Work in progress

Interprozesskommunikation in Unix

Ute Bormann, TI2

2023-10-13

Inhalt

1. Exkurs: Software Transactional Memory (STM)
2. Unix-Unterstützung für Nebenläufigkeit
3. Sockets
4. Unix select()

Teil 1:
Exkurs: Software Transactional Memory (STM)

Resümee Nebenläufigkeit

- Interprozesskommunikation und Synchronisation durch:
  - gemeinsame Daten (Zugriff durch Locks, Semaphore, Monitore... geschützt)
  - Nachrichtenaustausch

Resümee Nebenläufigkeit

- Interprozesskommunikation und Synchronisation durch:
  - gemeinsame Daten (Zugriff durch Locks, Semaphore, Monitore... geschützt)
  - Nachrichtenaustausch
  - STM (in Zukunft? ⇒ Exkurs...)

Exkurs: Software Transactional Memory (STM)

- Bisher: Schutz von kritischen Abschnitten durch Locks, Semaphore,...
- Probleme:
  a) Richtiges Erkennen der krititschen Abschnitte
  b) Richtiges Einsetzen von lock()/unlock(), P()/V(),...
     1) Vergessen, falsch plazieren...
     2) Gefahr von Verklemmungen bei komplexeren Problemen

Exkurs: Software Transactional Memory (STM)

- Bisher: Schutz von kritischen Abschnitten durch Locks, Semaphore,...
- Probleme:
  a) Richtiges Erkennen der krititschen Abschnitte
  b) Richtiges Einsetzen von lock()/unlock(), P()/V(),...
    1) Vergessen, falsch plazieren...
    2) Gefahr von Verklemmungen bei komplexeren Problemen

a) Unvermeidbar

b1) Entschärfung durch Verwendung abstrakterer Konzepte:
Monitore, Petri-Netze, Nachrichtenaustausch,...

b2) Durch klassische Konzepte noch nicht gelöst!

Exkurs: Software Transactional Memory (STM)

- Bisher: Schutz von kritischen Abschnitten durch Locks, Semaphore,...
- Probleme:
  a) Richtiges Erkennen der krititschen Abschnitte
  b) Richtiges Einsetzen von lock()/unlock(), P()/V(),...
    1) Vergessen, falsch plazieren...
    2) Gefahr von Verklemmungen bei komplexeren Problemen

a) Unvermeidbar

b1) Entschärfung durch Verwendung abstrakterer Konzepte:
Monitore, Petri-Netze, Nachrichtenaustausch,...

b2) Durch klassische Konzepte noch nicht gelöst!

- Wdh.: Verklemmungen entstehen durch geschachtelte kritische Abschnitte bei einer Kombination von vier Randbedingungen

  ⇒ Diverse Ansätze zur Erkennung/Verhinderung...

Wdh.: Verklemmungen (Deadlocks)

- Situation: Zwei (oder mehr) Prozesse warten auf „Betriebsmittel“, die nur der/die andere(n) Wartende(n) freigeben kann/können
- Entsteht unter folgender Kombination von Randbedingungen

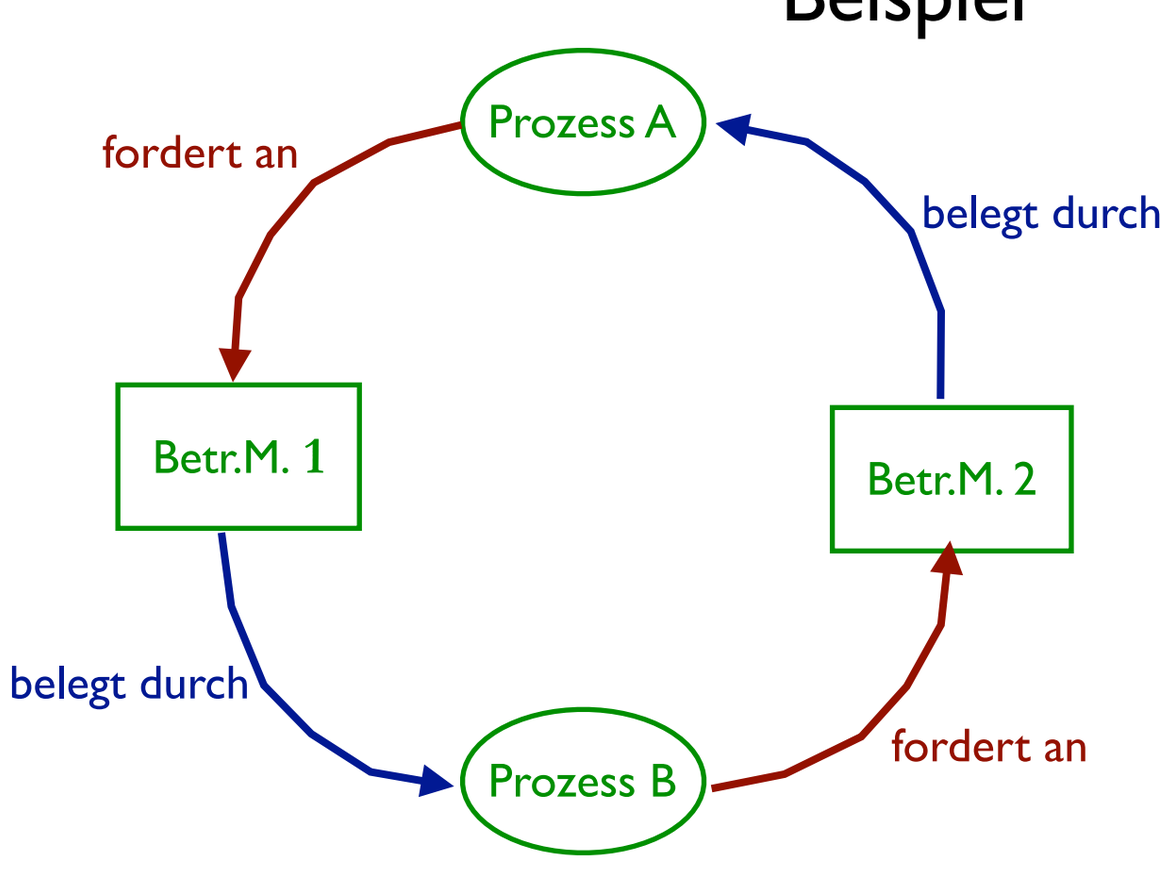
1. Nur bei Betriebsmitteln, die ein Prozess exklusiv für sich belegt hat
2. Jeder beteiligte Prozess hat bereits solche Belegungen und wartet auf weitere
3. Belegte Betriebsmittel können nicht zwangsentzogen werden

