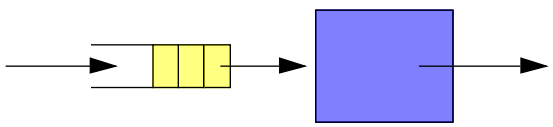
# D.3 Auswahlstrategien (3)

- Durchsatz
  - Möglichst hohe Anzahl bearbeiteter Prozesse pro Zeiteinheit
- Verweilzeit
  - Gesamtzeit des Prozesses in der Rechenanlage soll so gering wie möglich sein
- Wartezeit
  - Möglichst kurze Gesamtzeit, in der der Prozeß im Zustand „bereit“ ist
- Antwortzeit
  - Möglichst kurze Reaktionszeit des Prozesses im interaktiven Betrieb


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

# 1 First-Come, First Served

- Der erste Prozeß wird zuerst bearbeitet *(FCFS)*
  - „Wer zuerst kommt ...“
  - Nicht-verdrängend
- Warteschlange zum Zustand „bereit“
  - Prozesse werden hinten eingereiht
  - Prozesse werden vorne entnommen



- Bewertung
  - fair (?)
  - Wartezeiten nicht minimal
  - nicht für Time sharing-Betrieb geeignet

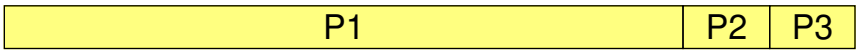

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# 1 First Come, First Served (2)

- Beispiel zur Betrachtung der Wartezeiten

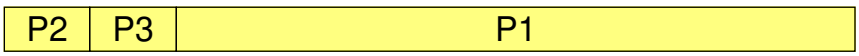
| | | |
|---|---|---|
| Prozeß 1: | 24 | } Zeiteinheiten |
| Prozeß 2: | 3 | |
| Prozeß 3: | 3 | |

  - Reihenfolge: P1, P2, P3




mittlere Wartezeit: (0+24+27)/3 = 17

  - Reihenfolge: P2, P3, P1




mittlere Wartezeit: (6+0+3)/3 = 3


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

# 2 Shortest Job First

- Kürzester Job wird ausgewählt *(SJF)*
  - Länge bezieht sich auf die nächste Rechenphase bis zur nächsten Warteoperation (z.B. Ein-, Ausgabe)
- „bereit“-Warteschlange wird nach Länge der nächsten Rechenphase sortiert
  - Vorhersage der Länge durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert, exponentielle Glättung)
  - ... Protokollierung der Länge der vorherigen Rechenphase
- SJF optimiert die mittlere Wartezeit
  - Da Länge der Rechenphase in der Regel nicht genau vorhersagbar, nicht ganz optimal.
- Variante: verdrängend *(PSJF)* und nicht-verdrängend


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

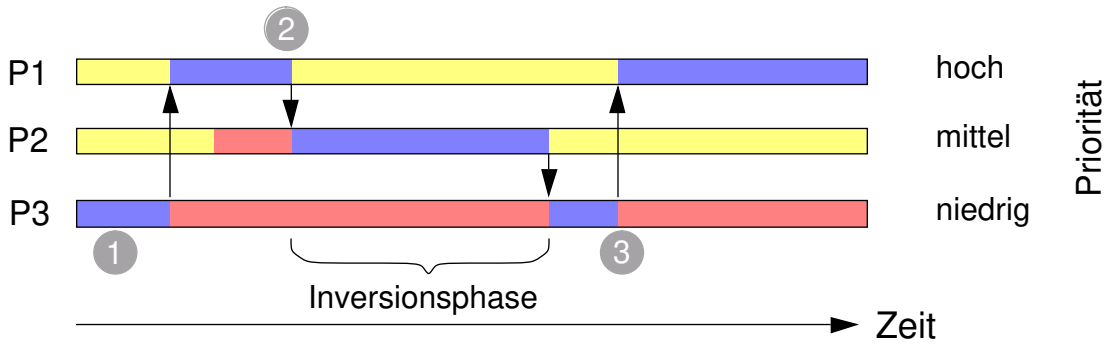
# 3 Prioritäten

- Prozeß mit höchster Priorität wird ausgewählt
  - dynamisch — statisch
    (z.B. SJF: dynamische Vergabe von Prioritäten gemäß Länge der nächsten Rechenphase)
    (z.B. statische Prioritäten in Echtzeitsystemen; Vorhersagbarkeit von Reaktionszeiten)
  - verdrängend — nicht-verdrängend
- Probleme
  - Aushungerung
    Ein Prozeß kommt nie zum Zuge, da immer andere mit höherer Priorität vorhanden sind.
  - Prioritätenumkehr *(Priority inversion)*


