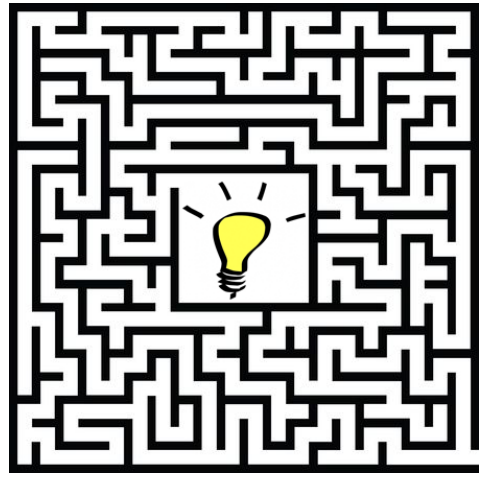
Advanced IT

Prof. Dr. Henning Pagnia

DHBW Mannheim

Herbst 2023



Wichtiges zur Vorlesung

Prof. Dr. Henning Pagnia

- Wirtschaftsinformatik
- Email: *pagnia@dhbw-mannheim.de*
- Telefon: *0621 / 4105-1131*
- Raum: *149 B*

Themengebiete

- Shell-Programmierung
- Prozesse und Threads
- Synchronisation
  - Synchronisation im BS-Kern
  - Semaphore
  - Standardsynchronisationsprobleme
  - Monitorkonzept
- Sockets
  - TCP
  - UDP
  - Programmierung in Java
  - Multi-Threaded Server

Literaturliste (Bücher)

- [Abt2015] Masterkurs Client/Server-Programmierung mit Java, 4. Auflage, D. Abts, Springer Vieweg, 2015
- [Man2017] TCP und UDP Internals, P. Mandl, Springer Vieweg, 2017
- [TB2016] Moderne Betriebssysteme, 4. akt. Auflage, A. S. Tanenbaum, H. Bos, Pearson Deutschland, 2016
- [TW2013] Computer Networks, 5th edition, A. S. Tanenbaum, D. J. Wetherall, Prentice Hall, 2013
- [Ull2016] Java ist auch eine Insel , 12. Auflage, C. Ullenboom, Rheinwerkverlag, 2016. URL: http://openbook.rheinwerk-verlag.de/javainsel/

Unix

Allgemeines

Entwicklung

- 1969 Bell Laboratories
- Mehrbenutzerbetriebssystem
- zu großen Teilen in C geschrieben
- für Großrechner, Server, Computer-Cluster, Supercomputer, Desktops, Notebooks, Smartphones, Embedded Devices, Unterhaltungselektronik

Derivate

- proprietär: MacOS, HP-UX, AIX, ...
- Open-Source: Free-BSD, Linux, Android, ...

POSIX (Portable Operating System Interface)

- standardisierte Programmierschnittstelle für Unix'e
- Konzepte und Konventionen
- System-Schnittstelle (z. B. System Calls)
- Shell und Hilfsprogramme

Shell

Definition

- textbasierte Benutzungsschnittstelle
- Programm, das vom login-Prozess gestartet wird, Kommandos entgegen nimmt und diese dann ausführt

Typische Funktionen

- Stoppen und Starten von Programmen als Prozesse im Vorder- oder Hintergrund, Verkettung über Pipes
- Abbruch eines Programms mittels `Ctrl-C`
- Bereitstellen von Umgebungsvariablen (z. B. $HOME, $PATH)
- Ein- / Ausgabeumlenkung
- Bedingungen (if, case) und Schleifen (while, for)
- interne Kommandos (cd, read)
- Editieren der Kommandozeile und früherer Kommandos

Bash

Bourne-Again-Shell

- freie Unix-Shell mit vielen Funktionen

Platzhalter

*	für beliebig viele Zeichen (also auch null)
?	für genau ein Zeichen

Ein- Ausgabeumleitungen

<	Umleitung der Standardeingabe stdin (Kanal 0)
>	Umleitung der Standardausgabe stdout (Kanal 1); auch >>
2>	Umleitung der Standardfehlerausgabe stderr (Kanal 2)
|	Pipe: verbindet Ausgabe mit Eingabe zweier Kommandos
`/dev/null`	Gerätedatei: Nirwana ...
`/dev/zero`	Gerätedatei: liefert eine endlose Folge von Null-Bits
`/dev/urandom`	Gerätedatei: liefert eine endlose Folge von Zufalls-Bits

Bash (Forts.)

Kurzer Exkurs: Programmierung

- Bash-Shell-Programme beginnen mit `#!/bin/bash`
- Kommentare sind Zeilen, die mit # beginnen
- `read` liest eine Eingabe von `stdin`
- if-Anweisung (Syntax)

```
if [ Bedingung ] ; then
  # falls Bedingung wahr
else
  # falls Bedingung falsch
fi
```

- while-Schleife (Syntax)

```
while [ Bedingung ] ; do
  # Befehl(e) in Schleife
done
```

- for-Schleife (Beispiel)

```
for i in $( ls ); do
            echo item: $i
        done
```

Unix-Befehle

Wichtige System-Befehle

`man`	der Manual-Befehl; Aufruf von Hilfeseiten
`cd`	Wechsel des Arbeitsverzeichnisses
`ls`	Auflisten von Dateinamen in Verzeichnissen
`ls -l`	lange Ausgabe inkl. Zugriffsrechte, Besitzername usw.
`rm`	Löschen von Dateien und Verzeichnissen (Parameter: `-r -f`)
`cp`	Kopieren von Dateien
`ln`	Anlegen eines Hard-Link
`ln -s`	Anlegen eines Symbolic-Link
`chmod`	Ändern von Zugriffsrechten
`sudo`	Ausführung eines Kommandos mit Systemrechten (root)

Unix-Befehle (Forts.)

Ausgabe- und Filter-Befehle

`echo`	Ausgabe erzeugen, z. B. `echo "Hallo"`
`cat`	Ausgeben des Dateiinhalts
`less`	seitenweises Ausgeben des Dateiinhalts (mit Stopps)
`head`	Ausgeben des Dateianfangs
`tail`	Ausgeben des Dateiendes
`wc`	Zählen von Zeichen, Wörtern und Zeilen
`grep`	Filtern nach Mustern

Unix-Befehle (Forts.)

Systeminformationen

`ps`	Liste der aktuellen Prozesse und deren Zustände
`top`	Systemübersicht inkl. sortierter, sich selbst aktualisierender Liste der aktuellen Prozesse
`df`	freier Speicherplatz von Datenspeichern
`du`	belegter Speicherplatz von Dateien / Verzeichnisssen
`id`	Informationen über die eigene Kennung (z. B. UID, Gruppen)
`mount`	Übersicht über gemountete Dateisysteme und deren Mount-Punkte
`dmesg`	Liste der Betriebssystemmeldungen und -fehlerinformationen (nur root)

Unix-Befehle (Forts.)

Tools

`nano` nicht-grafischer Text-Editor
Ctrl-O: Datei sichern
Ctrl-X: Editor verlassen

`find` durchsucht Verzeichnisse rekursiv nach (Teilen von) Dateinamen

`sed`* Stream Editor, z. B. ersetzen von Mustern
Beispiel: `echo IT 1 | sed 's/1/Advanced/'`
Tutorium: `https://tty1.net/sed-tutorium/index.html`

`awk`* interpretierte Skriptsprache mit C-ähnlicher Syntax
zum Bearbeiten von Dateien und Filtern von Ein- und Ausgaben
Beispiele: `echo Advanced IT | awk '{print $2}'`

```
awk '{if (NR>1){sum = sum+$2; max++}} END{print "avg =", sum/max}' notenliste
```

*Buch über die Shell-Programmierung inkl. sed und awk:
`http://openbook.rheinwerk-verlag.de/shell_programmierung/index.htm`

Nebenläufigkeit

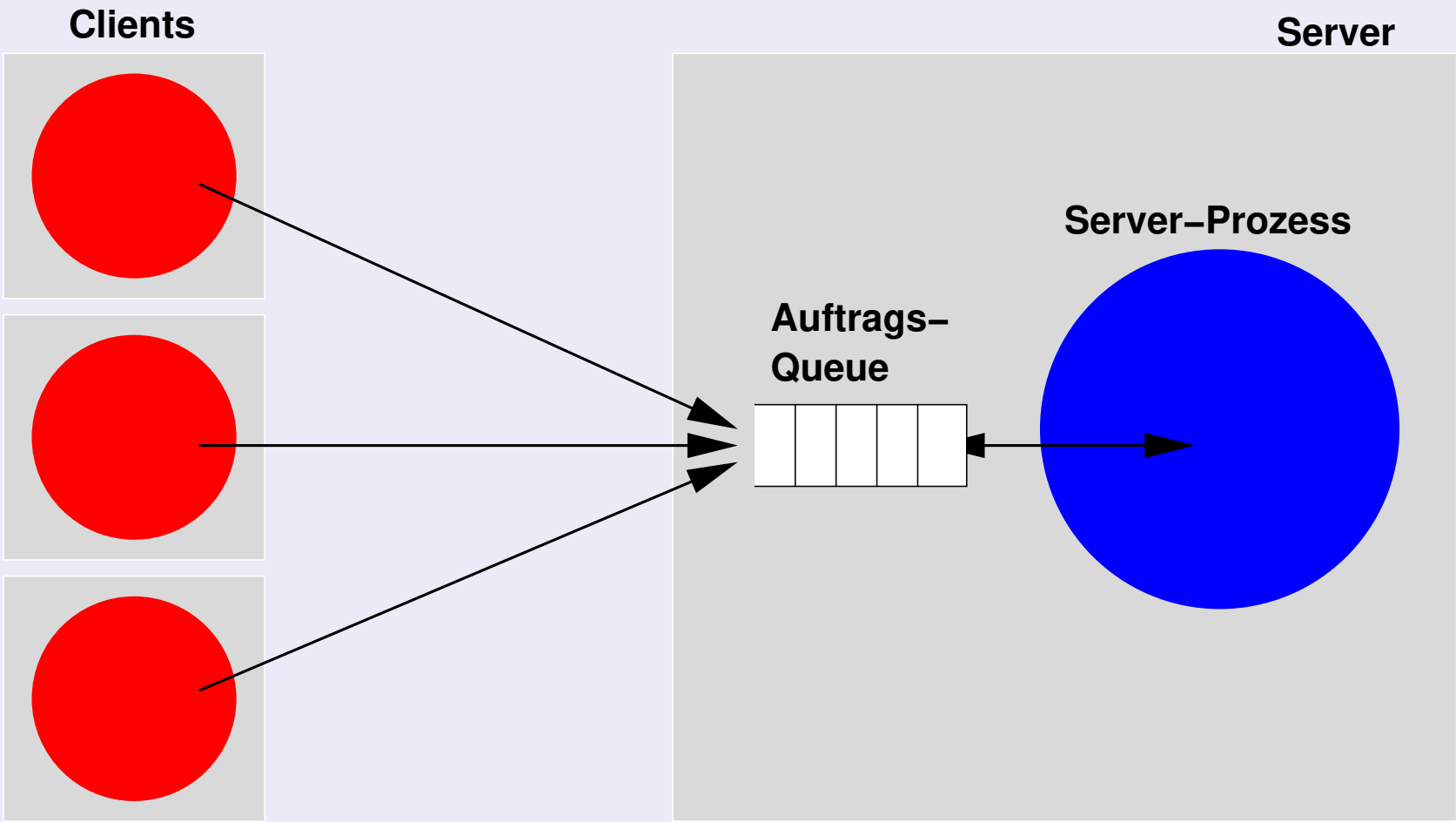
Einführung

Begriffe

- **Programm:**
  passiv, besteht aus Folge von (ausführbaren) Maschinenbefehlen
- **Prozess:**
  stellt *Ausführungsumgebung* für Programme bereit
  - Prozesszustände: Running, Ready, Blocked
- **Multi-Processing System:**
  System, das mehrere Prozesse gleichzeitig laden kann
  - jeder Prozess besitzt einen eigenen Adressraum
  - Adressraumtrennung
- **Time-Sharing System:**
  Prozesse werden in sehr kurzen Intervallen alternierend ausgeführt
  - Prozessor-Multiplexing
  - Programme werden auf Einprozessorsystem scheinbar parallel ($\Rightarrow$ *pseudoparallel*) abgearbeitet
  - Programmierer muss sich *(fast)* nicht um Nebenläufigkeit kümmern
  - Jeder Prozess besitzt scheinbar eigenen (virtuellen) Prozessor

Prozesse

Beispiel: Client/Server-Architektur mit sequenziellem Server



Prozesse (Forts.)

Sequenzieller Server

- wird in einem Prozess ausgeführt
- arbeitet Client-Aufträge sequenziell ab
  Nachteil: niedriger Durchsatz
  - durch häufige Prozesswechsel kommt es zu längeren Wartezeiten
  - Multi-Core Architekturen werden nicht ausgenutzt
- **Besser:** parallele Abarbeitung mittels *Multi-Processing*

Scheduling von Prozessen

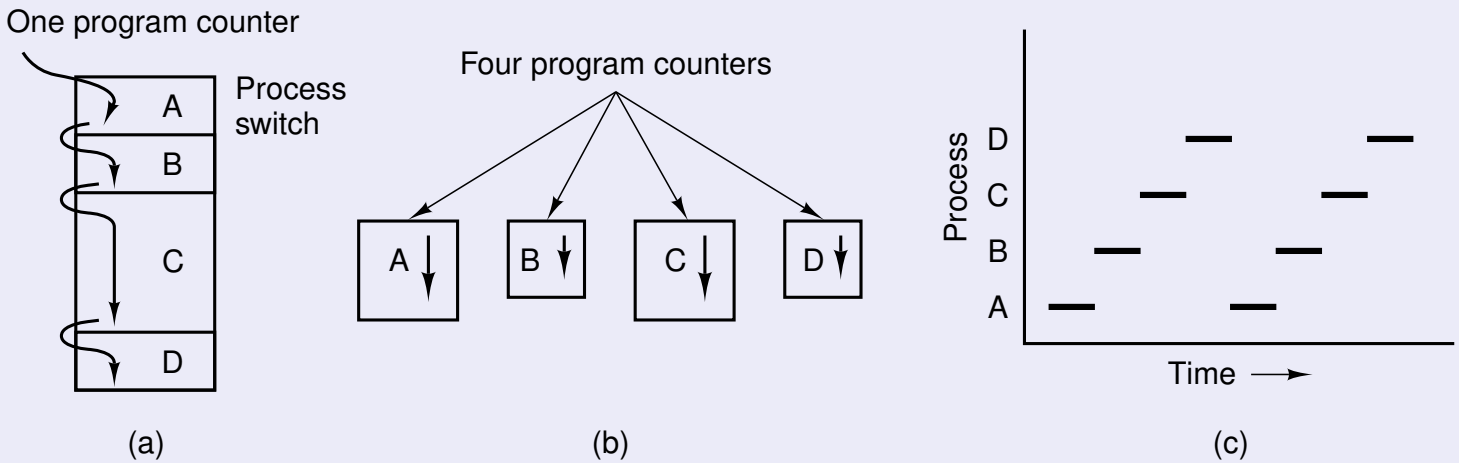
- Round-Robin ($\Rightarrow$ Time-Sharing-Verhalten mittels Zeitscheiben)
  - i. Allg. nicht für Echtzeitanforderungen ($\Rightarrow$ **Wieso?**)
- Prioritäten gesteuert (auch: *preemptive*)
- Kombination ($\Rightarrow$ Multi-Level-Feedback)

Prozesse (Forts.)

Time-Sharing

- Wann kommt es zu Prozesswechseln?
  - nach Ablauf der Zeitscheibe
  - immer dann, wenn ein aktiver Prozess warten muss (synchrone E/A, Seitenfehler, expliziter Wartebefehl)

Verschiedene Sichten



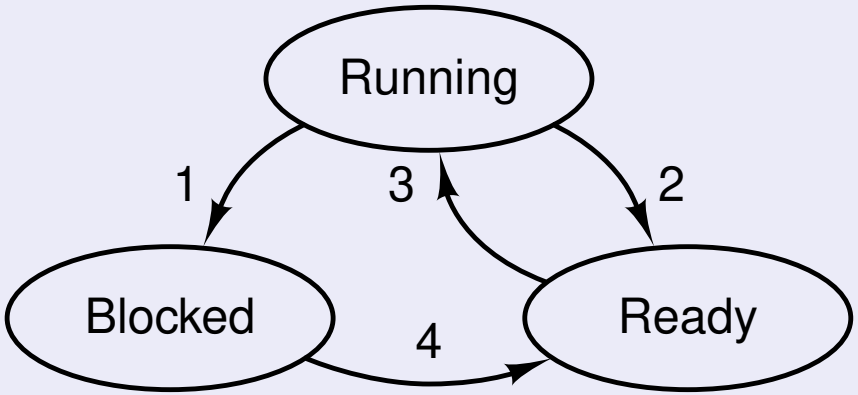
(a) Program Counter, (b) konzeptuelles Modell, (c) Sicht des Prozessors

Prozesse (Forts.)