⇒ geschachtelte kritische Abschnitte

4. Es entsteht ein Zyklus von „Fordert-An/Belegt-Durch“-Relationen

⇒ unterschiedliche Schachtelung

Beispiel



Alternative Idee:

- Nebenläufige Ausführung kritischer Abschnitte erstmal zulassen
- Annahme: kommt bei gelegentlichen, kurzen kritischen Abschnitten selten vor.

⇒ Aber natürlich dennoch Inkonsistenzen nicht ausgeschlossen:

- Leser ermittelt u.U. inkonsistente Daten
- Schreiber produzieren u.U. unsinnige Datenbestände

Alternative Idee:

- Nebenläufige Ausführung kritischer Abschnitte erstmal zulassen
- Annahme: kommt bei gelegentlichen, kurzen kritischen Abschnitten selten vor.

⇒ Aber natürlich dennoch Inkonsistenzen nicht ausgeschlossen:

- Leser ermittelt u.U. inkonsistente Daten
- Schreiber produzieren u.U. unsinnige Datenbestände

- Abhilfe:

  1) Im Nachhinein Erkennen eines nebenläufigen Schreibvorgangs erforderlich

  2) Ausführung muss rückgängig gemacht werden können

  ⇒ Verwendung eines Transaktionskonzepts (wie in Datenbanken üblich)

  ⇒ STM (Software Transactional Memory)

Grundidee:

- Ggf. Verwendung von speziellen Maschineninstruktionen
  ⇒ Vermerken von nebenläufigen Schreibvorgängen
  (ähnlich zu load_locked()/store_conditional())

Wdh.: Load_locked / Store_conditional

- Test_and_set ist eigentlich zu mächtig
- „Laden“ und „speichern“ können getrennte Operationen sein, sofern nebenläufiger Zugriff anderer Prozesse bemerkt wird

```
bool key = false;
lock() {
  do {
    k = load_locked(&key);
  } while (k==true || !store_conditional(&key,true));
}

unlock() {
  key = false;
}
```

- Store_conditional liefert false, wenn anderer Prozess seit load_locked auf key zugegriffen hat (+ key bleibt false) ⇒ erneut probieren
- Algorithmus hat potentielles „After-you-after-you“-Problem

Grundidee:

- Ggf. Verwendung von speziellen Maschineninstruktionen
  ⇒ Vermerken von nebenläufigen Schreibvorgängen
  (ähnlich zu load_locked()/store_conditional())

- Mitführen eines Logs der durchgeführten Änderungen
  ⇒ Zurückfallen auf alten Zustand möglich (Achtung: Ausnahmen, z.B. im I/O-Bereich)

Grundidee:

- Ggf. Verwendung von speziellen Maschineninstruktionen
  ⇒ Vermerken von nebenläufigen Schreibvorgängen
  (ähnlich zu load_locked()/store_conditional())

- Mitführen eines Logs der durchgeführten Änderungen
  ⇒ Zurückfallen auf alten Zustand möglich (Achtung: Ausnahmen, z.B. im I/O-Bereich)

- Kapselung der relevanten Programmabschnitte (geeignete Syntax dafür erforderlich), z.B.:

```
atomic {
  p.x = q.x;
  p.y = q.y;
}
```

z.B. syntaktisch ähnlich zu Java-Syntax:

```
synchronized (myObj) {
 … kritischer Abschnitt …
}
```

Grundidee:

- Ggf. Verwendung von speziellen Maschineninstruktionen
  ⇒ Vermerken von nebenläufigen Schreibvorgängen
  (ähnlich zu load_locked()/store_conditional())