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

# 3 Prioritäten (2)

- Prioritätenumkehr
  - hochpriorer Prozeß wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozeß besitzt; dieser wiederum wird durch einen mittelprioren Prozeß verdrängt und kann daher das Betriebsmittel gar nicht freigeben




laufend
bereit
wartend (blockiert)

1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel
3. P3 gibt Betriebsmittel frei


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

# 3 Prioritäten (3)

★ Lösungen

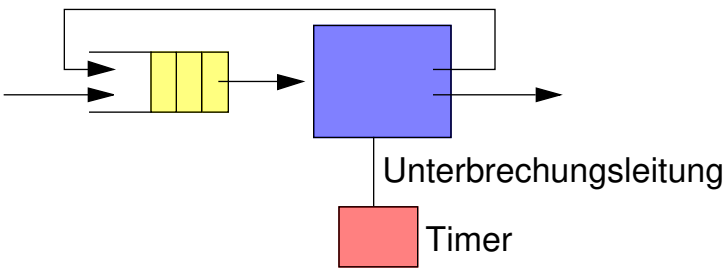
- ◆ zur Prioritätenumkehr:
  dynamische Anhebung der Priorität für kritische Prozesse
- ◆ zur Aushungerung:
  dynamische Anhebung der Priorität für lange wartende Prozesse
  (Alterung, *Aging*)


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

# 4 Round Robin Scheduling

■ Zuteilung und Auswahl erfolgt reihum

- ◆ ähnlich FCFS aber mit Verdrängung
- ◆ Zeitquant *(Time quantum)* oder Zeitscheibe *(Time slice)* wird zugeteilt
- ◆ geeignet für *Time sharing*-Betrieb




- ◆ Wartezeit ist jedoch eventuell relativ lang

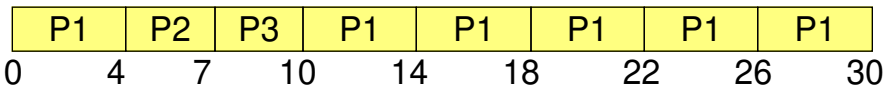

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# 4 Round Robin Scheduling (2)

- Beispiel zur Betrachtung der Wartezeiten

| | | |
|---|---|---|
| Prozeß 1: | 24 | } Zeiteinheiten |
| Prozeß 2: | 3 | |
| Prozeß 3: | 3 | |

  - Zeitquant ist 4 Zeiteinheiten
  - Reihenfolge in der „bereit“-Warteschlange: P1, P2, P3




mittlere Wartezeit: (6+4+7)/3 = 5.7


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

# 4 Round Robin Scheduling (3)

- Effizienz hängt von der Größe der Zeitscheibe ab
  - kurze Zeitscheiben: Zeit zum Kontextwechsel wird dominant
  - lange Zeitscheiben: Round Robin nähert sich FCFS an
- Verweilzeit und Wartezeit hängt ebenfalls von der Zeitscheibengröße ab
  - Beispiel: 3 Prozesse mit je 10 Zeiteinheiten Rechenbedarf
    - Zeitscheibengröße 1:
      durchschnittliche Verweilzeit: 29 Zeiteinheiten = (28+29+30)/3
      durchschnittliche Wartezeit: 19 Zeiteinheiten = (18+19+20)/3
    - Zeitscheibengröße 10:
      durchschnittliche Verweilzeit: 20 Zeiteinheiten = (10+20+30)/3
      durchschnittliche Wartezeit: 10 Zeiteinheiten = (0+10+20)/3