Prozesszustandsmodell (für ein Einprozessorsystem)

- Struktur
  - es ist immer genau *ein* Prozess *aktiv* (running), ggf. der Idle-Prozess
  - weitere Prozesse können *ablaufbereit* (ready) sein
  - andere Prozesse sind auf ein bestimmtes Ereignis *wartend* (blocked)
- Vorteil des Modells: vereinfachtes Denken auch über komplexe Systemabläufe

Ein einfaches Prozesszustandsmodell



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Scheduler: BS-Komponente, die bestimmt, welcher Prozess aktiviert wird

Threads

Nachteile von Multi-Processing innerhalb einer Anwendung

- Prozesswechsel bedingt Adressraumwechsel
- hoher Zeitaufwand bei Adressraumwechsel wegen notwendiger Invalidierung der Hardware-Caches
- Bearbeitung von gemeinsam verwendeten Daten erfordert System-Call ($\Rightarrow$ aufwändiger Kontextwechsel ins BS)

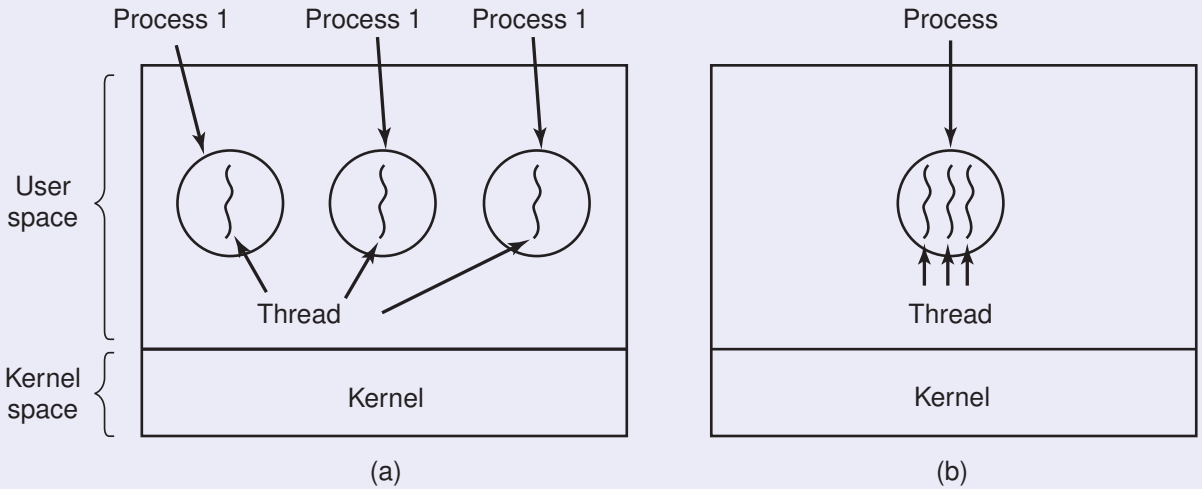
$\Rightarrow$ Prozesse sind vergleichsweise langsam!

Thread

- **Thread of Execution** $\Rightarrow$ *Thread* (dtsch. Kontrollfluss, Faden)
- Definition von Nebenläufigkeit *innerhalb* eines Prozesses: mehrere nebenläufige Kontrollflüsse innerhalb eines Adressraums
- Vorteile:
  - Threads teilen sich Adressraum und Ressourcen
    $\Rightarrow$ Hardware-Caches müssen nicht invalidiert werden
    $\Rightarrow$ keine System-Calls nötig, um Daten zu teilen

Threads (Forts.)

Multi-Processing vs. Multi-Threading



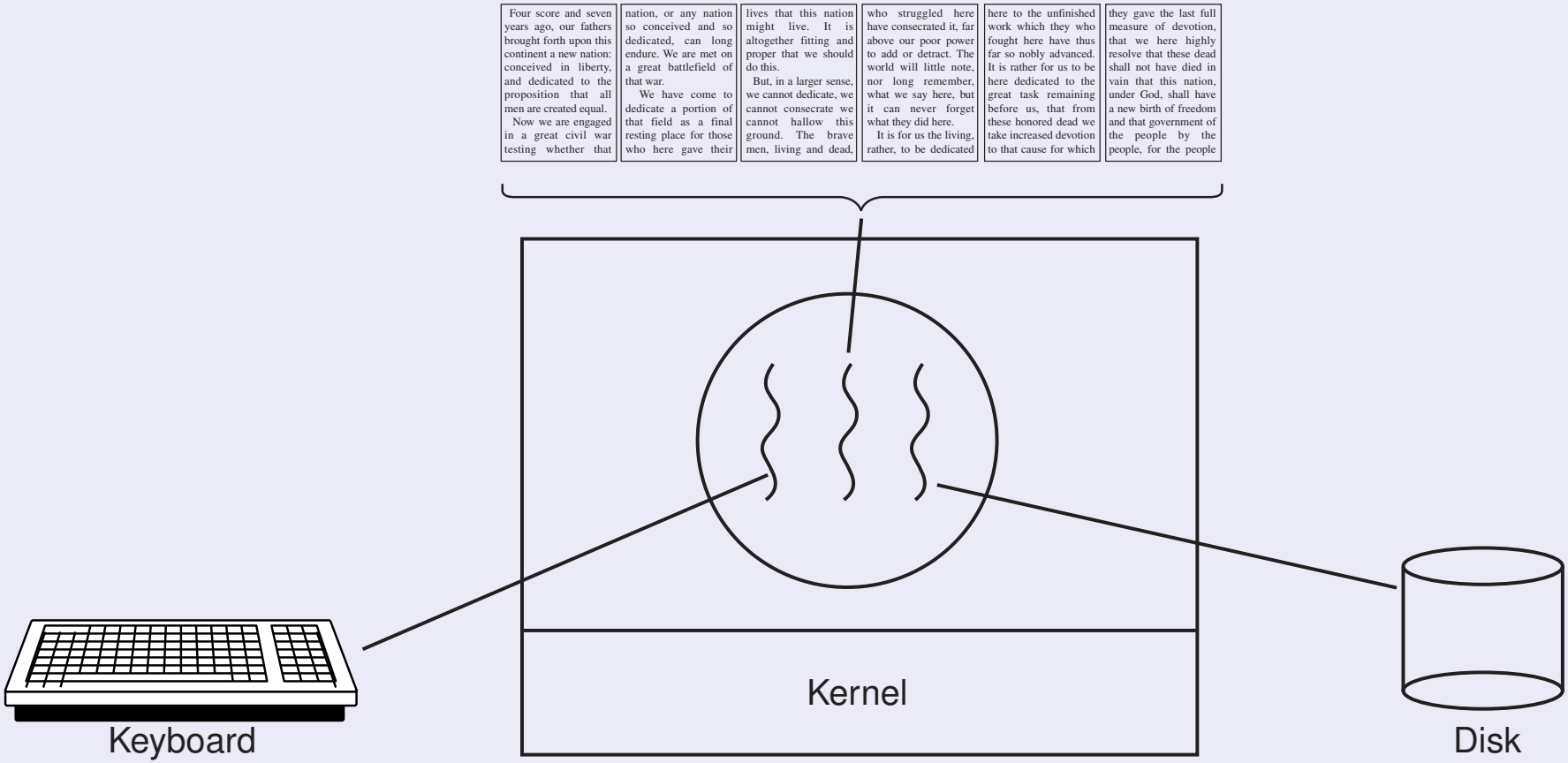
(a) Drei Prozesse mit je einem Thread, (b) ein Prozess mit drei Threads

Was gehört zu einem Thread?

- Zustand (Running, Ready, Blocked)
- Program-Counter
- Stack
- Register (halten momentanen Funktionszustand)

Threads (Forts.)

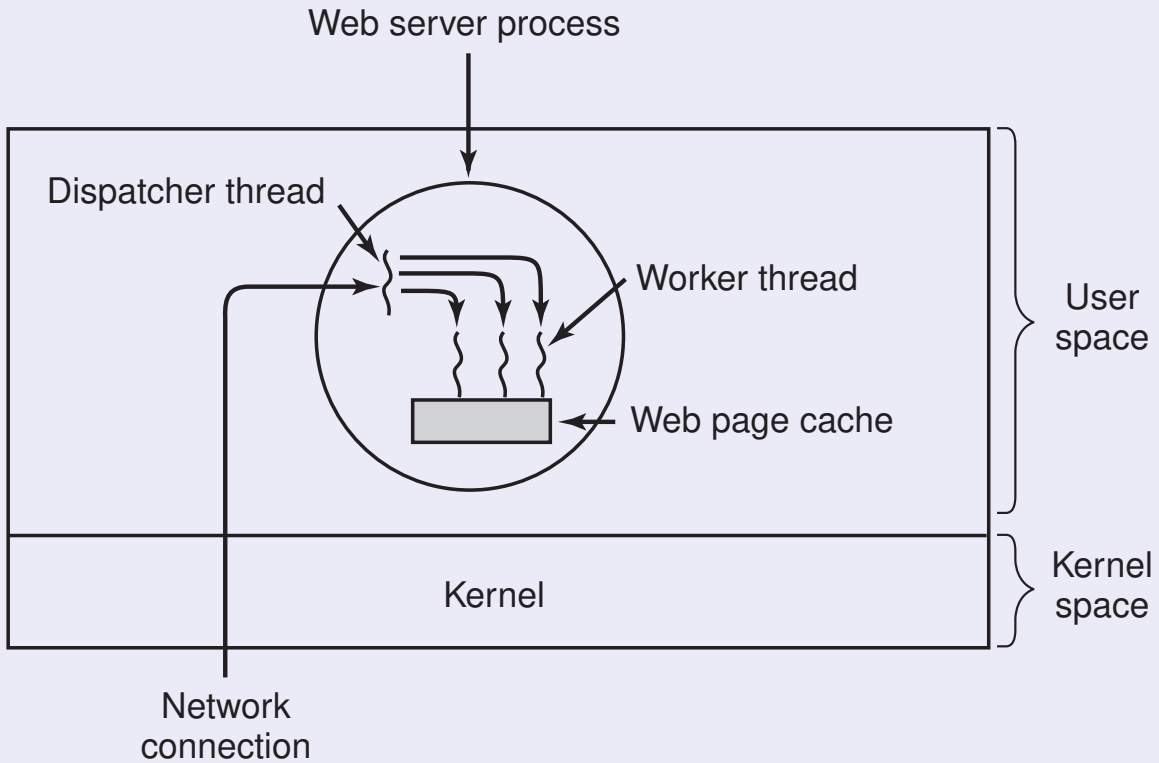
Beispiel: Textverarbeitung mit 3 Threads



Tastaturbedienung, Textformatierung, Dateispeicherung

Threads (Forts.)

Beispiel: Multi-Threaded Webserver



Dispatcher-Code:

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

Worker-Code:

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Threads (Forts.)

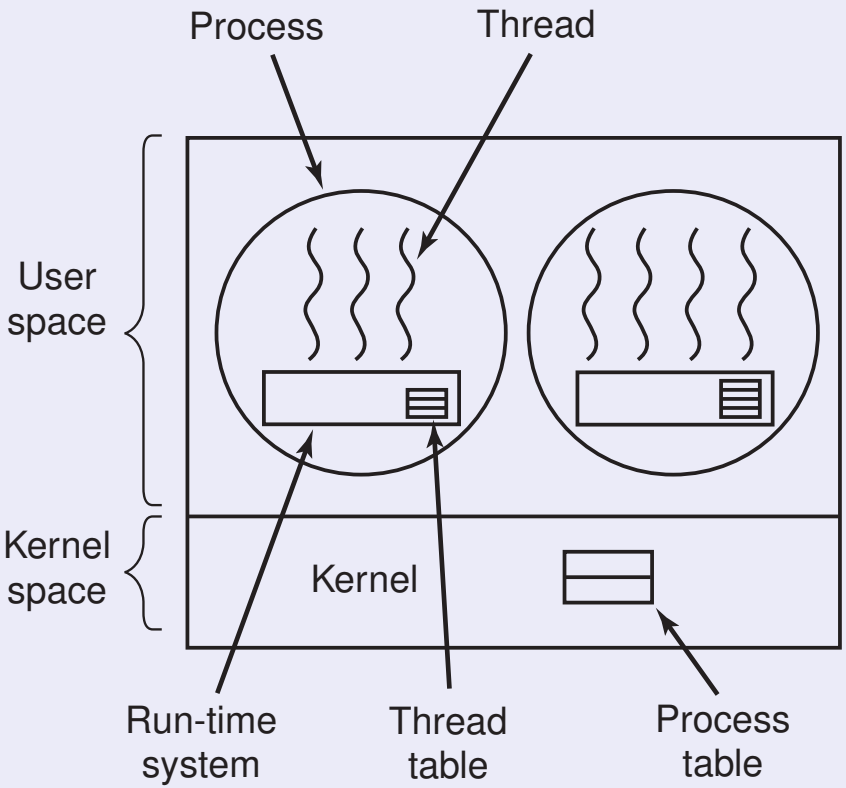
Thread-Bibliotheksfunktionen (in C)

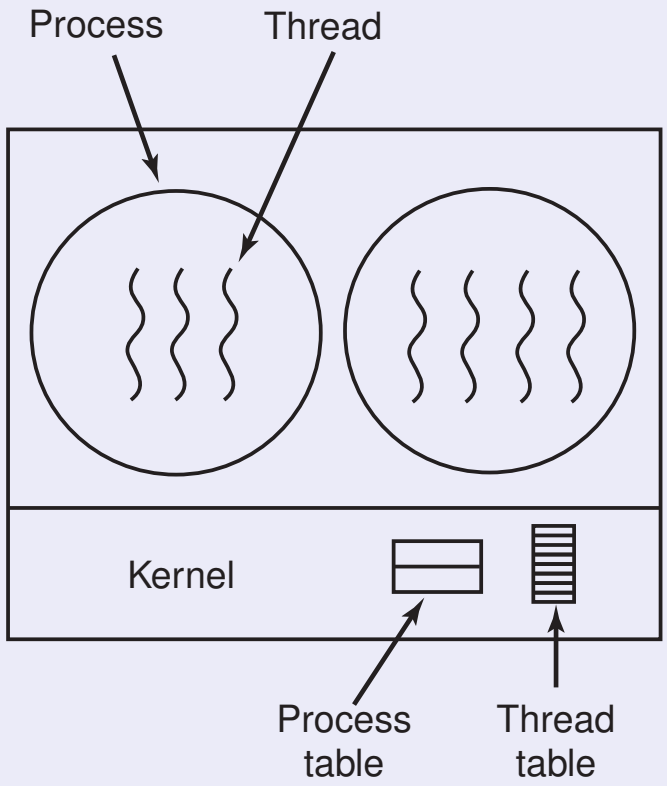
- **thread_create (procedureName)**
  neuen Thread erzeugen
- **thread_exit()**
  Thread beenden
- **thread_wait (threadID)**
  auf Ende des anderen Thread warten
- **thread_yield()**
  Threadwechsel erzwingen
- ...

Threads (Forts.)

Alternative Implementierungen

- im Benutzeradressraum (*User-Level*), *Abb. links*
- im Betriebssystem (*Kernel-Level*), *Abb. rechts*





Threads (Forts.)

User-Level (UL) Threads

- Laufzeitsystem implementiert als Bibliothek: *User-Level Threads Package*
- werden als Teil des User-Prozesses gestartet $\Rightarrow$ im User-Space
- haben eine Thread-Tabelle pro Prozess
- verwenden asynchrone E/A $\Rightarrow$ ggf. Threadwechsel anstatt Prozesswechsel
- UL Threads sind dem BS nicht bekannt

Kernel-Level (KL) Threads

- implementiert vom BS-Kern
- werden im System-Space implementiert
- haben eine einzige Thread-Tabelle für alle Threads
- KL Threads werden vom BS verwaltet ($\Rightarrow$ System Calls)

Threads (Forts.)