- Mitführen eines Logs der durchgeführten Änderungen
  ⇒ Zurückfallen auf alten Zustand möglich (Achtung: Ausnahmen, z.B. im I/O-Bereich)

- Kapselung der relevanten Programmabschnitte (geeignete Syntax dafür erforderlich), z.B.:

```
atomic {
  p.x = q.x;
  p.y = q.y;
}
```

- Außerdem: Einseitige Synchronisation

```
atomic {
  ...
  if (queuesize <= 0) atomic_retry();
  ...
}
```

⇒ Abbruch der Transaktion und neuer Versuch

Vergleich mit Java:

```
synchronized(myObj) {
   …
   myObj.wait();
   …
}
```

Vorteile von STM:

- Programmierer müssen nur noch atomic-Blöcke kennzeichnen
- Auch Schachtelung von atomic-Blöcken möglich

  ⇒ keine Gefahr von Verklemmungen, da Rückfall auf alten Zustand, sobald nebenläufiger Schreibvorgang bemerkt wird

  ⇒ nach Rückfall automatisch erneuten Versuch starten

  ⇒ allerdings weiterhin potentielles „After-you-after-you“-Problem

Vorteile von STM:

- Programmierer müssen nur noch atomic-Blöcke kennzeichnen
- Auch Schachtelung von atomic-Blöcken möglich

  ⇒ keine Gefahr von Verklemmungen, da Rückfall auf alten Zustand, sobald nebenläufiger Schreibvorgang bemerkt wird

  ⇒ nach Rückfall automatisch erneuten Versuch starten

  ⇒ allerdings weiterhin potentielles „After-you-after-you“-Problem

Jedoch:

- STM setzt Programmiersprachen-/Compiler-Unterstützung voraus
- STM setzt einiges an Plattformunterstützung voraus
  (geeignete Maschineninstruktionen, Schreiben von Logs)

  ⇒ recht aufwendige Umsetzung

- Nur sinnvoll einsetzbar bei kritischen Abschnitten geringen Wettbewerbs und sofern Rückfall möglich...

Vorteile von STM:

- Programmierer müssen nur noch atomic-Blöcke kennzeichnen
- Auch Schachtelung von atomic-Blöcken möglich

  ⇒ keine Gefahr von Verklemmungen, da Rückfall auf alten Zustand, sobald nebenläufiger Schreibvorgang bemerkt wird

  ⇒ nach Rückfall automatisch erneuten Versuch starten

  ⇒ allerdings weiterhin potentielles „After-you-after-you“-Problem

Jedoch:

- STM setzt Programmiersprachen-/Compiler-Unterstützung voraus
- STM setzt einiges an Plattformunterstützung voraus (geeignete Maschineninstruktionen, Schreiben von Logs)

  ⇒ recht aufwendige Umsetzung

- Nur sinnvoll einsetzbar bei kritischen Abschnitten geringen Wettbewerbs und sofern Rückfall möglich...

Bedeutung:

- STM-Konzept seit einigen Jahren in Erprobung (Hardware-Komponenten und diverse Implementierungen verfügbar)
- Aber eher Minderheitenprogramm, da in Mehrprozessorumgebung Kommunikation über Nachrichtenkanäle üblicher.…

Fragen – Teil 1

- Was ist *STM (Software Transactional Memory)*?

Teil 2:
Unix-Unterstützung für Nebenläufigkeit

Resümee Nebenläufigkeit

- Interprozesskommunikation und Synchronisation durch:
  - gemeinsame Daten (Zugriff durch Locks, Semaphore, Monitore... geschützt)
  - Nachrichtenaustausch
  - STM (in Zukunft? ⇒ Exkurs...)

→ • Bereitstellung von Mechanismen dieser Art über:
  - Programmiersprachenunterstützung (z.B. CSP, Java)
  - Bibliotheksunterstützung (z.B. Pthreads)
  - Betriebssystemunterstützung

Unterstützung der Kommunikation/Synchronisation zwischen Prozessen durch Unix

- Synchronisation im Kern (bei Zugriff auf gemeinsame Betriebsmittel):
  - Nicht-Präemption (Einprozessor-System)
  - ggf. Unterbrechungsausschluss
  - ggf. Spinlocks (Mehrprozessorsystem)

Unterstützung der Kommunikation/Synchronisation zwischen Prozessen durch Unix

- Synchronisation im Kern (bei Zugriff auf gemeinsame Betriebsmittel):
  - Nicht-Präemption (Einprozessor-System)
  - ggf. Unterbrechungsausschluss
  - ggf. Spinlocks (Mehrprozessorsystem)
- Kommunikation zwischen Benutzerprozessen/Threads:
  - gemeinsamer Adressraum (ggf. Shared-Memory)
  - gemeinsame Dateien
  - Nachrichtenkanäle (z.B. Pipes)

  ⇒ Diverse Synchronisationsmechanismen im Unix-Umfeld

Unterstützung der Kommunikation/Synchronisation zwischen Prozessen durch Unix

- Synchronisation im Kern (bei Zugriff auf gemeinsame Betriebsmittel):
  - Nicht-Präemption (Einprozessor-System)
  - ggf. Unterbrechungsausschluss
  - ggf. Spinlocks (Mehrprozessorsystem)
- Kommunikation zwischen Benutzerprozessen/Threads:
  - gemeinsamer Adressraum (ggf. Shared-Memory)
  - gemeinsame Dateien
  - Nachrichtenkanäle (z.B. Pipes)