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

# 5 Multilevel Queue Scheduling

- Verschiedene Schedulingklassen
  - z.B. Hintergrundprozesse (Batch) und Vordergrundprozesse (interaktive Prozesse)
  - jede Klasse besitzt ihre eigenen Warteschlangen und verwaltet diese nach einem eigenen Algorithmus
  - zwischen den Klassen gibt es ebenfalls ein Schedulingalgorithmus z.B. feste Prioritäten (Vordergrundprozesse immer vor Hintergrundprozessen)
- Beispiel: Solaris
  - Schedulingklassen
    - Systemprozesse
    - Real-time Prozesse
    - Time-sharing Prozesse
    - interaktive Prozesse


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

# 5 Multilevel Queue Scheduling

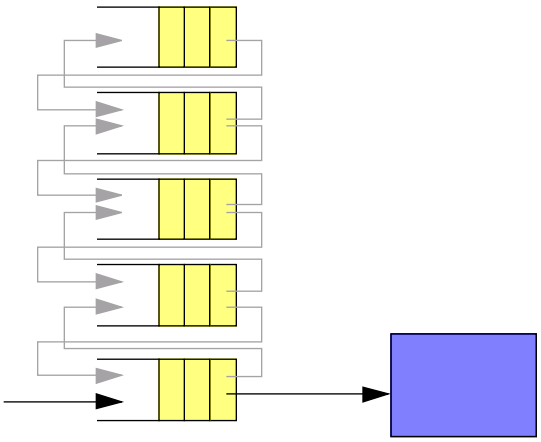
- Scheduling zwischen den Klassen mit fester Priorität (z.B. Real-time Prozesse vor Time sharing-Prozessen)
- In jeder Klasse wird ein eigener Algorithmus benutzt:
  - Systemprozesse: FCFS
  - Real-time Prozesse: statische Prioritäten
  - Time-sharing und interaktive Prozesse: ausgefeiltes Verfahren zur Sicherung von:
    - kurzen Reaktionszeiten
    - fairer Zeitaufteilung zwischen rechenintensiven und I/O-intensiven Prozessen
    - gewisser Benutzersteuerung

★ Multilevel Feedback Queue Scheduling


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

# 6 Multilevel Feedback Queue Scheduling

- Mehrere Warteschlangen *(MLFB)*
  - jede Warteschlange mit eigener Behandlung
  - Prozesse können von einer zur anderen Warteschlange transferiert werden




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

# 6 Multilevel Feedback Queue Scheduling (2)

- Beispiel:
  - mehrere Warteschlangen mit Prioritäten (wie bei Multilevel Queue)
  - Prozesse, die lange rechnen, wandern langsam in Warteschlangen mit niedrigerer Priorität (bevorzugt interaktive Prozesse)
  - Prozesse, die lange warten müssen, wandern langsam wieder in höherpriore Warteschlangen *(Aging)*


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

# 7 Beispiel: Time Sharing Scheduling in Solaris

- 60 Warteschlangen, Tabellensteuerung

| Level | ts_quantum | ts_tqexp | ts_maxwait | ts_lwait | ts_slpret |
|---|---|---|---|---|---|
| 0 | 200 | 0 | 0 | 50 | 50 |
| 1 | 200 | 0 | 0 | 50 | 50 |
| 2 | 200 | 0 | 0 | 50 | 50 |
| 3 | 200 | 0 | 0 | 50 | 50 |
| 4 | 200 | 0 | 0 | 50 | 50 |
| 5 | 200 | 0 | 0 | 50 | 50 |
| 6 | 200 | 0 | 0 | 50 | 50 |
| 7 | 200 | 0 | 0 | 50 | 50 |
| 8 | 200 | 0 | 0 | 50 | 50 |
| | | | ... | | |
| 44 | 40 | 34 | 0 | 55 | 55 |
| 45 | 40 | 35 | 0 | 56 | 56 |
| 46 | 40 | 36 | 0 | 57 | 57 |
| 47 | 40 | 37 | 0 | 58 | 58 |
| 48 | 40 | 38 | 0 | 58 | 58 |
| 49 | 40 | 39 | 0 | 59 | 58 |
| 50 | 40 | 40 | 0 | 59 | 58 |
| 51 | 40 | 41 | 0 | 59 | 58 |
| 52 | 40 | 42 | 0 | 59 | 58 |
| 53 | 40 | 43 | 0 | 59 | 58 |
| 54 | 40 | 44 | 0 | 59 | 58 |
| 55 | 40 | 45 | 0 | 59 | 58 |
| 56 | 40 | 46 | 0 | 59 | 58 |
| 57 | 40 | 47 | 0 | 59 | 58 |
| 58 | 40 | 48 | 0 | 59 | 58 |
| 59 | 20 | 49 | 32000 | 59 | 59 |