Vergleich

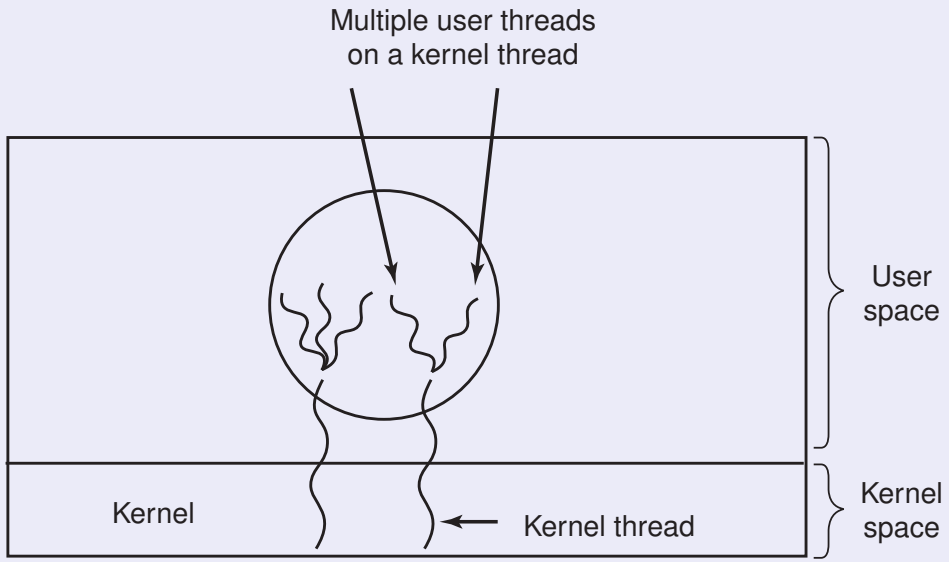
- UL Threads sind einsetzbar mit beliebigen Betriebssystemen, KL Threads müssen im BS implementiert sein
- Operationen auf UL Threads kommen im Unterschied zu KL Threads i. Allg. ohne System-Calls aus $\Rightarrow$ UL Threads sind effizienter
- UL Threads verwenden keinen BS-Speicher $\Rightarrow$ es können viele erzeugt werden
- UL Threads können pro Instanz unterschiedliche Scheduling-Verfahren einsetzen, bei KL Threads definiert BS eine globale Scheduling-Strategie
- blockierende Aktionen bei UL Threads blockieren den Prozess mit allen Threads (z. B. Seitenfehler), bei KL Threads werden nur die jeweiligen Threads blockiert
- UL Threads eines Prozesses laufen immer alle auf demselben Prozessor, KL Threads können ggf. parallel auf verschiedenen Prozessoren laufen

Threads (Forts.)

Arbeiten mit KL Threads

- Thread-Funktionen sind als System-Calls implementiert
  $\Rightarrow$ bspw. ist das Erzeugen eines Thread eine teure Operation
- **Besser:** Threads nicht immer beenden und neu erzeugen sondern *recyceln*: bestehenden Thread mit neuer Aufgabe betrauen anstatt zu beenden

Hybrid-Ansatz



Welche Vorteile hat das? Wo sind hier die Prozesse?

Threads (Forts.)

Worker-Thread-Pool

- Der *Dispatcher* erzeugt zu Beginn einen *Pool* mit einer bestimmten Anzahl von *Worker-Threads*.
- Jeder neue Auftrag wird vom nächsten freien *Worker-Thread* bearbeitet.
- Anschließend liefert dieser das Ergebnis zurück und wartet danach auf einen neuen Auftrag.
- Vorteile:
  - fixer Aufwand für die Thread-Erzeugung nur beim Start $\Rightarrow$ guter Durchsatz
  - Anzahl der Worker-Threads ist dynamisch an die Anfragelast anpassbar
  - Priorisierung der Aufträge leicht möglich
- Implementierung:
  - gemeinsame Queue des Dispatchers und der Worker-Threads für die Aufträge ($\Rightarrow$ E/V-Problem)

Threads in Java

Die Klasse *Thread*

- Threads in Java sind heute *User-Level Threads*, die auf mehrere *Kernel-Level Threads* abgebildet werden
- Wichtige Methoden:
  - `public void run()`
    Methodenrumpf enthält den auszuführenden Code
  - `public void start()`
    startet den Thread und ruft `run()` auf
  - `public static void sleep( int millisecs )`
    versetzt den aufrufenden Thread für die angegebene Zeit in den Schlafzustand
  - `public void join( int millisecs )`
    blockiert den Aufrufer bis der angegebene Thread fertig ist oder max. für die angegebene Zeit
  - `public static void yield()`
    der aufrufende Thread gibt die Prozessorkontrolle ab
    $\Rightarrow$ Thread-Wechsel möglich
  - `public static void interrupt()`
    deblockiert einen bisher blockierten Thread

Threads in Java (Code-Beispiel)

```
public class MyThread1 extends Thread {
  private int id;     // Thread ID
  public MyThread1( int id ){
      this.id = id;
  } // Konstruktor

  public void run() {
      try {
          Thread.sleep( (int) ( Math.random() * 1000 ) );
      } catch (Exception e) {}
      System.out.println( "Hello World (ID = "+ id + ")" );
  }  // run

  public static void main( String[] args ) {
      for (int i=1; i<10; i++){
          Thread t = new MyThread1( i );
          t.start();
      }
  } // main
} //class MyThread1
```

Threads in Java (2. Code-Beispiel)

```
public class MyThread2 implements Runnable {
  private int id;
  public MyThread2( int id ){
      this.id = id;
  }
  public void run() {
      try {
          Thread.sleep( (int) ( Math.random() * 1000 ) );
      } catch (Exception e) {}
      System.out.println( "Hello World (ID = "+ id + ")" );
  } // run

  public static void main( String[] args ) {
      for (int i=1; i<10; i++){
          MyThread2 myRunnable = new MyThread2(i);
          Thread t = new Thread(myRunnable);
          t.start();
      }
  } // main
} //class MyThread2
```

Threads in Java (Forts.)

Vorteil der Verwendung der `Runnable`-Schnittstelle

- in Java gibt es keine Mehrfachvererbung
  $\Rightarrow$ Erben von einer anderen Klasse ist nun immer noch möglich
- davon abgesehen sind beide Vorgehensweisen gleichwertig

Besonderheiten

- das Verhalten eines Programms mit Threads ist – wegen der Kernel-Level-Threads – leider nicht plattformunabhängig!
  - VM für Unix:
    verdrängendes prioritätenbasiertes Scheduling
  - VM für Windows:
    zeitscheibenbasiertes Scheduling

    FRAGE: Welchen Unterschied macht das?

  $\Rightarrow$ Anwendungen sollten auf jeder Plattform fehlerfrei ablaufen!

Speicherbasierte Synchronisation

Wiederkehrende Synchronisationsprobleme

Betriebsmittelverwaltung

- Eine begrenzte Anzahl an identischen Betriebsmitteln wird von mehreren Prozessen oder Threads nebenläufig verwendet
- sind alle Betriebsmittel belegt, müssen die Prozesse bzw. Threads warten

Erzeuger-Verbraucher-Problem

- Verbraucherprozesse oder -threads verarbeiten Daten, die von Erzeugerprozessen oder -threads über einen Puffer bereitgestellt werden
- liegen keine Daten vor, müssen die Verbraucher warten
- ggf. müssen die Erzeuger warten, wenn die Speicherkapazität des Puffers erschöpft ist

Leser-Schreiber-Problem

- Lesende und schreibende Prozesse oder Threads greifen nebenläufig auf die dieselben Daten zu
- ausschließlich Leser dürfen i. Allg. nebenläufig arbeiten

Begriffe

Kritische Daten

- Daten, die von Prozessen oder Threads gemeinsam benutzt werden
- nebenläufige Zugriffe gefährden i. Allg. die Datenkonsistenz

Kritischer Abschnitt (*Critical Section*, *Critical Region*)

- Code-Abschnitt, in dem auf die kritischen Daten zugegriffen wird
- zu einer bestimmten Menge kritischer Daten gehören ein oder mehrere kritische Abschnitte
- kritische Abschnitte dürfen i. Allg. nicht in beliebiger zeitlicher Überlappung ausgeführt werden
- die Ausführung der kritischen Abschnitte ist nur unter Einhaltung gewisser *Synchronisationsbedingungen* erlaubt
  $\Rightarrow$ Konsistenz der kritischen Daten muss gewährleistet sein

Begriffe (Forts.)

Deadlock (Verklemmung)

- Zyklische Wartesituation, bei der mehrere Prozesse bzw. Threads wechselseitig aufeinander warten
- Da keiner in seiner Verarbeitung fortfahren kann, kommt das sich in Ausführung befindliche Programm zum Stillstand
- Tritt ein Deadlock im Betriebssystemkern auf, lässt sich das System ggf. nicht mehr kontrollieren

Lösungsidee

- Das System führt einen *Wartegraphen*
- Ein Deadlock lässt sich durch Auftreten eines *Zyklus* im Wartegraphen entdecken
- Das zwangsweise Beenden oder Zurücksetzen eines oder mehrerer der Wartenden, zerstört den Zyklus und beseitigt dadurch den Deadlock
- Alternativ lassen sich pessimistische Verfahren einsetzen, die Zyklen gar nicht erst zulassen, in einigen Fällen aber zu restriktiv vorgehen

Begriffe (Forts.)

Klassen von Ressourcen und Lösungstrategien

(1) Systemstrukturen (*Prozesskontrollblöcke*, ...)
⇒ *resource ordering*

(2) Speicherplatz im Benutzeradressraum
⇒ längerfristiges Zurückstellen des Prozesses (*swap out*)

(3) Job Ressourcen (Dateien, Geräte)
⇒ Vermeidung (z. B. *Banker's Algorithm*)

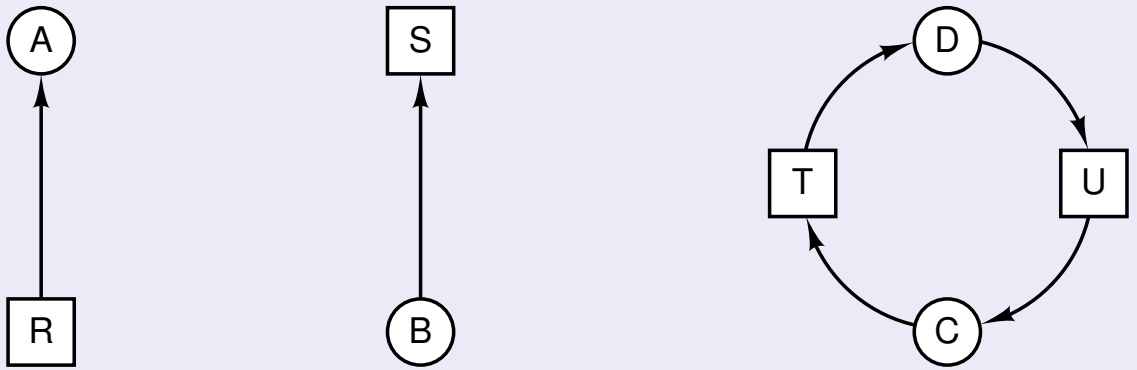
(4) Speicherplatz im Swap Space
⇒ Vollständige Anforderung (z. B. *two phase locking*)

Deadlocks

Erstellen eines Wartegraphen

- Prozesse bzw. Threads als Kreise
- Betriebsmittel als Rechtecke
- Kanten für Belegungen und Wartesituationen

Beispiele für Wartegraphen



(a) (b) (c)

(a) Prozess ***A*** hat Betriebsmittel ***R*** belegt

(b) Thread ***B*** wartet auf Betriebsmittel ***S***

(c) Zyklische Wartesituation: ***D*** wartet auf ***U***, ***C*** wartet auf ***T***

Deadlocks (Forts.)

Bedingungen für das Auftreten von Deadlocks

B1 *Exklusivität* (*mutual exclusion condition*)
Mindestens zwei Betriebsmittel können nur exklusiv benutzt werden

B2 *Nachforderung* (*wait for condition*)
Prozesse, die schon Betriebsmittel belegt haben, können weitere Betriebsmittel anfordern

B3 *Nichtentziehbarkeit* (*no preemption condition*)
Die belegten Betriebsmittel können den Prozessen nicht entzogen werden

Ein Deadlock liegt vor, falls folgende Situation auftritt

B4 *Zyklisches Warten* (*circular wait condition*)
Es gibt eine Folge von P_0 , ... , P_{n-1} derart, dass für $i = 0, \ldots, n\text{-}1$ gilt:
P_i hat ein Betriebsmittel angefordert, das $P_{(i+1) \bmod n}$ belegt hat.

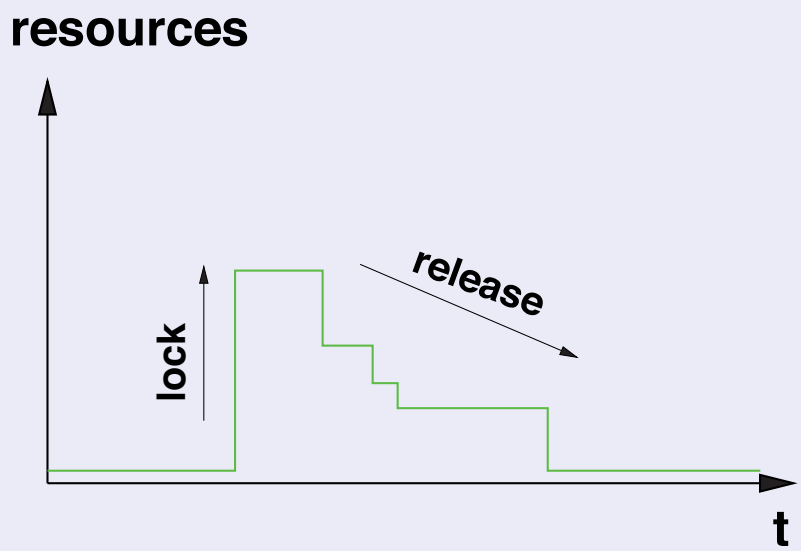
Deadlocks (Forts.)

Möglichkeiten zur Verhinderung

- eine der Voraussetzungen B1, B2 oder B3 wird aufgehoben
- Betriebsmittelverteilung erfolgt nach bestimmten Regeln, so dass B4 nicht eintreten kann

Statische Zuteilung

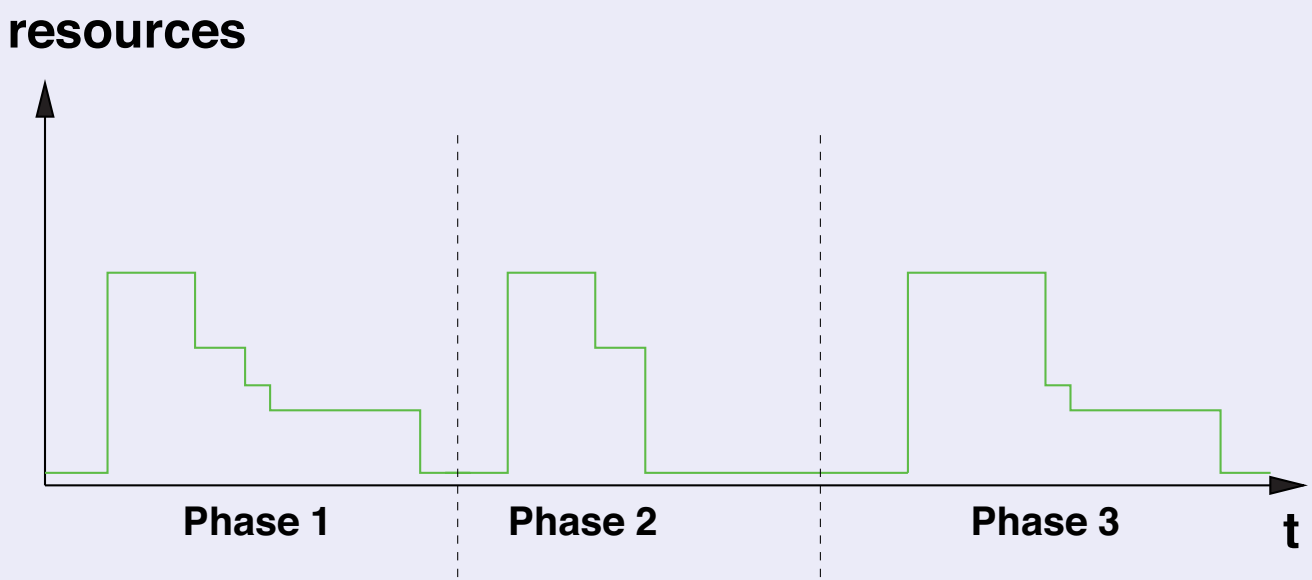
- Prozesse (Threads) melden vollständigen Betriebsmittelbedarf im Voraus an
- ein Prozess (Thread) läuft erst bei Verfügbarkeit aller Betriebsmittel los
  $\Rightarrow$ B2 ist aufgehoben
- Beispiel:Two-Phase-Locking



Deadlocks (Forts.)