  ⇒ Diverse Synchronisationsmechanismen im Unix-Umfeld
- Kommunikation über gemeinsamen Adressraum:
  - In Unix klassisch keine Synchronisationshilfsmittel

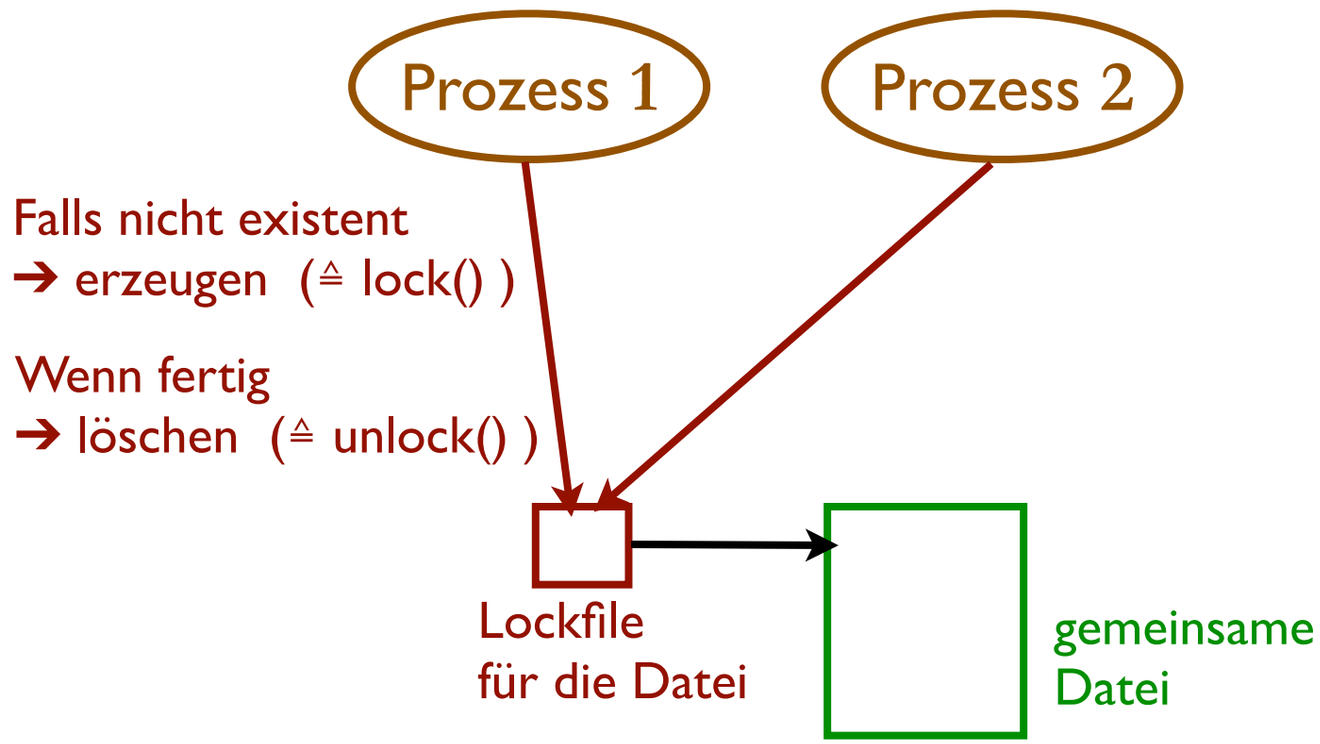
    ⇒ verschiedene Erweiterungen, z.B.
    - System V: Semaphore
    - Pthread-Bibliothek (Semaphore, Locks, ...)

- Kommunikation über gemeinsame Dateien:

  a) Verwendung von expliziten „Lock-Files“

  ⇒ Anlegen (= lock()) bzw. Löschen (= unlock()) durch die Benutzer

  - Unteilbares Anlegen/Löschen erforderlich
  - Zuordnung zu zu schützender Datei erforderlich (durch die Benutzer)
  - Bleibt bei Systemzusammenbrüchen u.U. liegen



b) Systemaufruf flock(): (BSD)

⇒ explizites lock()/unlock() für angegebene Datei
(falls belegt: blockieren oder Fehlerrückkehr)

- read()/write() davon unberührt
- kein automatisches (erzwungenes) Locking

c) Systemaufruf lockf(): (SysV)

⇒ über flock() hinausgehende Funktionalität:

- auch Teile von Dateien schützbar
- auch erzwungenes Locking realisierbar
  ⇒ bei read()/write()-Aufruf automatisch durchführen

- Kommunikation über Nachrichtenkanäle

  ⇒ asynchroner Nachrichtenaustausch

- Bisher betrachtet:
  - Unix-Signale ⇒ Sonderfall
  - Pipes (z.B. date | lpr)
- Außerdem:
  - Named Pipes (FIFOs)
  - Sockets