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

# 7 Beispiel: TS Scheduling in Solaris (2)

- Tabelleninhalt
  - kann ausgelesen und gesetzt werden
    (Auslesen: **`dispadmin -c TS -g`**)
  - **`Level:`** Nummer der Warteschlange
    Hohe Nummer = hohe Priorität
  - **`ts_quantum:`** maximale Zeitscheibe für den Prozeß (in Millisek.)
  - **`ts_tqexp:`** Warteschlangennummer, falls der Prozeß die Zeitscheibe aufbraucht
  - **`ts_maxwait:`** maximale Zeit für den Prozeß in der Warteschlange ohne Bedienung
    (in Sek.; min. 1)
  - **`ts_lwait:`** Warteschlangennummer, falls Prozeß zulange in dieser Schlange
  - **`ts_slpret:`** Warteschlangenummer für das Wiedereinreihen nach einer blockierenden Aktion


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

# 7 Beispiel: TS Scheduling in Solaris (3)

- Beispielprozeß:
  - 1000ms Rechnen am Stück
  - 5 E/A Operationen mit jeweils Rechenzeiten von 1ms dazwischen

| # | Warteschlange | Rechenzeit | Prozeßwechsel weil ... |
|---|---|---|---|
| *1* | 59 | 20 | Zeitquant abgelaufen |
| *2* | 49 | 40 | Zeitquant abgelaufen |
| *3* | 39 | 80 | Zeitquant abgelaufen |
| *4* | 29 | 120 | Zeitquant abgelaufen |
| *5* | 19 | 160 | Zeitquant abgelaufen |
| *6* | 9 | 200 | Zeitquant abgelaufen |
| *7* | 0 | 200 | Zeitquant abgelaufen |
| *8* | 0 | 180 | E/A Operation |
| *9* | 50 | 1 | E/A Operation |
| *10* | 58 | 1 | E/A Operation |
| *11* | 58 | 1 | E/A Operation |
| *12* | 58 | 1 | E/A Operation |


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

# 7 Beispiel: TS Scheduling in Solaris (4)

- Tabelle gilt nur unter der folgenden Bedingung:
  - Prozeß läuft fast alleine, andernfalls
    - könnte er durch höherpriore Prozesse verdrängt und/oder ausgebremst werden,
    - wird er bei langem Warten in der Priorität wieder angehoben.
- Beispiel:

| # | Warteschlange | Rechenzeit | Prozeßwechsel weil ... |
|---|---|---|---|
| | | ... | |
| *6* | 9 | 200 | Zeitquant abgelaufen |
| *7* | 0 | 20 | Wartezeit von 1s abgelaufen |
| *8* | 50 | 40 | Zeitquant abgelaufen |
| *9* | 40 | 40 | Zeitquant abgelaufen |
| *10* | 30 | 80 | Zeitquant abgelaufen |
| *11* | 20 | 120 | Zeitquant abgelaufen |
| | | ... | |


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

# 7 Beispiel: TS Scheduling in Solaris (5)

- Weitere Einflußmöglichkeiten
  - Anwender und Administratoren können Prioritätenoffsets vergeben
  - Die Offsets werden auf die Tabellenwerte addiert und ergeben die wirklich verwendete Warteschlange
  - positive Offsets: Prozeß wird bevorzugt
  - negative Offsets: Prozeß wird benachteiligt
  - Außerdem können obere Schranken angegeben werden
- Systemaufruf
  - Verändern der eigenen Prozeßpriorität

```
int nice( int incr );
```

(positives Inkrement: niedrigere Priorität;
negatives Inkrement: höhere Priorität)


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

# D.4 Prozeßkommunikation

- *Inter process communication (IPC)*
  - Mehrere Prozesse bearbeiten eine Aufgabe
    - gleichzeitige Nutzung von zur Verfügung stehender Information durch mehrere Prozesse
    - Verkürzung der Bearbeitungszeit durch Parallelisierung
- Kommunikation durch Nachrichten
  - Nachrichten werden zwischen Prozessen ausgetauscht
- Kommunikation durch gemeinsamen Speicher
  - F. Hofmann nennt dies Kooperation (kooperierende Prozesse)