Dynamische Zuteilung

- jeder Auftrag besteht aus *Laufphasen* (*job steps*)
- für jede Laufphase erfolgt eine statische Zuteilung
- Freigabe aller Betriebsmittel am Ende einer Laufphase
- jede Laufphase wird wie ein neuer Auftrag angesehen
  $\Rightarrow$ B2 ist aufgehoben



Deadlocks (Forts.)

Zuteilungsprioritäten (*resource ordering*)

- jede Ressource hat eine feste Zuteilungspriorität
- Prozess (Thread) erhält Ressource nicht, wenn er bereits Ressourcen höherer Priorität besitzt
  $\Rightarrow$ B2 ist derart eingeschränkt, dass B4 nicht eintreten kann
- Beispiel:

Ressource	Priorität
Band	3
Platte	2
Drucker	1

PROZESS 1:	PROZESS 2:
...	...
Platte $\rightarrow$ zuteilen	Band $\rightarrow$ zuteilen
...	...
Band $\rightarrow$ warten	Platte $\rightarrow$ abweisen

Prozess 2 läuft (hoffentlich) weiter und gibt Band irgendwann wieder frei

Deadlocks (Forts.)

Spooling

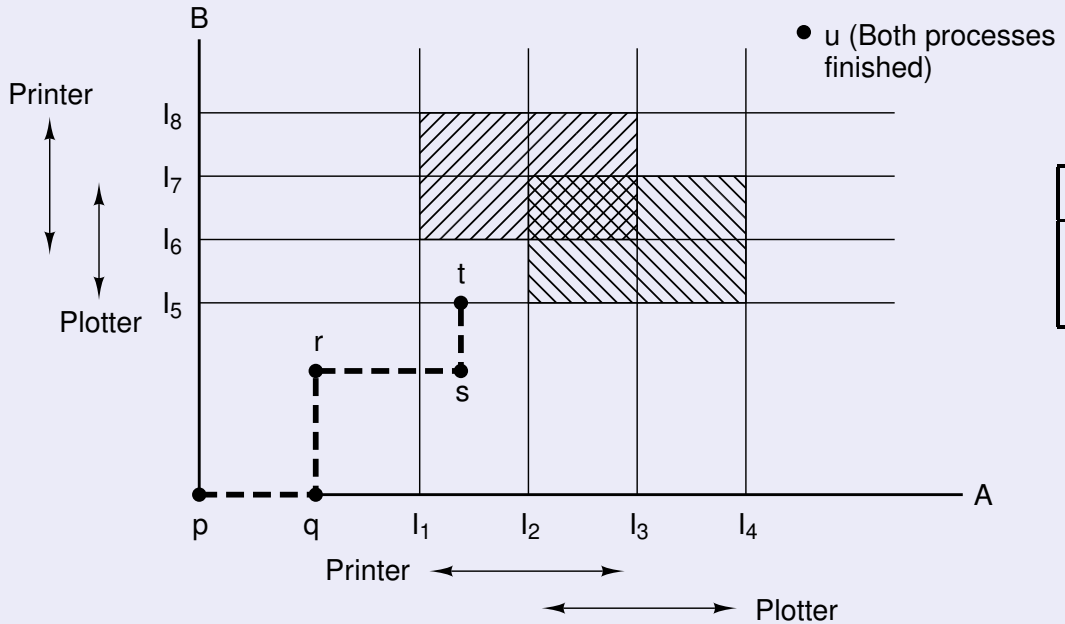
- ein Server-Prozess nimmt eingehende Aufträge für genau eine Ressource entgegen, puffert sie und teilt sukzessive Ressourcen zu
  $\Rightarrow$ B1 ist aufgehoben

- Nachteile
  - Prozesse können keine Gleichzeitigkeit der Belegung von Ressourcen forcieren
  - Spooling nicht für alle Arten von Ressourcen einsetzbar (z. B. Semaphore)
  - Spooling kann selbst Verklemmungen hervorrufen
    (z. B. Speicherplatz reicht nur aus, um genau einen Auftrag zu puffern; Freigabe erst nach Erledigung ...)

Deadlocks (Forts.)

Vermeidung von Deadlocks – Prozessfortschrittsdiagramm

- Annahme: Bedingungen B1, B2, B3 sind erfüllt
- Idee
  - prüfe vor jeder Ressourcenzuteilung, ob diese zu B4 führen kann
    $\Rightarrow$ schraffierte Bereiche dürfen nicht betreten werden!
  - falls ja: Anforderung nicht erfüllen, sondern Prozess in Warteschlange einreihen



Zeitpunkt	
s	Prozess A hat Printer
t	Prozess B will Plotter

Zum Zeitpunkt ***t*** ist kein Weg um schraffierte Bereiche herum mehr zu ***u*** möglich
$\Rightarrow$ Die Anforderung von Prozess ***B*** darf nicht erfüllt werden!

Deadlocks (Forts.)

Banker's Algorithm (Dijkstra, 1965)

- Annahmen
  - es existieren $k \geq 1$ unterschiedliche Betriebsmitteltypen
  - vom Typ i gibt es n_i gleichwertige Exemplare
  - jeder gibt im Voraus für jeden Typ seine *Maximalanforderung* bekannt
  - jeder fordert seine Betriebsmittel *sukzessive* an, jedoch niemals mehr als seine Maximalanforderung
- der aus der Erfüllung einer Anforderung resultierende Zustand ist
  - *sicher*, wenn es immer noch (mindestens) eine Prozessausführungsreihenfolge gibt, so dass alle Prozesse zum Ende kommen können
  - *unsicher*, wenn es für den Fall, dass alle Prozesse ihre Maximalanforderungen stellen, keine *Terminierungsmöglichkeit* mehr gibt $\Rightarrow$ Deadlock!
- unsicher bedeutet nicht, dass ein Deadlock tatsächlich auftreten muss!

Algorithmus

- Erfülle nur diejenigen Anforderungen bzw. Nachforderungen, die zu einem sicheren Zustand führen

Deadlocks (Forts.)

Beispiel mit $k = 10$ Exemplaren einer Ressource

- sicherer Zustand:

Prozess	max. Anforderung	belegt	nachgefordert
1	4	2	2
2	6	3	3
3	8	2	6

$\Rightarrow$ mögliche Ausführungsreihenfolgen: (1,2,3) (2,1,3) (2,3,1)

- unsicherer Zustand:

Prozess	max. Anforderung	belegt	nachgefordert
1	4	–	4
2	6	4	2
3	8	6	–

$\Rightarrow$ Kein Deadlock: P3 terminiert – ohne weitere Nachforderungen

- Deadlock:

Prozess	max. Anforderung	belegt	nachgefordert
1	4	–	2
2	6	4	2
3	8	6	2

Deadlocks (Forts.)

Allgemeiner Banker's Algorithm

- Annahmen
  - Vektor ***Exist***: Anzahl aller gleichartigen Ressourcen
  - Vektor ***Avail***: Anzahl der noch nicht zugeteilten Ressourcen
  - Matrix ***assigned***(P_i): Anzahl der bereits an Prozess P_i zugeteilten Ressourcen
  - Matrix ***needed***(P_i): Anzahl der von Prozess P_i noch max. benötigten Ressourcen

- Algorithmus zum Bewerten der Zustände

(1) Bestimme Prozess P_i mit ***needed***$(P_i) \leq$ ***Avail***

(2) Falls kein solcher Prozess existiert $\Rightarrow$ unsicherer Zustand; Ende

(3) Beende P_i und gib dessen belegte Ressourcen frei:
$$\textbf{\textit{Avail}} = \textbf{\textit{Avail}} + \textbf{\textit{assigned}}(P_i)$$

(4) Falls es noch laufende Prozesse gibt $\Rightarrow$ zurück zu (1)

(5) Sicherer Zustand; Ende

Deadlocks (Forts.)

Beispiel

assigned(P_i):

Prozess	Band	Scanner	Drucker	CD
P_1:	3	0	1	1
P_2:	0	1	0	0
P_3:	1	1	1	0
P_4:	1	1	0	1
P_5:	0	0	0	0

needed(P_i):

Prozess	Band	Scanner	Drucker	CD
P_1:	1	1	0	0
P_2:	0	1	1	2
P_3:	3	1	0	0
P_4:	0	0	1	0
P_5:	2	1	1	0

Exist $= (6, 3, 4, 2)$

Avail $= (1, 0, 2, 0)$

(a) Zustand ist derzeit sicher
$\Rightarrow$ mögliche Reihenfolge: $P_4 \Rightarrow P_5 \Rightarrow P_1 \Rightarrow P_3 \Rightarrow P_2$

(b) P_2 fordert nun 1 Drucker an $\Rightarrow$ sicher
$\Rightarrow$ nach Zuteilung: $P_4 \Rightarrow P_5 \Rightarrow P_1 \Rightarrow P_3 \Rightarrow P_2$

(c) P_5 fordert anschließend 1 Drucker an $\Rightarrow$ unsicher!
$\Rightarrow$ Falls keine Ressourcen mehr freigegeben werden, kann kein Prozess seine Maximalanforderung noch erfüllen

Beispiele für Synchronisationsbedingungen

Gegenseitiger Ausschluss (*Mutual Exclusion*)

- wenn sich ein kritischer Abschnitt in Ausführung befindet, darf keiner der anderen (ausgeschlossenen) kritischen Abschnitte betreten werden

Reihenfolgebedingungen

- die kritischen Abschnitte dürfen nur in einer bestimmten Reihenfolge durchlaufen werden
- Beispiel: Daten müssen zuerst generiert werden, bevor darauf zugegriffen werden kann

Graphische Darstellung von Synchronisationsbedingungen

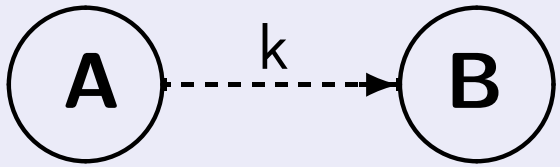
Synchronisationsgraph: Ausschluss

- Knoten: kritischer Abschnitt
- Kante: Relation *schließt aus* (einseitiger Ausschluss)
  - solange `A` in Ausführung ist, darf `B` nicht betreten werden
  - Folge:
    Prozesse bzw. Threads, die `B` aufrufen, müssen warten, bis `A` frei ist

Frage: Wie stellt man *gegenseitigen Ausschluss* dar?

Graphische Darstellung (Forts.)

Synchronisationsgraph: Reihenfolge



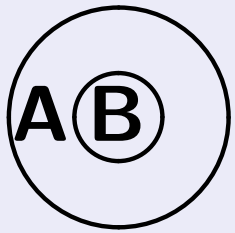
- gestrichelte Kante: Relation *k-folgt*
  - gelesen als: `B` *k-folgt* `A` (mit $\boldsymbol{k} \geq 0$)
  - entspricht dem Synchronisationstyp der *Bedingungssynchronisation*
- für jeden kritischen Abschnitt definieren wir zwei Zähler
  - ***Anf***: Anzahl der *angefangenen Ausführungen* des kritischen Abschnitts
  - ***End***: Anzahl der *beendeten Ausführungen* des kritischen Abschnitts
- Die Synchronisation stellt sicher, dass zu jedem Zeitpunkt die nachfolgende Bedingung eingehalten wird:

$$\boldsymbol{Anf}(\mathtt{B}) \leq \boldsymbol{End}(\mathtt{A}) + \boldsymbol{k}$$

- Der Anfang von `B` wird solange verzögert, bis die Bedingung eingehalten werden kann.

Graphische Darstellung (Forts.)

Synchronisationsgraph: geschachtelte kritische Abschnitte



- *geschachtelte* kritische Abschnitte $\Rightarrow$ `B` liegt innerhalb von `A`
- es gilt folgende Regel:
  bevor `B` betreten werden kann, muss `A` durchlaufen werden, aber nicht umgekehrt

Basismechanismen

Wdh.: Unterbrechungssperren

- Klammern des kritischen Abschnitts mittels `itrs_off` ... `itrs_on`
  $\Rightarrow$ Unterbrechungen werden dazwischen verboten
- bei Einprozessorsystemen mit Round-Robin Scheduling korrekt
- Prozess / Thread ist nicht unterbrechbar $\Rightarrow$ falls Prozessor nicht freiwillig abgegeben wird, wird der kritische Abschnitt *atomar* ausgeführt
- Nachteile:
  - bei Multiprozessorsystemen nicht ausreichend
  - herabgesetzte Reaktionsfähigkeit auf externe Unterbrechungen
    $\Rightarrow$ möglicher Verlust von Daten bei E/A-Operationen
  - Unterbrechungen können nur im *privilegierten Zustand* ausgeschlossen werden
    $\Rightarrow$ privilegierter Maschinenbefehl, System-Mode notwendig
  - kein System-Call hierfür, da im User-Mode ungeeignet
    $\Rightarrow$ Programmierer könnte vergessen, die Unterbrechungen wieder zuzulassen
- Einsatzgebiet:
  - bei Einprozessorsystemen für die Realisierung von zeitlich kurzen kritischen Abschnitten innerhalb des Betriebssystemkerns

Basismechanismen (Forts.)

Wdh.: Atomare Speicheroperationen

- Das Abspeichern eines Wertes in den Hauptspeicher erfolgt *atomar*
- falls mehrere Prozessoren zeitgleich versuchen, je einen Wert in dasselbe Wort zu speichern, entscheidet die Hardware, welche Speicheroperation zuerst ausgeführt wird $\Rightarrow$ Busarbitrierung

Spezielle Hardware-Befehle

- moderne Mikroprozessoren besitzen einen oder mehrere spezielle Maschinenbefehle, z. B.
  - `TSL`: *Test and Set Lock*: Das Lesen des momentanen Wertes und nachfolgende Schreiben eines Speicherwortes auf den Wert 1 werden *atomar* durchgeführt
  - `SWAP`: die Inhalte zweier Speicherworte werden *atomar* vertauscht

Basismechanismen (Forts.)

Spin-Lock mittels TSL (Pseudocode)

```
public class MutualExclusionTSL {
  // Eintrittsprotokoll
  public static void enterMutex ( Integer busy) {
  // busy should initially be set to 0
    Integer local;
    do
       local = TSL (busy);
    while ( local = = 1 );
  }

  // Austrittsprotokoll
  public static void exitMutex (Integer busy) {
    busy = 0;
  }
} // MutualExclusionTSL
```

Basismechanismen (Forts.)

Spin-Lock mittels TSL

- Vorteile
  - Spin Lock funktioniert bei Multiprozessorsystemen
  - durch Parametrisierung mit verschiedenen globalen Variablen kann der gegenseitige Ausschluss auf bestimmte kritische Abschnitte eingeschränkt werden
- Nachteile
  - *Busy Waiting*
  - nicht fair $\Rightarrow$ *Starvation* möglich
  - während des kritischen Abschnitts sollte wegen der Verklemmungsgefahr kein Prozess- bzw. Threadwechsel erfolgen
    $\Rightarrow$ Unterbrechungen während der Ausführung verbieten
- Einsatzgebiet
  - Realisierung des gegenseitigen Ausschlusses kurzer kritischer Abschnitte bei Multiprozessorsystemen im Betriebssystemkern
  - Impementierung mächtigerer Synchronisationsmechanismen für längere kritische Abschnitte auch im User-Space

Das Semaphor-Konzept

Semaphor-Definition

- eingeführt von *Dijkstra*
- Objekt, auf dem genau 2 *atomare* Operationen `p` und `v` existieren
- interne Komponenten eines Semaphors `sem`
  - Wert des Semaphors: `sem.ctr`
  - Warteschlange für Prozesse: `sem.queue`
- Wirkungsweise der Operationen
  - `p`: ctr−−; if (ctr $<$ 0) { *warten;* }
  - `v`: ctr++; if (ctr $\leq$ 0) { *einen Wartenden aufwecken;* }
- Der Wert des Semaphors lässt sich wie folgt interpretieren
  - falls positiv: verbleibende Anzahl der Aufrufe von `p` ohne warten zu müssen
  - falls negativ: Betrag entspricht der Anzahl der am Semaphor Wartenden

Gegenseitiger Ausschluss

- Es wird ein Semaphor mit dem Startwert 1 verwendet und der kritische Abschnitt wird mittels `p` und `v` umschlossen

Exkurs: Exception Handling in Java

Wie geht's?