Pipes

- Sequentieller Bytestrom ⇒ „Puffer“ (Kanal)
- Unidirektional
- Zugriff über Dateisystemschnittstelle:

  write() ≙ send()

  read() ≙ receive()

Pipes

- Sequentieller Bytestrom ⇒ „Puffer“ (Kanal)
- Unidirektional
- Zugriff über Dateisystemschnittstelle:
  write() ≙ send()
  read() ≙ receive()

- Wie erfahren Prozesse davon?
  - Müssen „verwandt“ sein (z.B. Vater/Kind, Geschwister)
  - Vater erzeugt Pipe

pipe() Liefert zwei Filedeskriptoren auf Pipe-Objekt
– fd[0] = Lesen
– fd[1] = Schreiben

fork()

fork() Kinder erben Filedeskriptoren des Vaters

close(fd[0])

write(fd[1],buf,len)

close(fd[1]) Überflüssigen Deskriptor schließen
+ ggf. Ein-/Ausgabeumlenkung auf stdout/stdin

...

read(fd[0],buf,len)

- Beispiel: date | lpr (Beides Kinder von Shell)

Named Pipes (FIFOs)

- Eigenschaften wie Pipes
- Jedoch keine „Verwandtschaft“ erforderlich

  ⇒ Pipe-Objekt benötigt eindeutigen Namen zur Identifikation („Kanal-Bezeichner“)

  ⇒ Pfadname im Dateisystem

  mkfifo (path,...) ⇒ Dateityp = FIFO

- Allgemeines Problem: Prozesse müssen Namen kennen

Fragen – Teil 2

- Worin unterscheiden sich die Eigenschaften der folgenden Unix-Mechanismen zur Interprozesskommunikation:

  a) *Pipes*,

  b) *Named Pipes*,

  c) *Sockets*? (s. Teil 3)

Teil 3:
Sockets

Sockets

IPC: interprocess communication

- Pipes nicht für alle IPC-Probleme ausreichend

  ⇒ Zusätzliche Anforderungen:

  - bidirektionale Kommunikation
  - nicht nur sequentielle Byteströme
  - auch über Systemgrenzen hinweg
  - Unterschiedliche Kommunikationsformen unterstützen

Sockets

- Pipes nicht für alle IPC-Probleme ausreichend

  ⇒ Zusätzliche Anforderungen:

  - bidirektionale Kommunikation
  - nicht nur sequentielle Byteströme
  - auch über Systemgrenzen hinweg
  - Unterschiedliche Kommunikationsformen unterstützen

- Stream-Sockets („Bidirektionale Pipe“) ⇒ TCP

  - Bytestrom
  - sequentielle Übertragung
  - zuverlässig
  - verbindungsorientiert

    ⇒ Vergleich: Telefongespräch

Sockets

- Pipes nicht für alle IPC-Probleme ausreichend

  ⇒ Zusätzliche Anforderungen:

  - bidirektionale Kommunikation
  - nicht nur sequentielle Byteströme
  - auch über Systemgrenzen hinweg
  - Unterschiedliche Kommunikationsformen unterstützen

- Stream-Sockets („Bidirektionale Pipe“) ⇒ TCP
  - Bytestrom
  - sequentielle Übertragung
  - zuverlässig
  - verbindungsorientiert

    ⇒ Vergleich: Telefongespräch

- Datagram-Socket („Paketübergabe“) ⇒ UDP
  - Nachrichten (Pakete)
  - beliebige Reihenfolge
  - u.U. unzuverlässig (Verluste, Duplikate)
  - verbindungslos

    ⇒ Vergleich: Brief-/Paketzustellung

Sockets

- Pipes nicht für alle IPC-Probleme ausreichend

  ⇒ Zusätzliche Anforderungen:

  - bidirektionale Kommunikation
  - nicht nur sequentielle Byteströme
  - auch über Systemgrenzen hinweg
  - Unterschiedliche Kommunikationsformen unterstützen

- Stream-Sockets („Bidirektionale Pipe“) ⇒ TCP
  - Bytestrom
  - sequentielle Übertragung
  - zuverlässig
  - verbindungsorientiert

    ⇒ Vergleich: Telefongespräch

- Datagram-Socket („Paketübergabe“) ⇒ UDP
  - Nachrichten (Pakete)
  - beliebige Reihenfolge
  - u.U. unzuverlässig (Verluste, Duplikate)
  - verbindungslos

    ⇒ Vergleich: Brief-/Paketzustellung

Im folgenden nur Stream Sockets näher betrachtet

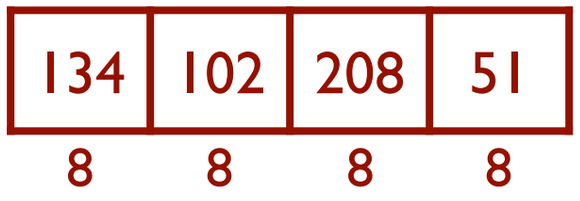
- Sockets repräsentieren Kommunikationsendpunkte,
  nicht Kanäle
- Zwei Formen der Adressierung von Kommunikationspartnern:
  - System-intern
  - System-übergreifend

- System-intern (Beispiel: Unix ⇒ AF_UNIX)
  - Socket erhalten Dateinamen (s. Named Pipes)
  - z.B. /dev/printer ⇒ Socket des Drucker-Spoolers