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

# D.4 Prozeßkommunikation (2)

- Klassifikation nachrichtenbasierter Kommunikation
  - ◆ Klassen
    - Kanäle *(Pipes)*
    - Kommunikationsendpunkte *(Sockets, Ports)*
    - Briefkästen, Nachrichtenpuffer *(Queues)*
    - Unterbrechungen *(Signals)*
  - ◆ Übertragunsrichtung
    - unidirektional
    - bidirektional (voll-duplex, halb-duplex)


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

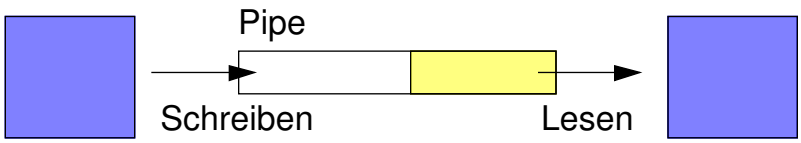
# D.4 Prozeßkommunikation (3)

- ◆ Übertraguns- und Aufrufeigenschaften
  - zuverlässig — unzuverlässig
  - gepuffert — ungepuffert
  - blockierend — nichtblockierend
  - stromorientiert — nachrichtenorientiert — RPC
- ◆ Adressierung
  - implizit: UNIX Pipes
  - explizit: Sockets
  - globale Adressierung: Sockets, Ports
  - Gruppenadressierung: Multicast, Broadcast
  - funktionale Adressierung: Dienste


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

# 1 Pipes

- Kanal zwischen zwei Kommunikationspartnern
  - unidirektional
  - gepuffert (feste Puffergröße), zuverlässig, stromorientiert




- Operationen: Schreiben und Lesen
  - Ordnung der Zeichen bleibt erhalten (Zeichenstrom)
  - Blockierung bei voller Pipe (Schreiben) und leerer Pipe (Lesen)


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

# 1 Pipes (2)

- Systemaufruf unter Solaris
  - Öffnen einer Pipe

```
int pipe( int fdes[2] );
```

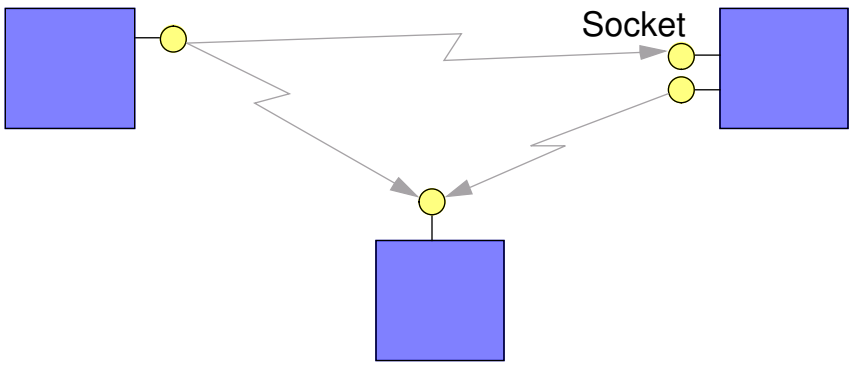
  - Es werden eigentlich zwei Pipes geöffnet
    **`fdes[0]`** liest aus Pipe 1 und schreibt in Pipe 2
    **`fdes[1]`** liest aus Pipe 2 und schreibt in Pipe 1

  - Zugriff auf Pipes wie auf eine Datei: **`read`** und **`write`**, **`readv`** und **`writev`**


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

# 2 Sockets

- Allgemeine Kommunikationsendpunkte
  - ◆ bidirektional, gepuffert




  - ◆ Auswahl einer Protokollfamilie
    - z.B. TCP/IP, UNIX (innerhalb von Prozessen der gleichen Maschine)


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

# 2 Sockets (2)

- ◆ Auswahl eines Protokolls der Familie
  - z.B. UDP
  - Wahl zwischen zuverlässigen und unzuverlässigen Protokollen
  - Wahl zwischen stromorientierten (verbindungsorientierten) und nachrichtenorientierten Protokollen
- ◆ explizite Adressierung
  - Unicast: genau ein Kommunikationspartner
  - Multicast: eine Gruppe
  - Broadcast: alle möglichen Adressaten
- ◆ können blockierend und nichtblockierend betrieben werden