- Wenn im Programmablauf etwas schief geht, können *Exceptions* (Ausnahmen) erzeugt werden, die den normalen (fehlerfreien) Programmfluss unterbrechen.
- Eigene Exceptions definieren und werfen

```
throw new IOException ("Heute nicht - ich hab Kopfweh!");
```

- Exceptions müssen von einer Methode an den Aufrufer weitergereicht werden, d. h. sie werden im Interface der entsprechenden Methode deklariert

```
public method( ... ) throws exceptionType1, exceptionType2,...
 {
 ...
 }
```

Exception Handling (Forts.)

Catching Exceptions

Spätestens im Hauptprogramm sollten Exceptions behandelt (*gefangen*) werden

```
try {
        ... // Programmcode ohne
        ... // Fehlerbehandlung
} // end try

catch (exceptionType1) {
    ... // Aktionen
}
:
:
catch (exceptionTypeN) {
    ... // Aktionen
}
finally {
    ... // wird IMMER ausgefuehrt, sogar bei vorherigem return
}
```

Exception Handling (Forts.)

Testausgaben

- *catch*-Blöcke sollten i. d. R. nicht leer bleiben, sondern es sollte zum Testen wenigstens der Text der Exception (am besten auf der Standard-Fehlerausgabe *System.err*) ausgegeben werden

```
... catch (Exception e) { System.err.println(e); }
```

oder

```
... catch (Exception e) { e.printStackTrace(); }
```

Das Semaphor-Konzept (Forts.)

Semaphore in *Java*

- Klasse `Semaphore` enthalten in `java.util.concurrent.Semaphore`
- Semaphore wirken hier auf Threads
- Konstruktur muss Startwert festlegen:
  `public Semaphore (int initValue, true)`
  `true` wird hier angegeben, um FIFO-Reihenfolge zu gewährleisten
- `p` wird implementiert durch die Methode `acquire()`
- `v` wird implementiert durch die Methode `release()`
- `acquire()` und `release()` erzeugen ggf. eine `InterruptedException`
  $\Rightarrow$ Exception Handling !

Das Semaphor-Konzept (Forts.)

Java-Beispiel für gegenseitigen Ausschluss

```
import java.util.concurrent.Semaphore;

public class MutexSem {
  static Semaphore mutex = new Semaphore( 1, true ); // true => FIFO
  static Runnable r = new Runnable() {
    public void run() {
      while ( true ) {
        try {
          mutex.acquire();
          // k.A. beginnt
          System.out.println( Thread.currentThread().getName() + " im k.A."  );
          Thread.sleep(3000);
          // k.A. endet
          mutex.release();
        } catch ( InterruptedException e ) {
          e.printStackTrace();
          Thread.currentThread().interrupt();
          break;
        } catch ( Exception e ) {
          e.printStackTrace();
        }
      } // while
    } // run
  }; // Runnable

  public static void main( String[] args ) {
    new Thread( r ).start();
    new Thread( r ).start();
    new Thread( r ).start();
  } // main
} // class
```

Erzeuger-Verbraucher-Problem

Lösung des einfachen Erzeuger-Verbraucher-Problems

- hierbei gibt es nur 1 Erzeuger und 1 Verbraucher
- der Erzeuger legt die generierten Daten in einem Puffer ab
  oft wird hierbei ein Ringpuffer (*bounded buffer*) verwendet:
  - ist das Ende des Puffers erreicht, werden die folgenden Daten wieder am Anfang abgelegt, falls dort wieder frei ist
  - ist der Puffer voll, muss gewartet werden
- der Verbraucher entnimmt nacheinander die im Puffer abgelegten Datenelemente
  - ist der Puffer leer, muss gewartet werden
- benötigte Datenstrukturen:
  - ein Array der Größe ***MAX*** als Puffer
  - ein Zeiger auf des nächste leere Pufferelement $\Rightarrow$ `nextfree`
  - ein Zeiger auf des nächste volle Pufferelement $\Rightarrow$ `nextfull`
  - ein Zähler, der mitzählt, wieviele Datenelemente der Puffer enthält $\Rightarrow$ `ctr`
    $\Rightarrow$ Erzeuger und Verbraucher dürfen `ctr` nicht nebenläufig bearbeiten!

Erzeuger-Verbraucher-Problem (Forts.)

Lösung des einfachen Erzeuger-Verbraucher-Problems (Forts.)

- Erzeugen (***E***) und Verbrauchen (***V***) bilden jeweils einen kritischen Abschnitt
- Synchronisationsbedingungen:
  - ***E*** und ***V*** müssen unter gegenseitigem Ausschluss ausgeführt werden
  - es können nur Daten verbraucht werden, die zuvor erzeugt wurden
  - wenn der Puffer voll ist, müssen zuerst Daten verbraucht werden

Frage: Wie sieht der Synchronisationsgraph hierfür aus?

- Das nachfolgend dargestellte Java-Programm zeigt eine mögliche Lösung

Erzeuger-Verbraucher-Problem (Forts.)

Lösung des einfachen Erzeuger-Verbraucher-Problems (Forts.)

```
import java.util.concurrent.Semaphore;

public class BB1 {
        private int size;
        private String[] buffer;
        private int ctr = 0;
        private int nextfree = 0;
        private int nextfull = 0;
        private Semaphore mutex = new Semaphore( 1, true );
        private Semaphore full = new Semaphore (0, true);
        private Semaphore empty;

        public BB1 (int size){
                this.size = size;
                buffer = new String[size];
                empty = new Semaphore (size, true);
        }

        public void append (String data){
                try{
                        System.out.println("Prod arriving");
                        empty.acquire();
                        mutex.acquire();
                        System.out.println("Prod active with "+ data);
                        buffer[nextfree] = data;
                        nextfree = (nextfree+1) % size;
                        ctr++;
                        mutex.release();
                        full.release();
                        System.out.println("Prod gone");
                } catch (InterruptedException e){e.printStackTrace();}
        } // append
```

Das Semaphor-Konzept (Forts.)

Lösung des einfachen Erzeuger-Verbraucher-Problems (Forts.)

```
        public String remove (){
                String data="";
                try{
                        System.out.println("                    Cons arriving");
                        full.acquire();
                        mutex.acquire();
                        System.out.println("                    Cons active");
                        data = buffer[nextfull];
                        nextfull = (nextfull+1) % size;
                        ctr--;
                        mutex.release();
                        empty.release();
                        System.out.println("                    Cons gone with " + data);
                } catch (InterruptedException e){e.printStackTrace();}
                return data;

        } // remove


        public static void main( String[] args ) {
                BB1 bb = new BB1(5);
                new Thread( new Cons(bb) ).start();
                new Thread( new Prod(bb) ).start();
        } // main
} // class
```

Das Semaphor-Konzept (Forts.)

Lösung des einfachen Erzeuger-Verbraucher-Problems (Forts.)

```
class Cons implements Runnable {
        BB1 bb;
        public Cons(BB1 bb){
                this.bb = bb;
        }

        public void run() {
                String data;
                while (true) {
                        data = bb.remove();
                }

        } // run
} // class


class Prod implements Runnable {
        BB1 bb;
        public Prod(BB1 bb){
                this.bb = bb;
        }

        public void run() {
                for (int i=0; i<100; i++) {
                        bb.append ("Data"+i);
                } // for
        } // run
} // class
```

Erzeuger-Verbraucher-Problem (Forts.)

Lösung des einfachen Erzeuger-Verbraucher-Problems (Forts.)

- Beobachtung beim Ringpuffer:
  - ***E*** und ***V*** greifen nie gleichzeitig auf dasselbe Pufferelement zu:
  - `nextfree == nextfull` gilt nur, wenn Puffer leer oder voll ist
    dann muss aber entweder ***V*** oder ***E*** warten $\Rightarrow$ keine Nebenläufigkeit
    $\Rightarrow$ `mutex`-Semaphor kann eingespart werden, falls wir `ctr` einfach weglassen!

Lösung des allgemeinen Erzeuger-Verbraucher-Problems

- mehrere Erzeuger und mehrere Verbraucher
- Folge: wir brauchen jetzt jeweils ein eigenes `mutex`-Semaphor für die Erzeuger und die Verbraucher

Frage: Wie sieht der Synchronisationsgraph nun aus?

Betriebsmittelverwaltung

Problemstellung

- zu einer Konfiguration gehören `n` identische Drucker, die hier die Rolle der Betriebsmittel spielen
- welcher Drucker verwendet wird, ist einem Prozess egal

Lösung mit Semaphoren

Wir verwenden ein Semaphor mit Startwert `n`:

```
import java.util.concurrent.Semaphore;

public class Bmv {
        private Semaphore printer;

        public Bmv (int anz) {
                printer = new Semaphore (anz, true);
        }

        public void printFile (File f) {
                try{
                        printer.acquire();
                        f.print();   // AUF WELCHEM DRUCKER???
                        printer.release();
                } catch (InterruptedException e){e.printStackTrace();}
        }
}
```

Betriebsmittelverwaltung (Forts.)

Lösung mit Zuweisung der Drucker

```
public class Bmv {
        int anz = -1;
        private Semaphore printer;
        private Semaphore mutex = new Semaphore (1, true);
        private boolean[] printerFree;

        public Bmv (int anz) {
                this.anz = anz;
                printer = new Semaphore (anz, true);
                for (int i=0; i < anz; i++)
                        printerFree[i] = true;
        }

        public void printFile (File f) {
                int usePrinterNo = -1;
                try{
                        printer.acquire();
                        mutex.acquire();
                        for (int i=0; i < anz; i++)
                                if (printerFree[i]) {
                                        usePrinterNo = i;
                                        printerFree[usePrinterNo] = false;
                                        break;
                                }
                        mutex.release();
                        f.printOnPrinter(usePrinterNo);
                        mutex.acquire();
                        printerFree[usePrinterNo] = true;
                        mutex.release();
                        printer.release();
                } catch (InterruptedException e){e.printStackTrace();}
        } // printFile
} // class
```

Leser-Schreiber-Problem

Problemstellung

- Leseoperationen `read` verwenden einen Datensatz ohne diesen zu verändern
- Schreiboperationen `write` verändern diesen Datensatz an beliebigen Stellen (und lesen ihn ggf.)
  $\Rightarrow$ Leserprozesse sollen vor einer Veränderung der Daten während des Lesevorgangs geschützt werden

Beobachtungen

- nebenläufige Leseoperationen sind i. Allg. unkritisch
- nebenläufige Schreiboperationen können zu inkonsistenten Daten führen
- nebenläufiges Lesen und Schreiben kann zu inkonsistenten Daten führen

Frage: Wie sieht der Synchronisationsgraph hier aus?

Leser-Schreiber-Problem (Forts.)

Lösung des Leser-Schreiber-Problems

```
public class LS {
        private File f;
        private int readcount = 0;  // counting all readers
        private Semaphore mutex = new Semaphore (1, true);
        private Semaphore w = new Semaphore (1, true); // mutex for writers

        public LS ( File f ) {
                this.f = f;
        }

        public void write ( int index, byte[] data ) throws InterruptedException {
                w.acquire();
                f.write ( index, data ); // k.A.
                w.release();
        } // write

        public int read ( int index, byte[] data ) throws InterruptedException {
                mutex.acquire();
                readcount++;
                if ( readcount == 1 ) w.acquire(); // first reader
                mutex.release();

                int byteCount = f.read ( index, data );  // k.A.

                mutex.acquire();
                readcount--;
                if ( readcount == 0 ) w.release(); // last reader
                mutex.release();
                return byteCount;
        } // read
}// class
```

Leser-Schreiber-Problem (Forts.)

Einführung von Prioritäten

- Erstes Leser-Schreiber-Problem: *Leserpriorität*
  - Leser müssen nur dann warten, wenn ein Schreiber schreibt
  - Schreiber müssen warten bis keine Leser mehr lesen wollen
- Zweites Leser-Schreiber-Problem: *Schreiberpriorität*
  - falls Schreiber warten, darf kein Leser mehr mit dem Lesen beginnen
  - ein wartender Schreiber beginnt, wenn der letzte aktive Leser fertig ist

Frage: Wie sehen die beiden Synchronisationsgraphen aus?

Leser-Schreiber-Problem (Forts.)

Leserpriorität

- die gerade gezeigte Lösung gewährleistet nicht die Leserpriorität!

  Gegenbeispiel:
  - Schreiber ist aktiv im kritischen Abschnitt
  - ein weiterer Schreiber und ein Leser blockieren sich in dieser Reihenfolge am Semaphor `w`
  - der wartende Schreiber wird vor dem wartenden Leser aktiviert
    dies widerspricht der Leserpriorität!
  - zur Lösung des Problems muss ein zusätzlicher kritischer Abschnitt *Anmeldung zum Lesen* eingeführt werden

Leser-Schreiber-Problem (Forts.)

Lösung des Ersten Leser-Schreiber-Problems

```
private Semaphore extra = new Semaphore (1, true);

        public void write ( int index, byte[] data ) {
                extra.acquire();
                w.acquire();
                        f.write ( index, data ); // k. A.
                w.release();
                extra.release();
} // write
```

Leserpriorität – Erläuterung des Codes

- die wesentliche Änderung liegt in der Hinzunahme des Semaphors `extra`
- Schreiber müssen jetzt bereits vor `extra` warten
  $\Rightarrow$ Realisierung der Leserpriorität
- die Operation `read` bleibt unverändert

Leser-Schreiber-Problem (Forts.)

Lösung des Zweiten Leser-Schreiber-Problems

```
public class RW2 {
        private File f;
        private int readcount = 0;
        private int writecount = 0;

        private Semaphore mutex1 = new Semaphore (1, true);
        private Semaphore mutex2 = new Semaphore (1, true);
        private Semaphore mutex3 = new Semaphore (1, true);
        private Semaphore w = new Semaphore (1, true);
        private Semaphore r = new Semaphore (1, true);
        // Semaphore r for writer priority

        public RW2 ( File f ) {
                this.f = f;
        }
```

Leser-Schreiber-Problem (Forts.)

Lösung des Zweiten Leser-Schreiber-Problems (Forts.)

```
public int read ( int index, byte[] data )
                          throws InterruptedException {
        mutex3.acquire();
        r.acquire();
        mutex1.acquire();
        readcount = readcount + 1;
        if ( readcount == 1 ) w.acquire();
        mutex1.release();
        r.release();
        mutex3.release();
        int byteCount = f.read ( index, data ); // k. A.
        mutex1.acquire();
        readcount = readcount - 1;
        if ( readcount == 0 ) w.release();
        mutex1.release();
} // read
```

Leser-Schreiber-Problem (Forts.)

Lösung des Zweiten Leser-Schreiber-Problems (Forts.)

```
        public void write ( int index, byte[] data )
                                    throws InterruptedException {
                mutex2.acquire();
                writecount = writecount + 1;
                if ( writecount == 1 ) r.acquire();
                mutex2.release();
                w.acquire();
                f.write ( index, data ); // k. A.
                w.release();
                mutex2.acquire();
                writecount = writecount - 1;
                if ( writecount == 0 ) r.release();
                mutex2.release();
        } // write
} // class
```

Leser-Schreiber-Problem (Forts.)

Schreiberpriorität – Erläuterung des Codes

- `writecount` wird von mehreren Schreibern verwendet
  $\Rightarrow$ `mutex2` erforderlich
- Verwendung des Semaphors `r`, damit sich Schreiber anmelden können:
  - falls ein Schreiber angemeldet ist und Leser noch aktiv sind, werden weitere Leser vor `r` blockiert, damit sie nicht `readcount` erhöhen können
- ohne das Semaphor `mutex3` könnte der folgende Fall eintreten:
  (a) im kritischen Abschnitt befinden sich mehrere aktive Leser;
  es gibt keinen wartenden Schreiber und damit auch keine wartenden Leser
  (b) ein Schreiber `W1` meldet sich an; er passiert das Semaphor `r`
  und wartet vor dem Semaphor `w` $\{\Rightarrow$ `r.ctr=0`$\}$
  (c) zwei weitere Leser `L1` und `L2` kommen hinzu;
  beide warten vor dem Semaphor `r` $\{\Rightarrow$ `r.ctr=-2`$\}$
  (d) nachdem alle Leser aus (a) fertig sind, wird `W1` durch den letzten Leser aktiviert
  (e) nachdem `W1` fertig ist, aktiviert er `L1` $\{\Rightarrow$ `r.ctr=-1`$\}$
  (f) bevor `L1` losläuft, betritt ein neuer Schreiber `W2` den kritischen Abschnitt und
  muss vor dem Semaphor `r` warten $\{\Rightarrow$ `r.ctr=-2`$\}$
  (g) `L1` läuft los und gibt den Leser `L2` frei, obwohl der Schreiber `W2` wartet

Leser-Schreiber-Problem (Forts.)