- System-übergreifend (Beispiel: Internet ⇒ AF_INET)
- Zweiteilige Adressen;

  a) Adressierung des Systems (klassisch: 32-Bit-Hostadresse)

  z.B. 134.102.208.51



  ⇒ „weltweiter“ Nummernraum

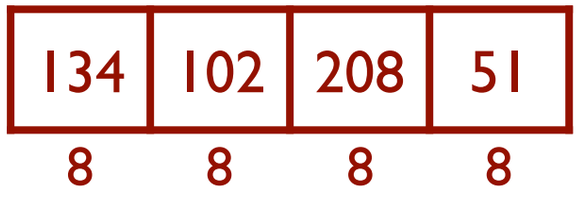
  ⇒ unterstrukturiert: Netzteil + Rechner/Host-Teil

  - Übergang IPv4-Adresse (32 Bit) ⇒ IPv6-Adresse (128 Bit)

- System-übergreifend (Beispiel: Internet ⇒ AF_INET)
- Zweiteilige Adressen;

  a) Adressierung des Systems (klassisch: 32-Bit-Hostadresse)

  z.B. 134.102.208.51



  ⇒ „weltweiter“ Nummernraum

  ⇒ unterstrukturiert: Netzteil + Rechner/Host-Teil

  - Übergang IPv4-Adresse (32 Bit) ⇒ IPv6-Adresse (128 Bit)

  b) Adressierung des Sockets innerhalb des Systems (16-Bit-Portnummer)

  z.B. HTTP 80 / HTTPS 443
  SMTP 25
  ssh 22

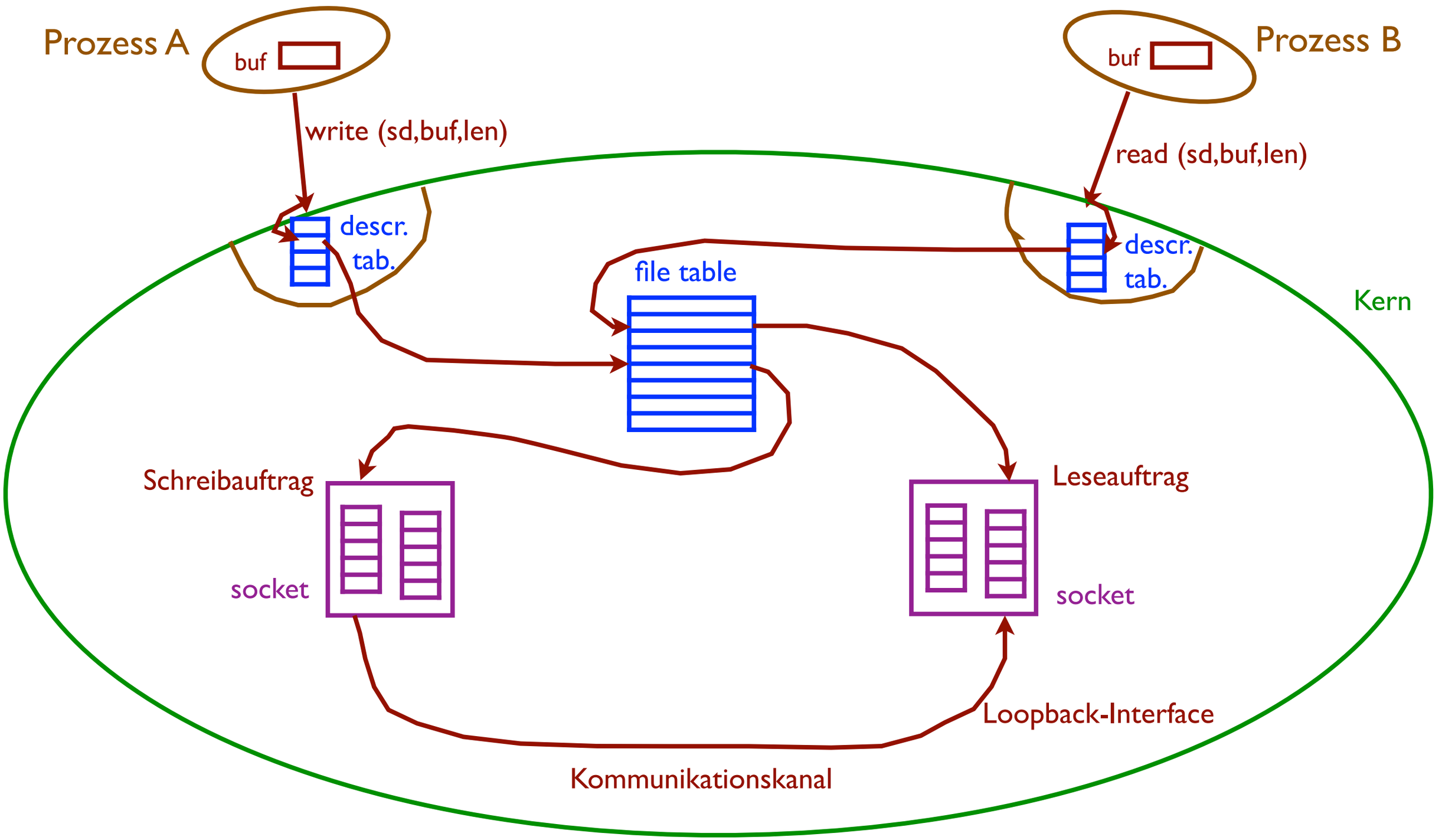
- Hostadresse + Portnummer
  ⇒ bei bind() zuweisen
  ⇒ bei connect() zur Identifikation des Empfängers angeben

