit prepare1 und prepare2:

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Wechselseitiger Ausschluss

Implementation mit prepare1 und prepare2:

```
bool prepare1 = false;
bool prepare2 = false;
```

Main Init

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Verklemmung (Deadlock)

Algorithmus von Peterson

- Der Algorithmus von Peterson implementiert einen sicheren wechselseitigen Ausschluss, indem mit Hilfe der Zustandsvariable turn entschieden wird welcher Prozess nun wirklich den kritischen Abschnitt betreten darf.
- Entgegen der ersten Implementierung mit turn, ist es für einen Prozess jedoch nicht notwendig, dass der vorherige Prozess schon einen Durchlauf hatte, da hier zusätzlich die Thread Flags mit ausgewertet werden.
- Problem dieser Lösung ist die Implementation für mehrere Prozesse als zwei. Diese gestaltet sich dann wiederrum sehr aufwendig.

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Algorithmus von Peterson

```
bool prepare1 = false;
bool prepare2 = false;
int turn = 1;
```

Main Init

```
do {
   prepare1 = true;
   turn = 2;
   while (prepare2 && turn == 2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess1

```
do {
   prepare2 = true;
   turn = 1;
   while (prepare1 && turn == 1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Prozess2

Spinlocks

Problem der vorgestellten Lösungen ist jeweils das Problem des aktiven Wartens, in dem Prozess der blockiert wird. Dies verbraucht unter anderem Ressourcen wie Rechenleistung.

Das Problem das aktiven Wartens bezeichnet man auch als Spinlock.

Main Init

```
bool prepare1 = false;
bool prepare2 = false;
```

Prozess1

```
do {
   prepare1 = true;
   while (prepare2);

   /* critical section */

   prepare1 = false;

}
while (true);
```

Prozess2

```
do {
   prepare2 = true;
   while (prepare1);

   /* critical section */

   prepare2 = false;

}
while (true);
```

Semaphore

- Semaphor (von altgriechisch ‚Zeichen' + ‚tragen' – „Zeichenträger"):
  - ist eine Datenstruktur mit einer Initialisierungsoperation und zwei Nutzungsoperationen
  - die Datenstruktur besteht aus einem Zähler und einer Warteschlange für die Aufnahme blockierter Prozesse
  - Zähler sowie Warteschlange sind geschützt und können nur über die Semaphoroperationen verändert werden.
- Die Funktion der Semaphoren wird wie folgt in zwei Bestandteile aufgeteilt:
  - Semaphore regeln Wechselwirkungssituationen von Prozessen durch Zähler
  - Semaphore realisieren ein passives Warten der Prozesse, wenn eine Weiterausführung nicht gestattet werden kann

Vorteile einer Semaphor-Implementierung im Betriebssystem

- der Schedulers des Betriebssystems wird in die Semaphor-Operationen eingebunden
- im Gegensatz zum Lock bzw. Mutex brauchen die Threads, die „reservieren“ und „freigeben“, nicht identisch zu sein.
- Semaphore beheben den Nachteil des aktiven Wartens anderer Synchronisationslösungen (Spinlocks)
- ein Prozess der blockiert ist, wird bis der Blockadegrund entfallen ist in eine Warteschlange geschoben
- die Blockierzeit der Prozesse der Warteschlage kann durch andere Prozesse genutzt werden

Semaphore

- Semaphore als Mechanismus für die Prozesssynchronisation wurden von Edsger W. Dijkstra konzipiert und 1965 in „Cooperating sequential processes“ vorgestellt.
- Die Nutzungsoperationen wurden von Dijkstra mit P und V bezeichnet:
  - P-Operation = Wait (warten)
  - V-Operation = Release (freigeben)
- Aufruf der P-Operation:
  - Zähler wird dekrementiert
  - Zähler größer gleich 0 Prozess setzt seine Aktionen fort
  - Zähler kleiner als 0, Prozess wird blockiert und in die Warteschlange eingereiht
- Aufruf der V-Operation:
  - Zähler wird inkrementiert
  - Prozess wird aus der Warteschlange entnommen, entblockiert und fortgesetzt

Semaphore

- Szenario I - Recht auf Betreten eines kritischen Abschnitts
- Gemeinsam von A und B genutzter Semaphor: mutex
- Initialisierung: init(mutex, 1)