Schreiberpriorität – Erläuterung des Codes (Forts.)

- Einführung von `mutex3` löst das Problem wie folgt:
  - höchstens ein Leser wartet vor `r`
  - alle weiteren Leser warten vor `mutex3`
  - $\Rightarrow$ in Schritt (c) wartet
    - `L1` vor `r` und
    - `L2` vor `mutex3`
  - $\Rightarrow$ in Schritt (g) wird
    - durch `r.release()` der Schreiber `W2` freigegeben;
    - `W2` wartet dann vor `w`,
    - bis `L1` den kritischen Abschnitt verlassen hat

Private Semaphore

Prioritäten beim Aufwecken

- Semaphore werden in der Regel unter Verwendung einer *FIFO*-Warteschlange implementiert
- manchmal ist jedoch eine andere Aufweckreihenfolge wichtig
- Lösung:
  jeder einzelne Wartende bzw. jede Gruppe von Wartenden wartet an einem eigenen Semaphor $\Rightarrow$ *privates* Semaphor
- Realisierung im Allgemeinen als Array von Semaphoren

- der Test, ob gewartet werden muss, muss i. Allg. unter gegenseitigem Ausschluss augeführt werden
  $\Rightarrow$ um eine Verklemmung zu vermeiden, ist eine komplexere Programmstruktur notwendig

Private Semaphore (Forts.)

Programmstruktur

(1) das private Semaphor wird mit `0` initialisiert

(2) Test vor Beginn des kritischen Abschnitts, ob dieser betreten werden darf

- falls ja $\Rightarrow$ `release` auf dem privaten Semaphor $\{\texttt{ctr} \Rightarrow 1\}$
- falls nein, wird in einer geeigneten Datenstruktur vermerkt, dass der kritische Abschnitt betreten werden soll $\{\texttt{ctr} = 0\}$

(3) nach dem Test wird grundsätzlich ein `acquire` auf dem privaten Semaphor ausgeführt

- war der Test zuvor erfolgreich, kann der kritische Abschnitt betreten werden
- andernfalls wird vor dem privaten Semaphor gewartet

(4) nach Beendigung des kritischen Abschnitts wird überprüft, ob andere vor ihren privaten Semaphoren warten

- falls ja, wird einer (oder mehrere) der Wartenden ausgewählt und auf deren privaten Semaphoren jeweils ein `release` ausgeführt

Private Semaphore (Forts.)

Beispiel: Verwaltung von Druckaufträgen

- mehrere Threads (oder Prozesse) wollen nebenläufig Dateien auf demselben Drucker drucken
- falls der Drucker gerade belegt ist, muss gewartet werden
- nach Beendigung eines Druckauftrags wird derjenige ausgewählt, der den kürzesten Druckauftrag besitzt
- das nachfolgend gezeigte Programm enthält eine Lösung für die Verwaltung der Druckaufträge

  - *ein wesentlicher Nachteil dieser Lösung ist, dass wegen der Verwendung der Thread-IDs die Anzahl der Threads fest vorgegeben ist*

    $\Rightarrow$ *Verbesserungsvorschlag:*
  - *ein Thread übergibt zusätzlich eine Referenz auf sein privates Semaphor als Parameter der* `print`*-Operation*
  - *diese Semaphore werden dann, genauso wie die Dateireferenzen, in einer internen Liste verwaltet*
  - *nach Ende des Druckauftrags wird die Semaphorreferenz wieder gelöscht*

Private Semaphore (Forts.)

Programmcode

```
public class PrintJobManagement {
        private boolean printerFree = true;
        private File[] files; // Verwaltung der auszudruckenden Dateien
        private boolean[] waiting; // Verwaltung, welche Threads warten
        private Semaphore[] privsem; // private Semaphore
        private Semaphore mutex = new Semaphore (1, true); // wegen Verwaltungsdaten
        private static final int NO_ID = -1; // undefinierte Thread-ID

        public PrintJobManagement ( int threadCount ) {
                waiting = new boolean[threadCount];
                privsem = new Semaphore[threadCount];
                files = new File[threadCount];
                for ( int i=0 ; i < threadCount ; i++ ) {
                        waiting[i] = false;
                        files[i] = null;
                        privsem[i] = new Semaphore (0, true);
                }
        } // Konstruktor

        public void print ( File f, int threadId ) {
                mutex.acquire(); // Eintrittsprotokoll
                if ( printerFree ) { // Drucker reservieren
                        printerFree = false;
                        privsem[threadId].release(); // nachher NICHT warten!
                }
                else { // Belegungswunsch eintragen
                        waiting[threadId] = true;
                        files[threadId] = f;
                }
                mutex.release();
                privsem[threadId].acquire(); // ggf. warten

                        // k. A. jetzt Drucken der Datei f
```

Private Semaphore (Forts.)

Programmcode (Forts.)

```
                mutex.acquire(); // Austrittsprotokoll
                printerFree = true; // Drucker wieder freigeben
                files[threadId] = null;
                // suche wartenden Thread mit dem kuerzesten Druckauftrag
                int fileLength = Integer.MAX_VALUE;
                int selectedthread = NO_ID;
                for ( int i=0 ; i < waiting.length ; i++ )
                        if ( waiting[i] && files[i].length() <= fileLength ) {
                                selectedthread = i;
                                fileLength = files[i].length();
                        }
                // falls ein Thread gefunden wurde, wird er nun aufgeweckt
                if ( selectedthread != NO_ID ) {
                        waiting[selectedthread] = false;
                        printerFree = false;
                        privsem[selectedthread].release();
                }
                mutex.release();
        } // print

} // PrintJobManagement
```

Bewertung des Semaphorkonzepts

Vorteile

- Semaphore sind relativ einfache, flexibel nutzbare Objekte
- fast alle Synchronisationsprobleme sind mit Semaphoren lösbar
- gegenseitiger Ausschluss, Betriebsmittelverwaltungs- sowie Erzeuger-Verbraucher-Probleme sind elegant und effizient lösbar

Nachteile

- Korekte Lösungen für komplexe Aufgabenstellungen sind oft sehr schwierig
  $\Rightarrow$ *insb. Aufgabenstellungen, mit Prioritäten bereiten Schwierigkeiten (vgl. Leser-Schreiber-Problem mit Schreiberpriorität)*
- Verstehen und Analysieren der Lösungen ist oft schwierig
- Semaphore haben keine strukturierenden Eigenschaften
  $\Rightarrow$ *mehrere kritische Abschnitte, die logisch einer Datenmenge zugeordnet sind, stehen oft textuell verstreut in den Programmen $\Rightarrow$ unübersichtlich!*
- fehlerhafte Verwendung von Semaphoren kann leicht zu laufzeitabhängigen Fehlern führen, die nicht systematisch reproduzierbar sind!

Was ist ein Monitor ?

Idee

- Ein Monitor dient als komfortables Beschreibungsmittels für komplexere Synchronisationsprobleme
  (parallel publiziert von von *Brinch Hansen* (1973) und *Hoare* (1974))
- Ein Monitor ist dabei eine programmiersprachliche Einheit aus
  - **Monitordaten:** die Menge *kritischer Daten*
  - **Monitoroperationen:** *kritische Abschnitte*, die auf die Monitordaten zugreifen
- Monitoroperationen werden grundsätzlich unter *gegenseitigem Ausschluss* ausgeführt
  $\Rightarrow$ maximal ein Thread kann im Monitor aktiv sein,
  d. h. er hat die *Monitorkontrolle*
- Auf Monitordaten darf nur über die Operationen des Monitors zugegriffen werden
- Das Monitor-Konzept wurde in zahlreiche Programmiersprachen eingebaut

Das Monitorkonzept (Forts.)

Allgemeine Notation (nicht Java!)

```
monitor Xyz {
  // Monitordaten
  int cnt;
  ...

  // Entry-Operationen
  entry op1( ... ) {
    ...
  }

  ...

  entry opN( ... ) {
    ...
  }

  // Monitor-lokale Operationen
  local loc1..N ( ... ) {
      ...
  }

}
```

Das Monitorkonzept (Forts.)

Condition-Variable

- Neben dem *gegenseitigen Ausschluss* wird auch ein Mechanismus benötigt, um Prozesse bzw. Threads innerhalb eines Monitors abhängig von *bestimmten Zuständen der Monitordaten* zu steuern:
  - im Monitor auf das Eintreten einer Bedingung warten
  - einen Wartenden (nach der Erfüllung der Bedingung) gezielt weiterlaufen lassen
- Erzeugen mittels `new Condition()`
- Warten mittels `wait()`
- Aufwecken mittels `signal()`

Das Monitorkonzept (Forts.)

Semantik der *wait*-Operation

- Eine wait-Operation wird üblicherweise innerhalb einer `if`-Anweisung oder `while`-Schleife verwendet
- Der zugehörige boolesche Ausdruck definiert die *Wartebedingung*
- Wirkungsweise des Aufrufes `cond.wait()`:
  - Der Aufrufer geht in einen Wartezustand über
    $\Rightarrow$ er wartet *vor der Condition* `cond`
  - dabei wird der Monitor entweder freigegeben oder die Monitorkontrolle geht an einen Wartenden über, der auf den Eintritt in den Monitor wartet
- Ein *wait*-Aufruf führt also zur Aufhebung des gegenseitigen Ausschlusses!
  $\Rightarrow$ notwendig, damit ein anderer in der Lage ist, den Wartenden wieder zu aktivieren

Das Monitorkonzept (Forts.)

Semantik der *signal*-Operation

- *signal* wird aufgerufen, wenn die Wartebedingung für einen vor der Condition wartenden Thread nicht mehr besteht
- Wirkungsweise des Aufrufes *cond.signal()*:
  - falls niemand vor der Condition wartet, ist der Aufruf wirkungslos
  - anderenfalls wird mindestens ein Wartender wieder aktiviert
- Aber nur genau einer kann die Monitorkontrolle haben!
  - Der bzw. die Anderen müssen vorübergehend warten
  - Implementierung mittels einer *urgent*-Queue
  - Die in der *urgent*-Queue Wartenden haben beim Monitoreintritt Priorität gegenüber denen, die beim Aufruf einer Monitoroperation warten
- Es sind drei Varianten möglich
  (1) Es wird genau ein Wartender aktiviert und der Aufrufer von *signal* behält die Monitorkontrolle
  (2) Es werden alle Wartenden aktiviert und der Aufrufer von *signal* behält die Monitorkontrolle
  (3) Es wird genau ein Wartender aktiviert und der Aktivierte übernimmt die Monitorkontrolle (Variante von Hoare)

Das Monitorkonzept (Forts.)

Unterschiede der drei *signal*-Varianten

- Die Wartebedingung basiert auf dem momentanen Zustand von Monitordaten
- Bei den Varianten 1 und 2
  - Die Wartebedingung muss von jedem gerade aktivierten Thread bzw. Prozess erneut überprüft werden, da der signalisierende oder ein anderer gerade aktivierter Thread (Prozess) bereits die Monitordaten so verändert haben könnte, dass die Wartebedingung nun wieder gilt

  $\Rightarrow$ Verwendung einer `while`-Schleife zur Formulierung der Wartebedingung
- Bei der 3. Variante
  - Durch die sofortige Übergabe der Monitorkontrolle hatte kein anderer Thread (Prozess) die Chance, die Monitordaten derart zu verändern, dass die Wartebedingung wieder gilt

  $\Rightarrow$ Verwendung einer `if`-Anweisung zur Formulierung der Wartebedingung

Das Monitorkonzept (Forts.)

Beispiel: Leser-Schreiber-Problem

- Gleichzeitiges Schreiben desselben Datensatzes ist verboten $\Rightarrow$ warten
- Lesen und gleichzeitiges Schreiben desselben Datensatzes ist verboten $\Rightarrow$ warten
- beliebig viele Leser dürfen einen Datensatz jedoch gleichzeitig lesen

  $\Rightarrow$ Wie geht das mit dem Monitorkonzept: Monitoroperationen werden ja immer unter gegenseitigem Ausschluss ausgeführt?!

- Lösungsidee
  - Verwendung des Konzepts des *Zugangsmonitors*:
    Der Monitor ist ausschließlich für die Synchronisation zuständig; die eigentlichen Operationen finden außerhalb des Monitors statt

Das Monitorkonzept (Forts.)

Beispiel: Leser-Schreiber-Problem (mit Leserpriorität)

```
class RW1 {

    public void lesen() {
        RW1mon.startRead();
        // lesen
        RW1mon.endRead();
    }

    public void schreiben() {
        RW1mon.startWrite();
        // schreiben
        RW1mon.endWrite();
    }
} //class RW1
```

```
monitor RW1mon {
    boolean activeWriter = false;
    int readctr = 0;
    Condition reader = new Condition();
    Condition writer = new Condition();

    entry startRead() {
        readctr++;
        if (activeWriter)
            reader.wait();
        reader.signal();
    }

    entry endRead() {
        readctr--;
        if (readctr == 0)
            writer.signal();
    }

    entry startWrite(){
        if (readctr > 0 || activeWriter)
            writer.wait();
        activeWriter = true;
    }

    entry endWrite(){
        activeWriter = false;
        // LESERPRIORITAET
        if (readctr > 0)
            reader.signal();
        else
            writer.signal();
    }
} // monitor
```

Frage: Welche *signal*-Variante wird verwendet?

Das Java-Monitorkonzept

Monitore in Java

- sind Objekte, die von Klassen erzeugt wurden, die mind. eine *synchronized*-Methode besitzen

  $\Rightarrow$ die *synchronized*-Methoden entsprechen den Monitoroperationen

  $\Rightarrow$ alle *synchronized*-Methoden eines Objekts werden unter gegenseitigem Ausschluss durchlaufen

- *synchronized* lässt sich auch für einzelne Blöcke definieren
  $\Rightarrow$ unübersichtlich, kann u. U. zu Deadlocks führen!

- es gibt keine *Conditions* (so wie bereits vorgestellt)
  - stattdessen besitzt jeder Monitor genau eine (namenlose) Warteschlange, an der Threads warten können (und dabei die Monitorkontrolle aufgeben)
- in Java sind die Monitorvarianten 1 und 2 implementiert
  $\Rightarrow$ alle Wartebedingungen müssen mit `while`-Schleifen abgesichert werden!

Das Java-Monitorkonzept (Forts.)

Methode *wait* (der Klasse Object)

- ein Thread testet seine Wartebedingung und kann mittels *wait* ggf. an einem Monitorobjekt warten

Methode *notify* (der Klasse Object)

- mittels *notify* wird der erste an einem Monitorobjekt wartende Thread geweckt, d. h. in den *ready*-Zustand versetzt

Methode *notifyAll* (der Klasse Object)

- mittels *notifyAll* werden alle an einem Monitorobjekt wartenden Threads geweckt, d. h. in den *ready*-Zustand versetzt

Vorsicht!

- bei geschachtelten Monitoraufrufen treten fast immer Verklemmungen auf!

Server-Programmierung

IP

Die Vermittlungsschicht (*Internet Layer*)

- übernimmt das Routing
- realisiert Ende-zu-Ende-Kommunikation
- überträgt **Pakete**
- ist im Internet durch **IP** realisiert

Zu lösende Probleme

- Sender und Empfänger brauchen netzweit eindeutige Bezeichner
  $\Rightarrow$ *IP-Adressen*
- spezielle Geräte ($\Rightarrow$ *Router*) müssen die Pakete weiterleiten
- Welche Route ist **im Moment** die beste?

IP

Adressierung

- IPv6-Adressen
  - 16 Byte lang (acht Bereiche hexadezimal)
  - z. B. `fb60:0:321:6533:fd5e:1965:2aa4:abba`
- IPv4-Adressen
  - 32 Bit lang (*dotted quads*)
- für das Routing braucht man die Netzadresse
  - Angabe mittels IP-Adresse und Netzmaske
    z. B. 141.72.70.0/23 bzw. Netmask: 255.255.254.0
- eine besondere Adresse ist die Broadcast-Adresse
  - alle Bits, die nicht zur Adressierung des Netzes verwendet werden, sind gesetzt
    z. B. 141.72.71.255

Frage: Welche Angaben sind zum Konfigurieren eines Netz-Interface notwendig?