Kleine Aufgabe

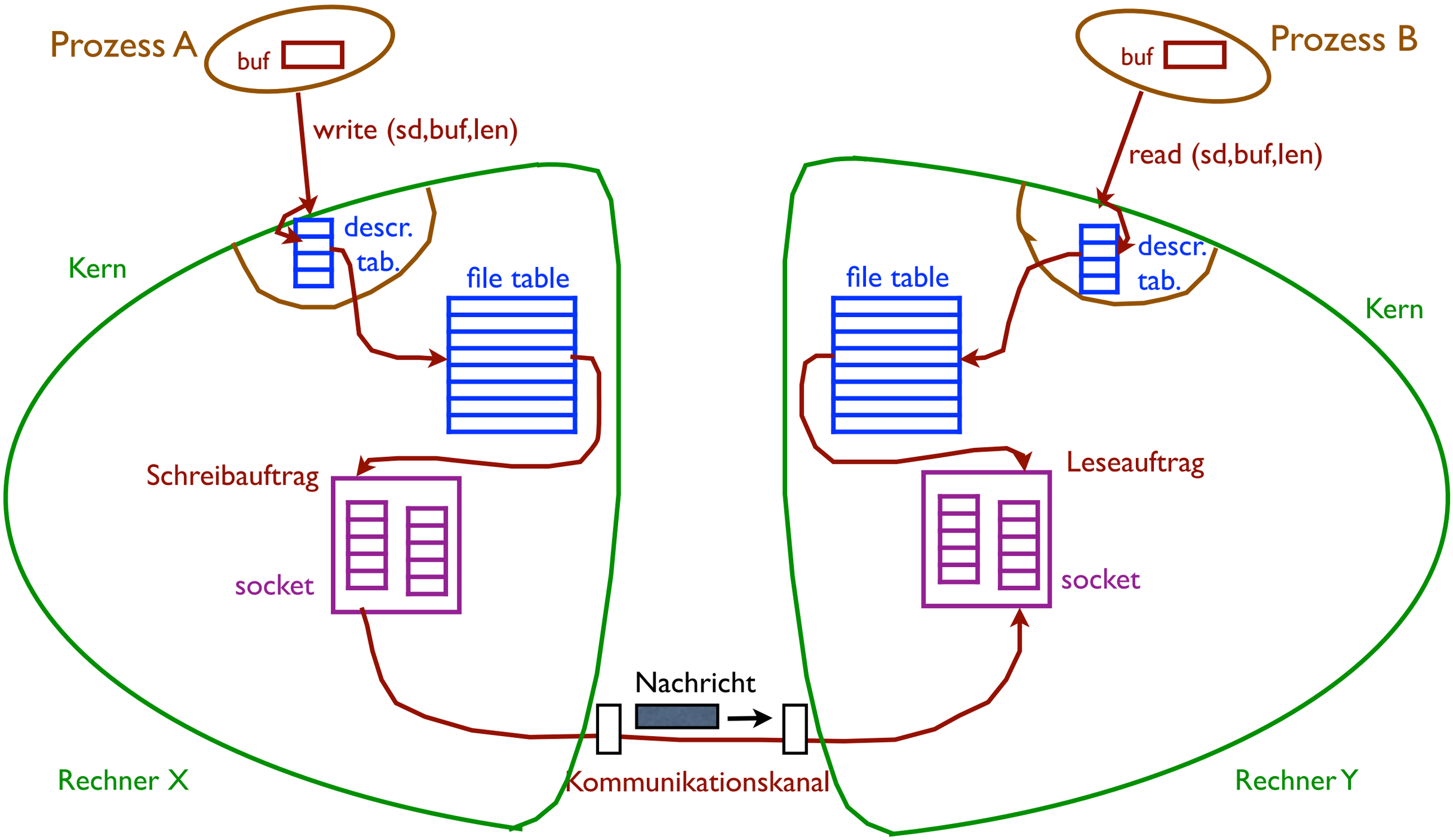
Worin unterscheiden sich *Stream-Sockets* und *Datagram-Sockets* hinsichtlich der Adressierung der Kommunikationspartner?

(Hinweis: vgl. Telefongespräch vs. Paketpost)

Kern-interne Verwaltung von Sockets (vereinfacht)



Kern-interne Verwaltung von Sockets (vereinfacht)



Verwendung von Sockets (Beispiel: Stream-Sockets)

- Erzeugung:

```
sd = socket (domain, type, protocol)
sd = socket (AF_INET, SOCK_STREAM, 0)
```

socket descriptor

IP-Adressen (statt systemintern)

Stream-Socket (statt Datagram-Socket)

- Verbindungsaufbau: (≠ open())

```
connect (sd,  addr, addlen)
```

Pointer auf Adressstruktur (Typ: sockaddr)

sizeof(...)