Prozess A

```
...

P(mutex)
… /* kritischer_Abschnitt_A */
V(mutex)

...
```

Prozess B

```
...

P(mutex)
… /* kritischer_Abschnitt_A */
V(mutex)

...
```

Semaphore

- Szenario II - Semaphor zur Betriebsmittelverwaltung
- Zur Betriebsmittelverwaltung genutzter Semaphor: s_available
- Initialisierung: init(s_available, n)

```
Prozess
...

P(s_available)
… /* Nutzung des Betriebsmittels */
V(s_available)

...
```

Semaphore

- Szenario III – Semaphor in Signalisierungsfunktion
- Gemeinsam von A und B genutzter Semaphor: s_inform
- Initialisierung: init(s_inform, 0)

Prozess A

```
...

C_I
V(s_inform)

...
```

Prozess B

```
...

P(s_inform)
C_V

...
```

Prozess B erst nach Prozess A ausführbar

Spezialfall Hyperthreading

- Hyper-Threading Technology (kurz HTT, üblicherweise nur Hyper-Threading und dann HT genannt) ist eine spezielle Implementierung von hardwareseitigem Multithreading in Intel-Prozessoren, die auch von AMD übernommen wurde.
- Hyper-Threading nutzt die Idee, die Rechenwerke eines Prozessors besser auszulasten, indem zwei Threads sich die Ressourcen teilen, die für einen vollständigen Kern notwendig wären.
- Hyper-Threading teilt mehrere vollständige Registersätze und ein komplexes Steuerwerk intern parallel arbeitende Pipeline-Stufen(Siblings) mit zwei parallelen Befehls- und Datenströmen zu

Kategorien CPU-Ressourcen

- replicated resources (replizierte Ressourcen):
  Diese werden von jedem Sibling unabhängig in eigener Kopie vorgehalten. Dazu zählen in jedem Fall der vollständige Registersatz inklusive Stackpointer und Befehlszähler.
- partitioned resources (unterteilte Ressourcen):
  Diese werden durch Unterteilung zwischen den Siblings aufgeteilt, das heißt, sie sind zwar nur einmal vorhanden, aber einzelne Teile der Ressourcen lassen sich genau einem Sibling zuordnen. Zu diesen gehören die Instruction Queues, der Reorder Buffer und die Load/Store Buffer.
- shared resources (geteilte Ressourcen):
  Alle übrigen Ressourcen gehören zu den geteilten Ressourcen und werden von beiden Siblings benutzt, meist dergestalt, dass sie nur von einem der Siblings gleichzeitig verwendet werden können. Hierzu zählen derzeit insbesondere die Arithmetisch-logische Einheit (ALU) und die Gleitkommaeinheit (FPU).

Zusammenfassung

ZUSAMMENFASSUNG

Ausgeführte Programme und Anwendungen in einem Betriebssystem werden nach dem Starten zu Prozessen.

Betriebssysteme sind für die Verwaltung von Prozessen zuständig. Prozesse in Multitasking Systemen müssen dazu Betriebsmittelzeit zugesprochen bekommen. Für das Umschalten der Prozesse gibt es den Scheduler der über verschiedene Auswahlstrategien die Prozesse auswählt und umschaltet.

Die Aufteilung von Prozessen in Threads ist neben den Multiprocess Systemen eine Möglichkeit Betriebsmittelressourcen effizienter einzusetzen.

Dafür das Prozesse oder Threads gleichzeitig parallel oder nebenläufig ausgeführt werden können bedarf es aber besonderer Techniken zum Schutz vor Inkonsistenzen.

Vielen Dank

Prof. Ronny Bodach

Hochschule Mittweida | University of Applied Sciences
Technikumplatz 17 | 09648 Mittweida
Fakultät Angewandte Computer- und Biowissenschaften

T +49 (0) 3727 58-1011
F +49 (0) 3727 58-21011
bodach@hs-mittweida.de
www.cb.hs-mittweida.de

Haus 8 | Richard-Stücklen Bau| Raum 8-205
Am Schwanenteich 6b| 09648 Mittweida

Felix Fischer

Hochschule Mittweida | University of Applied Sciences
Technikumplatz 17 | 09648 Mittweida
Fakultät Angewandte Computer- und Biowissenschaften

fische11@hs-mittweida.de
www.cb.hs-mittweida.de

HOCHSCHULE MITTWEIDA
University of Applied Sciences

hs-mittweida.de