UDP

User Datagram Protocol, RFC 768

- verbindungslose Kommunikation via *Datagramme*
- unzuverlässig ($\Rightarrow$ keine Fehlerkontrolle)
- ungeordnet ($\Rightarrow$ beliebige Reihenfolge)
- wenig Overhead ($\Rightarrow$ schnell)
- Größe der Nutzdaten:
  theoretisch max.: 65507 Byte, in der Praxis jedoch oft nicht mehr als 2 KiB

Anwendungsfelder

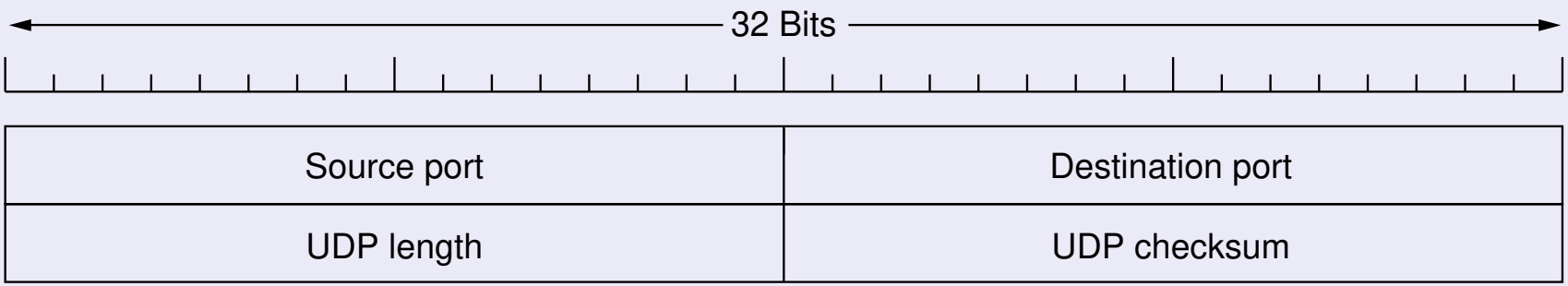
- Anwendungen mit vorwiegend kurzen Nachrichten (z.B. NTP, RPC, NIS), meist in lokalem Netz
- Anwendungen mit hohem Durchsatz, die ab und zu Fehler tolerieren (z.B. Multimedia)
- Multicasts sowie Broadcasts

UDP (Forts.)

Adressierung

- Socket: IP-Adresse und *Port*-Nummer
- Port bezeichnet rechnerinterne Kommunikationsadresse eines Prozesses
- Privilegierte Ports (1–1023): nur System-Prozesse dürfen sich daran binden

UDP-Header



TCP

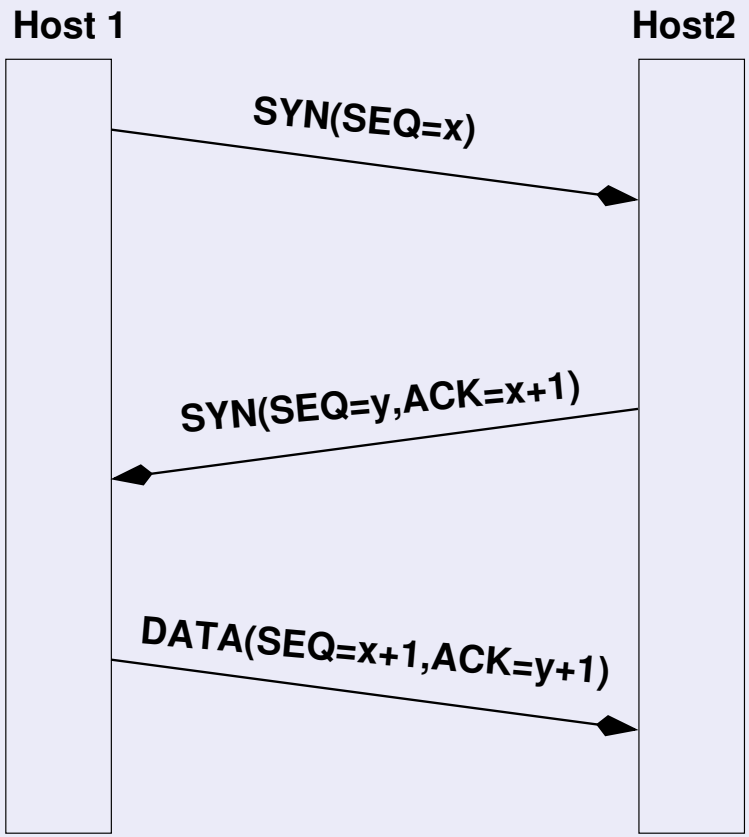
Transmission Control Protocol, RFC 793

- verbindungsorientierte Kommunikation
- ebenfalls Konzept der Ports
- Verbindungsaufbau zwischen zwei Prozessen (dreifacher Handshake, Full-Duplex-Kommunikation)
- geordnete Kommunikation
- zuverlässige Kommunikation
- Flusskontrolle
- hoher Overhead ($\Rightarrow$ vergleichsweise langsam)
- nur Unicasts

TCP (Forts.)

Verbindungsaufbau: TCP-Handshake

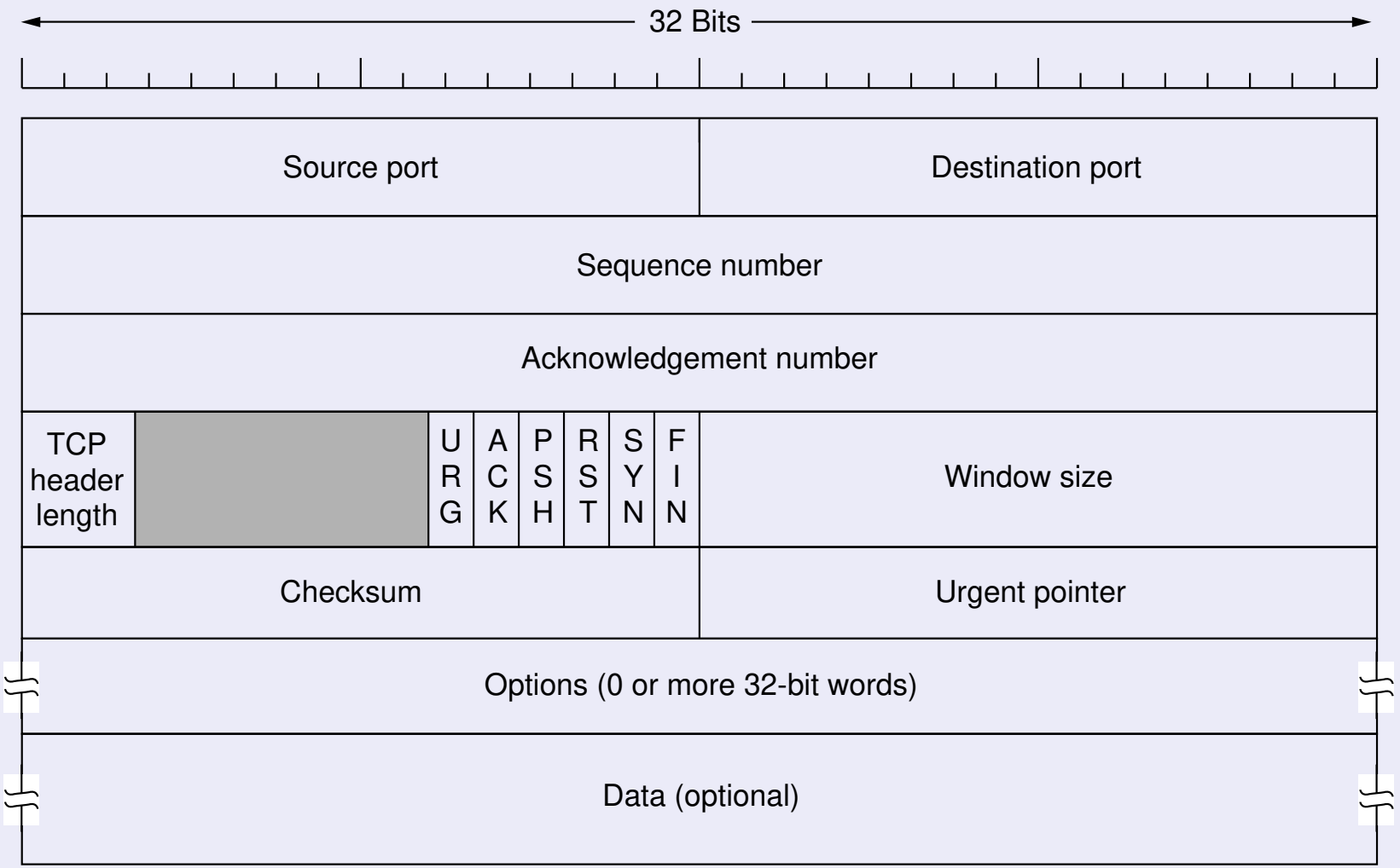
- zum Verbindungsaufbau werden die entsprechenden Flags im Header verwendet



Sequenznummern sind Byte-orientiert

TCP (Forts.)

TCP-Header



IP-Adressen und DNS

Adressauflösung

- wird entweder über die lokale hosts-Datei oder das *Domain Name System* (DNS) durchgeführt
- *class InetAddress*
- Achtung: es gibt keinen Konstruktor!
- Methode für eine allgemeine Adressauflösung

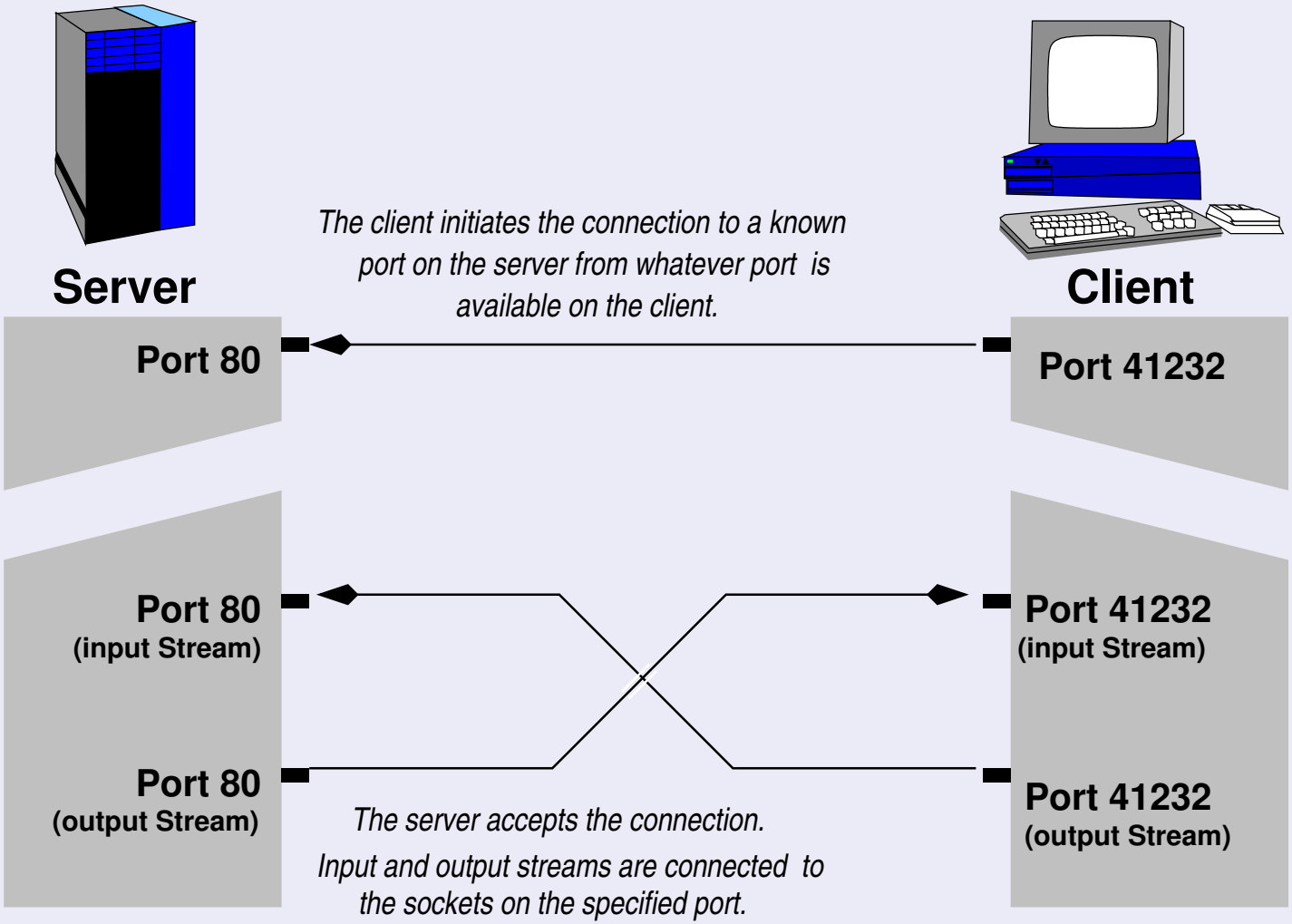
```
public static InetAddress getByName(String host)
  throws UnknownHostException
```

- *host* enthält entweder eine DNS-Rechneradresse (z.B. "www.dhbw-mannheim.de") oder eine IP-Adresse als String ("141.72.70.16") oder "localhost"

Frage: Was passiert eigentlich bei einer DNS-Namensauflösung?

TCP Sockets

Aufbau einer TCP-Verbindung über Ports



TCP Sockets (Forts.)

Programmablauf

(1) Der Server-Prozess wartet an dem bekannten Server-Port

(2) Der Client-Prozess erzeugt einen privaten Socket

(3) Der Socket baut zum Server-Prozess eine Verbindung auf, wenn der Server die Verbindung akzeptiert

(4) Für beide Parteien wird je ein Eingabestrom und ein Ausgabestrom eingerichtet, über den nun Daten ausgetauscht werden können

(5) Wenn alle Daten ausgetauscht wurden, schließen im Allg. beide Parteien die Verbindung

TCP-Sockets

class Socket

- implementiert in *java.net.Socket*
- Standardkonstruktor

```
public Socket(String host, int port)
  throws UnknownHostException, IOException
```

  - Es wird lokal ein privater Socket erzeugt.
  - Der Hostname `host` wird über DNS in die IP-Adresse aufgelöst, ggf. kann eine `UnknownHostException` auftreten
  - Es wird dann eine TCP-Verbindung zu dem entfernten Socket mit der Port-Nr. `port` aufgebaut, ggf. kann eine `IOException` auftreten

- Weiterer Konstruktor

```
public Socket(InetAddress host, int port)
  throws IOException
```

TCP-Sockets (Forts.)

Wichtige Methoden der Klasse Socket

```
public InputStream getInputStream() throws IOException

public OutputStream getOutputStream() throws IOException

public synchronized void close() throws IOException

public InetAddress getInetAddress()

public int getPort()

public int getLocalPort()

public InetAddress getLocalAddress()
```

Beispiel: Portscanner

Aufgabe

- Zu entwerfen ist ein JAVA-Programm, das für einen anzugebenden Rechner ausgibt, welche Netzdienste aktiv sind, d. h. an welchen privilegierten Ports Server-Prozesse warten.

! Portscanner **NIEMALS** auf fremde Systeme anwenden!

Lösungsidee

- Das Programm testet die Ports von 1 bis 1023. Für diese Ports werden der Reihe nach Sockets erzeugt.
- Das Erzeugen eines Client-Socket ist nur dann erfolgreich, wenn auf dem Server tatsächlich ein Prozess am fraglichen Port wartet.

Beispiel: Portscanner (Forts.)

Umsetzung in Java

```
import java.net.*;
import java.io.*;

public class LowPortScanner {
  public static void main(String[] args) {
    String host = "localhost";

    if (args.length > 0) {host = args[0];}

    for (int i = 1; i < 1024; i++) {
      try {
        Socket s = new Socket(host, i);
        System.out.println(
         "There is a server on port " + i + " at " + host);
        s.close();
      }
      catch (UnknownHostException e) {
        System.err.println(e);
        break;
      }
      catch (IOException e) {} // skip, probably no server waiting here
    } // end for
  }  // end main
}  // end LowPortScanner
```

TCP-Sockets (Forts.)