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

# 3 UNIX Queues

- Nachrichtenpuffer *(Queue, FIFO)*
  - Rechnerlokale Adresse *(key)*, prozeßlokale Nummer *(msqid)* ähnlich dem Filedeskriptor
  - Zugriffsrechte wie auf Dateien
  - ungerichtete Kommunikation, gepuffert (einstellbare Größe pro Queue)
  - Nachrichten haben einen Typ (long-Wert)
  - Operationen zum Senden und Empfangen einer Nachricht
  - blockierend — nichtblockierend, alle Nachrichten — nur ein bestimmter Typ
- Systemaufrufe unter Solaris 2.5
  - Erzeugen einer Queue bzw. Holen einer MSQID

```
int msgget( key_t key, int msgflg );
```


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

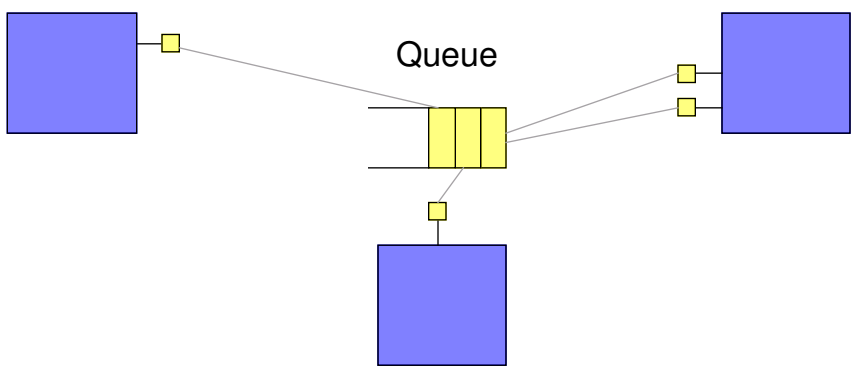
# 3 UNIX Queues (2)

- Senden einer Nachricht

```
int msgsnd( int msqid, const void *msgp, size_t msgsz,
            int msgflg);
```

- Empfangen einer Nachricht

```
int msgrcv( int msqid, void *msgp, size_t msgsz,
            long msgtyp, int msgflg);
```

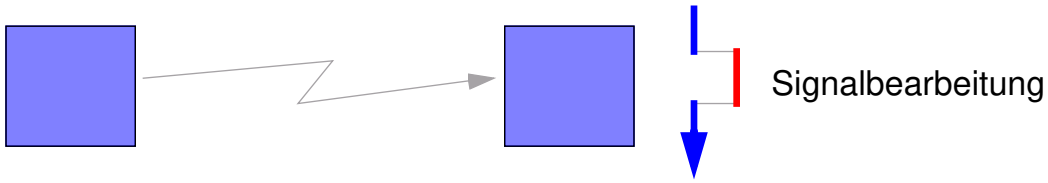




Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# 4 UNIX Signale

- Signale sind Unterbrechungen ähnliche denen eines Prozessors
  - Prozeß führt eine definierte Signalbehandlung durch
    - Ignorieren
    - Terminierung des Prozesses
    - Aufruf einer Funktion
  - Nach der Behandlung läuft Prozeß an unterbrochener Stelle weiter





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

# 4 UNIX Signale (2)

- Kommunikation über Signale
  - Signalisieren von Ereignissen
    z.B. Start und Ende einer Bearbeitung
  - festgelegte Signaltypen
- UNIX Systemaufrufe
  - Einfangen von Signalen

```
void (*signal( int sig, void (*disp)( int ) ))( int );
```

  - Zustellen von Signalen

```
int kill( pid_t pid, int sig );
```

- Signale implementieren externe Beendigung von Prozessen
  - bestimmtes Signal kann nicht ignoriert werden und führt zur endgültigen Terminierung