Austausch von Nutzdaten

- Nach erfolgtem Verbindungsaufbau können zwischen Client und Server mittels des *Socket-InputStream* und *Socket-OutputStream* Daten ausgetauscht werden
- Hierzu leitet man die rohen Daten am besten durch geeignete *Filter-Streams*, um eine möglichst hohe semantische Ebene zu erreichen
  - z. B. einfaches Umwandeln von Rohdaten in Zeichenketten, Datenkomprimierung, Verschlüsselung, usw.

    $\Rightarrow$ Anwendungsprogrammierung wird einfacher!
  - Ebenso für den Zugriff auf Dateien und verschiedene Geräte einsetzbar
  - *Reader* und *Writer* Filter-Streams eignen sich ausschließlich für die Verarbeitung von *Strings*!

TCP-Sockets (Forts.)

Wir verwenden

- *PrintWriter*
- *BufferedReader*
- *BufferedInputStream*
- *BufferedOutputStream*

- Die Netzwerkkommunikation kann damit bequem über wohlbekannte und komfortable Ein- und Ausgabe-Routinen (z. B. *readLine* oder *println*) durchgeführt werden

Filter-Streams

class PrintWriter

- Konstruktor

```
public PrintWriter ( OutputStream out )
```

- Wichtige Methoden

```
public void println (String s)
public void print (String s)
public void println ()
public void flush()
public void close()
```

- Die Ausgabe erfolgt immer zunächst in einen Puffer. Ein Weiterleiten an den tatsächlichen Ausgabestrom geschieht erst, wenn der Puffer voll ist bzw. nach *flush()* oder bei *close()*

Filter-Streams

class BufferedReader

- Konstruktor

  public BufferedReader (Reader in)

- Wichtige Methoden

  public String readLine ()

  - liest eine vollständige Zeile inkl. Zeilenende aus dem Stream; Rückgabewert enthält keine Zeilenendezeichen; Rückgabewert *null* kennzeichnet Stream-Ende
  - Die Eingabe erfolgt immer aus dem Puffer. Falls der Puffer leer ist, werden soviele Zeichen wie möglich in den Puffer eingelesen

  public void close() **throws** IOException

  - schließt den Strom

Filter-Streams

class BufferedOutputStream

- Konstruktor

```
public BufferedOutputStream ( OutputStream out )
```

- Wichtige Methoden

```
public void write(int b) throws IOException

public void write(byte[] data, int offset, int length)
        throws IOException

public void flush() throws IOException

public void close() throws IOException
```

- Die Ausgabe erfolgt immer zunächst in einen Puffer. Ein Weiterleiten an den tatsächlichen Ausgabestrom geschieht erst, wenn der Puffer voll ist bzw. nach *flush()* oder bei *close()*

Filter-Streams

class BufferedInputStream

- Konstruktor

```
public BufferedInputStream ( InputStream in )
```

- Wichtige Methoden

public int read() **throws** IOException

  - Der Rückgabewert von '-1' kennzeichnet dabei das Stream-Ende
  - Die Eingabe erfolgt immer aus dem Puffer. Falls der Puffer leer ist, werden soviele Zeichen wie möglich in den Puffer eingelesen

public void close() **throws** IOException

  - schließt den Strom

Beispiel: TCP Client

Echo-Client

- Der Echo-Dienst (Default TCP-Port 7) schickt alle Bytes, die an ihn gesendet werden, genauso zurück. Er wird i. Allg. zum Testen des Netzwerks verwendet
- Die Aufgabe: Es ist ein Java-Programm zu entwerfen, das sich mit dem Echo-Socket des Server-Rechners verbindet, von der Tastatur immer wieder eine Zeile einliest, diese an den Server sendet, dessen Antwort entgegennimmt und auf dem Bildschirm ausgibt
- Lösungsidee
  - Für die Kommunikation erzeugen wir drei *Streams*
  - Die Benutzereingaben lesen wir aus *userIn* über einen *BufferedReader*, der auf einem *InputStreamReader* aufsetzt, von der Standardeingabe *System.in* (i. Allg. die Tastatur)
  - Die Daten vom Server lesen wir aus *networkIn* über einen *BufferedReader*, der auf einem *InputStreamReader* aufsetzt, welcher seinerseits auf dem *Socket-InputStream* sitzt
  - Das Senden zum Server erfolgt von *networkOut* in einen *PrintWriter*, der auf dem *Socket-OutputStream* aufsetzt

Beispiel: TCP Client (Forts.)

Umsetzung in Java

```
public class EchoClient {
  public static final int serverPort = 7; // echo server

  public static void main(String[] args) {
    // declare variables first
    String hostname = "localhost";
    PrintWriter networkOut = null;
    BufferedReader networkIn = null;
    Socket s = null;
    try {
      s = new Socket(hostname, serverPort);
      System.out.println("Connected to echo server");
      networkIn = new BufferedReader( new InputStreamReader(s.getInputStream()));
      BufferedReader userIn = new BufferedReader( new InputStreamReader(System.in));
      networkOut = new PrintWriter(s.getOutputStream());

      // now do the real things:
      while (true) {
        String theLine = userIn.readLine();
        if (theLine.equals(".")) break;
        networkOut.println(theLine);
        networkOut.flush();
        System.out.println(networkIn.readLine());
      } // end while

    } catch (IOException e) {
      System.err.println(e);
    } finally {
      try {
        if (s != null)  s.close();
      } catch (IOException e) {}
    } // finally
  }  // end main
}  // end EchoClient
```

TCP-Sockets (Forts.)

Server-Programmierung: class ServerSocket

- Konstruktoren

```
public ServerSocket (int port)
        throws IOException, BindException

public ServerSocket (int port, int queueLength)
        throws IOException, BindException
```

- Wichtige Methoden

```
public Socket accept() throws IOException

public void close() throws IOException

public InetAddress getInetAddress()
public int getLocalPort()
```

  - *accept* wartet bis ein Client eine Verbindung aufbaut und liefert ein *Socket*-Objekt zurück, über das mit dem Client kommuniziert werden kann
  - Falls Verbindungsanfragen von Clients eintreffen, während der Server beschäftigt ist, werden diese in die Auftragswarteschlange eingereiht

Beispiel: TCP Server

DaytimeServer

- Die Aufgabe: Es ist ein einfacher Server zu entwerfen, der für jede aufgebaute Verbindung die Tageszeit an den jeweiligen Client zurücksendet
- Lösungsidee
  - Wir erzeugen ein Server-Socket und warten in einer Endlosschleife auf Clients. Nach erfolgreichem Verbindungsaufbau liefert *accept* den 'Kommunikations-Socket' zurück

Beispiel: TCP Server (Forts.)

Umsetzung in Java

```
import java.net.*;
import java.io.*;
import java.util.Date;

public class DaytimeServer {
  public final static int DEFAULT_PORT = 13;
  public static void main(String[] args) {
    int port = DEFAULT_PORT;
    PrintWriter out = null;
    Socket connection = null;
    try {
     ServerSocket server = new ServerSocket(port);
     while (true) {
       try {
         connection = server.accept();
         out = new PrintWriter(connection.getOutputStream());
         Date now = new Date();
         out.println(now.toString());
         out.flush();
       } catch (IOException e) {e.printStackTrace();}
       finally {
         try {
           if (connection != null) connection.close();
         } catch (IOException e) {}
       } // end finally
     }  // end while
   } catch (IOException e) {
     System.err.println(e);
   } // end catch
  } // end main
} // end DaytimeServer
```

UDP-Sockets

class DatagramSocket

- Ein *DatagramSocket* kann Nachrichten an beliebige Empfänger senden und von beliebigen Sendern empfangen
- Konstruktoren

```
public DatagramSocket () throws SocketException
    // Für Clients: Socket an privatem Port

public DatagramSocket ( int port ) throws SocketException
    // Für Server: Socket an festem  Port
```

- Wichtige Methoden

```
public void close() throws IOException

public void send (DatagramPacket dp) throws IOException

public void receive (DatagramPacket dp) throws IOException

public void setSoTimeout(int timeout)throws SocketException
```

UDP-Sockets (Forts.)

class DatagramPacket (Forts.)

- Ein *DatagramPacket* enthält eine Nachricht inklusive Headerinformationen
- Konstruktoren

```
public DatagramPacket ( byte[] data, int length,
                        InetAddress dest, int port )
```

  - Erzeugt ein Datagramm an den Port *port* des Zielrechners *dest* gerichtet mit dem Inhalt *data* (als Byte-Array); *length* gibt die Länge der Nachricht in Byte an

```
public DatagramPacket ( byte[] data, int length)
```

  - Erzeugt ein Datagramm für den Empfang der Sendung eines beliebigen Absenders in das Byte-Array *data*
    Das Byte-Array dient als Puffer zum Empfang der Nachricht und muss mit einer Größe > 0 zum Empfang bereits existieren; *length* gibt die Länge des Puffers in Byte an

UDP-Sockets (Forts.)

class DatagramPacket

- Wichtige Methoden

public InetAddress getAddress ()

- Liefert IP-Adresse des Absenders (falls empfangen)
  bzw. des Empfängers (falls zu senden)

public int getPort ()

- Liefert Port-Nummer des Senders (falls empfangen)
  bzw. des Empfängers (falls zu senden)

public byte[] getData ()

- Liefert den Datenteil (Pufferinhalt) des empfangenen bzw. des zu sendenden Datagramms
  Auf den Datenteil darf nur mittels dieser Getter-Methode zugegriffen werden!

UDP-Sockets (Forts.)

class DatagramPacket (Forts.)

- Wichtige Methoden (Forts.)

  public int getLength ()

  - Liefert die Anzahl der neu empfangenen Zeichen im Datenteil (Pufferinhalt) des empfangenen Datagramms

Verarbeitung eines *DatagramPacket*

- Konvertierung des Byte-Array in einen String

  public String (**byte**[] bytes, **int** start, **int** length)

  - Kopiert *length* Zeichen des Puffers des *DatagramPacket* – beginnend von Position *start* – in einen String

  public byte[] getBytes ()

  - Konstruiert ein Byte-Array aus einem String

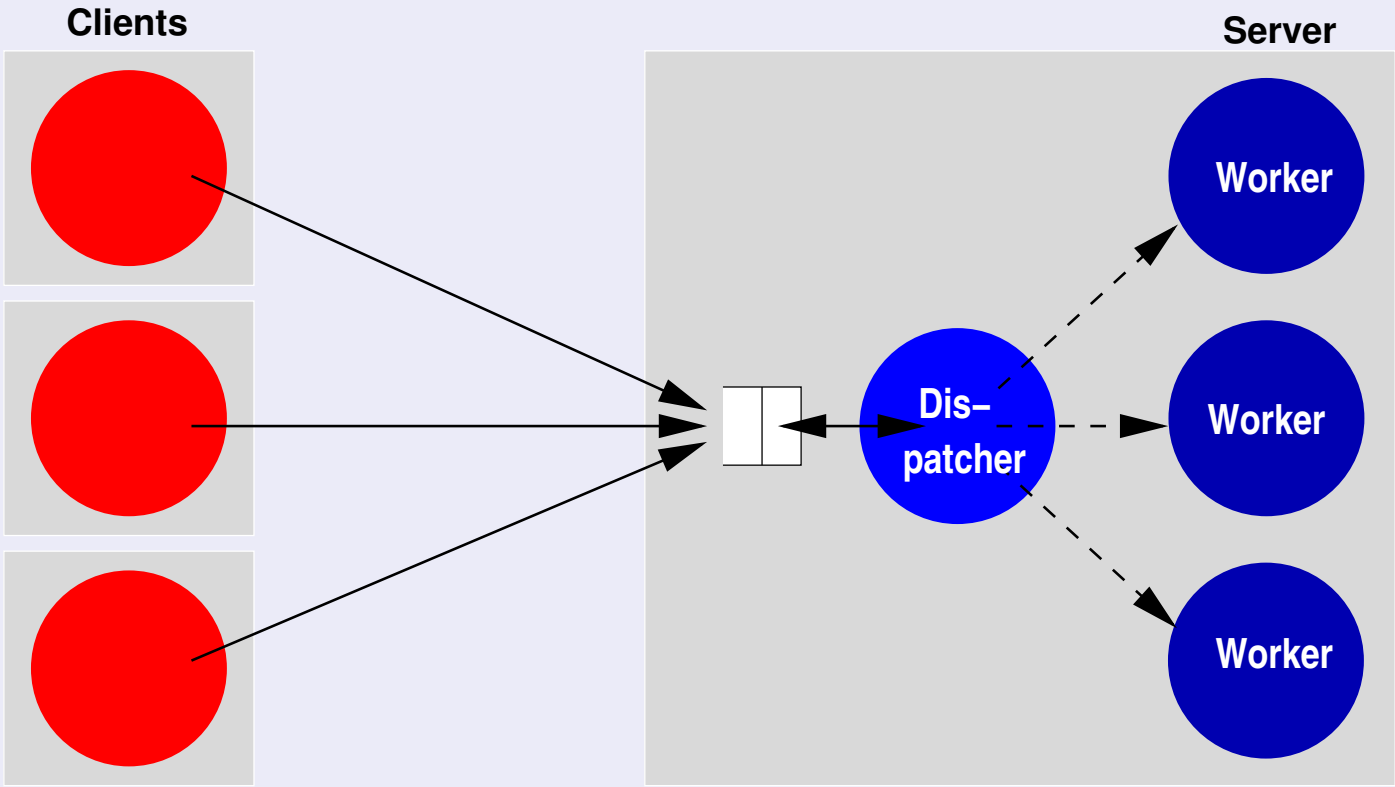
UDP-Sockets (Forts.)

Programmstrukturen

- für einen Client
  1. DatagramSocket auf privatem Port erzeugen
  2. DatagramPacket mit Daten und Zielinformation erzeugen
  3. DatagramPacket versenden
  4. ggf. Antwort empfangen und verarbeiten
- für einen Server
  1. DatagramSocket auf festem Port erzeugen
  2. Endlosschleife beginnen
  3. DatagramPacket vorbereiten
  4. DatagramPacket empfangen
  5. DatagramPacket verarbeiten
  6. ggf. Antwort erstellen und absenden

Multi-Threaded Server

Dispatcher- und Worker-Threads



Multi-Threaded Server (Forts.)

Thread-per-Request

- Idee
  - Für *jeden Auftrag* erzeugt der *Dispatcher Thread* einen neuen *Worker Thread*
  - Dieser *Worker Thread*
    – bearbeitet den Auftrag, – sendet ggf. das Ergebnis an den Aufrufer
    – beendet sich anschließend
- Vorteile
  - einfache Implementierung
  - keine gemeinsamen Synchronisationspunkte für die *Worker Threads*
    $\Rightarrow$ kein unnötiges Warten
  - scheinbar maximaler Durchsatz, da beliebig viele *Worker Threads* erzeugt werden können
- Nachteil
  - hoher Zeitaufwand für das Erzeugen und Beenden der *Worker Threads*
    $\Rightarrow$ Vorteile werden stark relativiert

Multi-Threaded Server (Forts.)

Thread-per-Connection

- Idee
  - Für *jede Verbindung* erzeugt der *Dispatcher* einen neuen *Worker*
  - Alle Aufträge dieser Verbindung werden an diesen *Worker* weitergereicht
  - Beim Schließen der Verbindung beendet sich der *Worker*
- Vorteile
  - weniger Aufwand für die Erzeugung und das Beenden der Threads
  - guter Durchsatz
- Nachteile
  - Implementierung komplexer bei verbindungsloser Kommunikation
  - schlechterer Durchsatz bei stark unterschiedlichen Anfragelasten der einzelnen Clients
  - bei Clients mit nur einer Anfrage pro Verbindung identisch zu Thread-per-Request
- Variante: Thread-per-Object (bei Objektorientierung)
  - Für *jedes Objekt* erzeugt der *Dispatcher* einen neuen *Worker*

Multi-Threaded Server (Forts.)

Worker-Pool

- Idee
  - Es wird ein *Pool* mit einer Anzahl von *Worker Threads* erzeugt
  - Jeder neue Auftrag wird an den nächsten freien *Worker* weitergereicht
  - Dieser
    – bearbeitet den Auftrag
    – sendet ggf. das Ergebnis an den Aufrufer
    – stellt sich anschließend in den Pool zurück
- Vorteile
  - weniger Aufwand für die Erzeugung und das Beenden der Threads
  - guter Durchsatz
  - Anzahl der *Worker Threads* dynamisch an die Anfragelast anpassbar
- Nachteile
  - Implementierung komplexer, Synchronisation notwendig
  - Verwaltung der gemeinsamen Warteschlange für die freien Worker ($\Rightarrow$ Pool)
- Variante
  - Eine Auftragswarteschlange einrichten, um Dispatcher und Worker zu entkoppeln