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

# 5 Fernaufruf *(RPC)*

- Funktionsaufruf über Prozeßgrenzen hinweg *(Remote procedure call)*
  - ◆ hoher Abstraktionsgrad
  - ◆ selten wird Fernaufruf direkt vom System angeboten; benötigt Abbildung auf andere Kommunikationsformen z.B. auf Nachrichten
  - ◆ Abbildung auf mehrere Nachrichten
    - Auftragsnachricht transportiert Aufrufabsicht und Parameter
    - Ergebnisnachricht transportiert Ergebnisse des Aufrufs


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

# 6 Gemeinsamer Speicher

- Zwei Prozesse können auf einen gemeinsamen Speicherbereich zugreifen
  - ◆ gemeinsame Variablen und Datenstrukturen (ähnlich wie bei Threads des selben Tasks)
- Näheres erst im Abschnitt E.5


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

# D.5 Koordinierung

- Beispiel: Beobachter und Protokollierer
  - Mittels Induktionsschleife werden Fahrzeuge gezählt. Alle 10min druckt der Protokollierer die im letzten Zeitraum vorbeigekommene Anzahl aus.

```
int cnt= 0;

Observer
    on indication
        cnt= cnt+1;

Logger
    every 10 minutes
        printf( "Count= %d\n", cnt );
        cnt= 0;
```


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

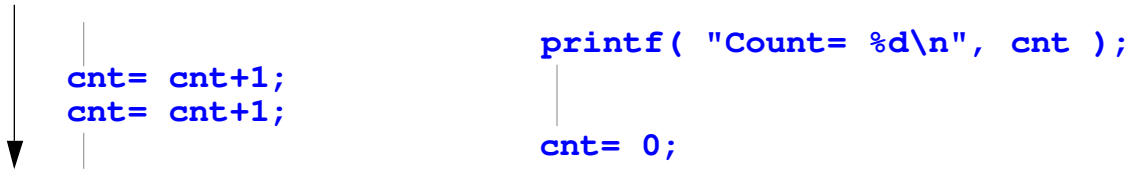
# D.5 Koordinierung (2)

- Effekte:
  - Fahrzeuge gehen „verloren“
  - Fahrzeuge werden doppelt gezählt
- Ursachen:
  - Befehle in C werden nicht atomar abgearbeitet, da sie auf mehrere Maschinenbefehle abgebildet werden.
  - Fahrzeuge gehen „verloren“:
    Nach dem Drucken wird der Protokollierer unterbrochen. Beobachter zählt weitere Fahrzeuge. Anzahl wird danach ohne Beachtung vom Protokollierer auf Null gesetzt.
  - Fahrzeuge werden doppelt gezählt:
    Beobachter will Zähler erhöhen und holt sich diesen dazu in ein Register. Er wird unterbrochen und der Protokollierer setzt Anzahl auf Null. Beobachter erhöht Registerwert und schreibt diesen zurück. Dieser Wert wird erneut vom Protokollierer registriert.

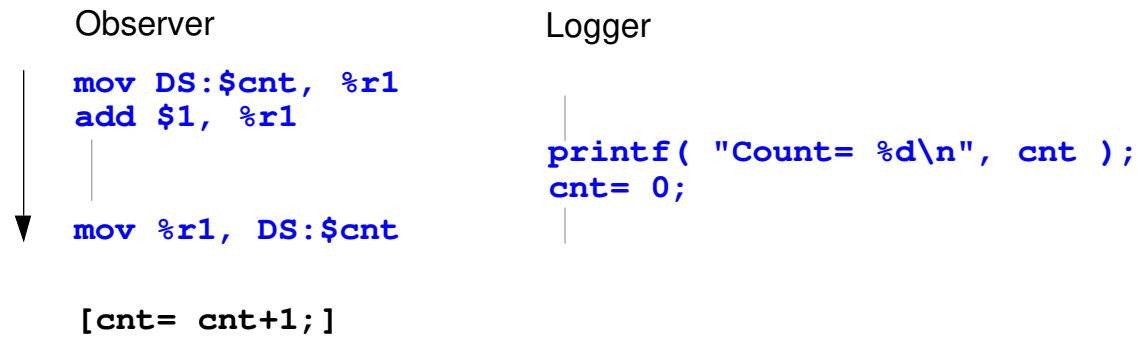

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# D.5 Koordinierung (3)

◆ Fahrzeuge gehen „verloren“:




◆ Fahrzeuge werden doppelt gezählt:





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

# D.5 Koordinierung (4)

- Gemeinsame Nutzung von Daten oder Betriebsmitteln
  - ◆ kritische Abschnitte:
    - nur einer soll Zugang zu Daten oder Betriebsmitteln haben (gegenseitiger Ausschluß, *Mutual exclusion*, *Mutex*)
    - kritische Abschnitte erscheinen allen anderen als zeitlich unteilbar
  - ◆ Wie kann der gegenseitige Ausschluß in kritischen Abschnitten erzielt werden?
- Koordinierung allgemein:
  - ◆ Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern


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

# 1 Gegenseitiger Ausschluß

- Zwei Prozesse wollen regelmäßig kritischen Abschnitt betreten
  - Annahme: Maschinenbefehle sind unteilbar (atomar)
- 1. Versuch

```
int turn= 0;
```

Prozeß 0

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

    ... /* critical sec. */

    turn= 1;

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

Prozeß 1

```
while( 1 ) {
    while( turn == 0 );

    ... /* critical sec. */

    turn= 0;

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


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

# 1 Gegenseitiger Ausschluß (2)

- Probleme der Lösung
  - nur alternierendes Betreten des kritischen Abschnitts durch $P_0$ und $P_1$ möglich
  - Implementierung ist unvollständig
  - aktives Warten
- Ersetzen von **`turn`** durch zwei Variablen **`ready0`** und **`ready1`**
  - **`ready0`** zeigt an, daß Prozeß 0 bereit für den kritischen Abschnitt ist
  - **`ready1`** zeigt an, daß Prozeß 1 bereit für den kritischen Abschnitt ist


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

# 1 Gegenseitiger Ausschluß (3)

- 2. Versuch

```
bool ready0= FALSE;
bool ready1= FALSE;
```

Prozeß 0

```
while( 1 ) {
    ready0= TRUE;
    while( ready1 );

    ... /* critical sec. */

    ready0= FALSE;

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

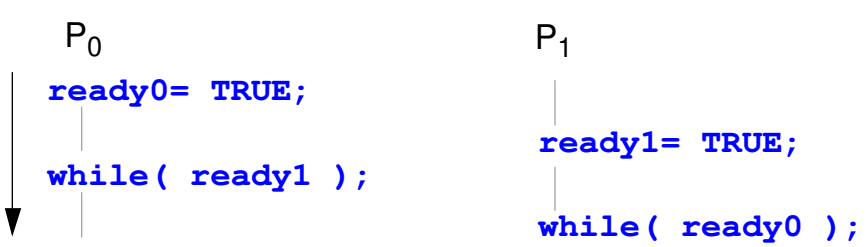
Prozeß 1

```
while( 1 ) {
    ready1= TRUE;
    while( ready0 );

    ... /* critical sec. */

    ready1= FALSE;

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


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

# 1 Gegenseitiger Ausschluß (4)

- Gegenseitiger Ausschluß wird erreicht
- Probleme der Lösung
  - ◆ aktives Warten
  - ◆ Verklemmung möglich *(Lifelock)*




- Kombination beider Algorithmen führt zu Versuch 3


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

# 1 Gegenseitiger Ausschluß (5)

- 3. Versuch (Algorithmus von Peterson, 1981)

```
bool ready0= FALSE;
bool ready1= FALSE;
int turn= 0;
```

Prozeß 0

```
while( 1 ) {
    ready0= TRUE;
    turn= 1;
    while( ready1 &&
           turn == 1 );

    ... /* critical sec. */

    ready0= FALSE;

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

Prozeß 1

```
while( 1 ) {
    ready1= TRUE;
    turn= 0;
    while( ready0 &&
           turn == 0 );

    ... /* critical sec. */

    ready1= FALSE;

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


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

# 1 Gegenseitiger Ausschluß (6)

- Algorithmus implementiert gegenseitigen Ausschluß
  - vollständige und sichere Implementierung
  - **`turn`** entscheidet für den kritischen Fall von Versuch 2, welcher Prozeß nun wirklich den kritischen Abschnitt betreten darf
  - in allen anderen Fällen ist **`turn`** unbedeutend
- Problem der Lösung
  - aktives Warten
- Algorithmus auch für mehrere Prozesse erweiterbar
  - Lösung ist relativ aufwendig


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