- Daten lesen/schreiben: (fast wie bei Datei)

```
alen = read (sd, buf, len)
alen = write (sd, buf, len)
```

- Verbindung schließen:

```
close (sd)
```

Automatisch bei Programmende

Verwendung von Sockets (Beispiel: Stream-Sockets)

- Erzeugung:

```
sd = socket (domain, type, protocol)
sd = socket (AF_INET, SOCK_STREAM, 0)
```

socket descriptor

IP-Adressen (statt systemintern)

Stream-Socket (statt Datagram-Socket)

- Verbindungsaufbau: (≠ open())

```
connect (sd,  addr, addlen)
```

Pointer auf Adressstruktur (Typ: sockaddr)

sizeof(...)

- Daten lesen/schreiben: (fast wie bei Datei)

```
alen = read (sd, buf, len)
alen = write (sd, buf, len)
```

- Verbindung schließen:

```
close (sd)
```

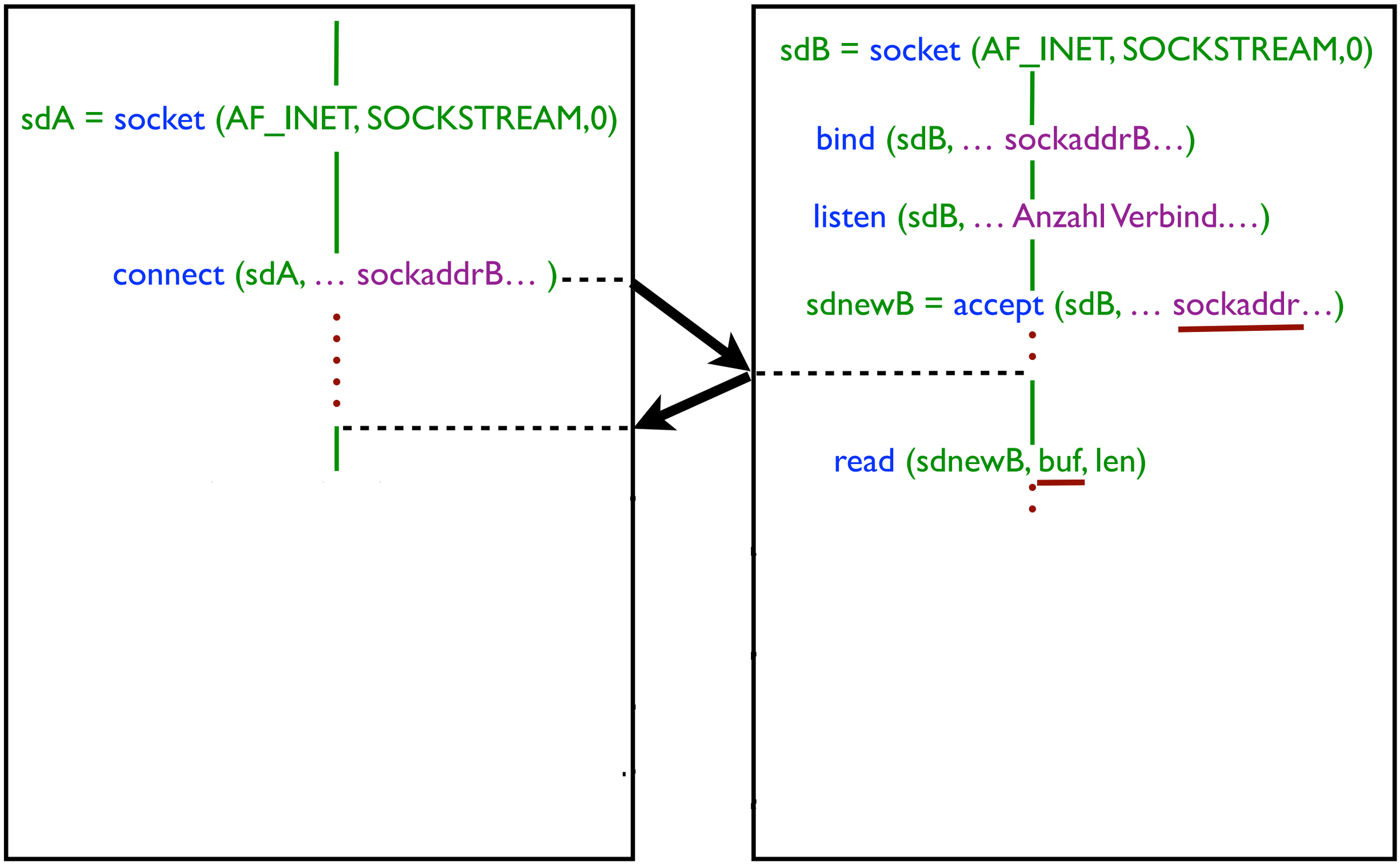
Automatisch bei Programmende

Weitgehend kompatibel mit generischer Systemaufrufschnittstelle zum Datei-/Geräte-/Pipezugriff
⇒ Vereinheitlichtes Lesen/Schreiben

Beispiel:

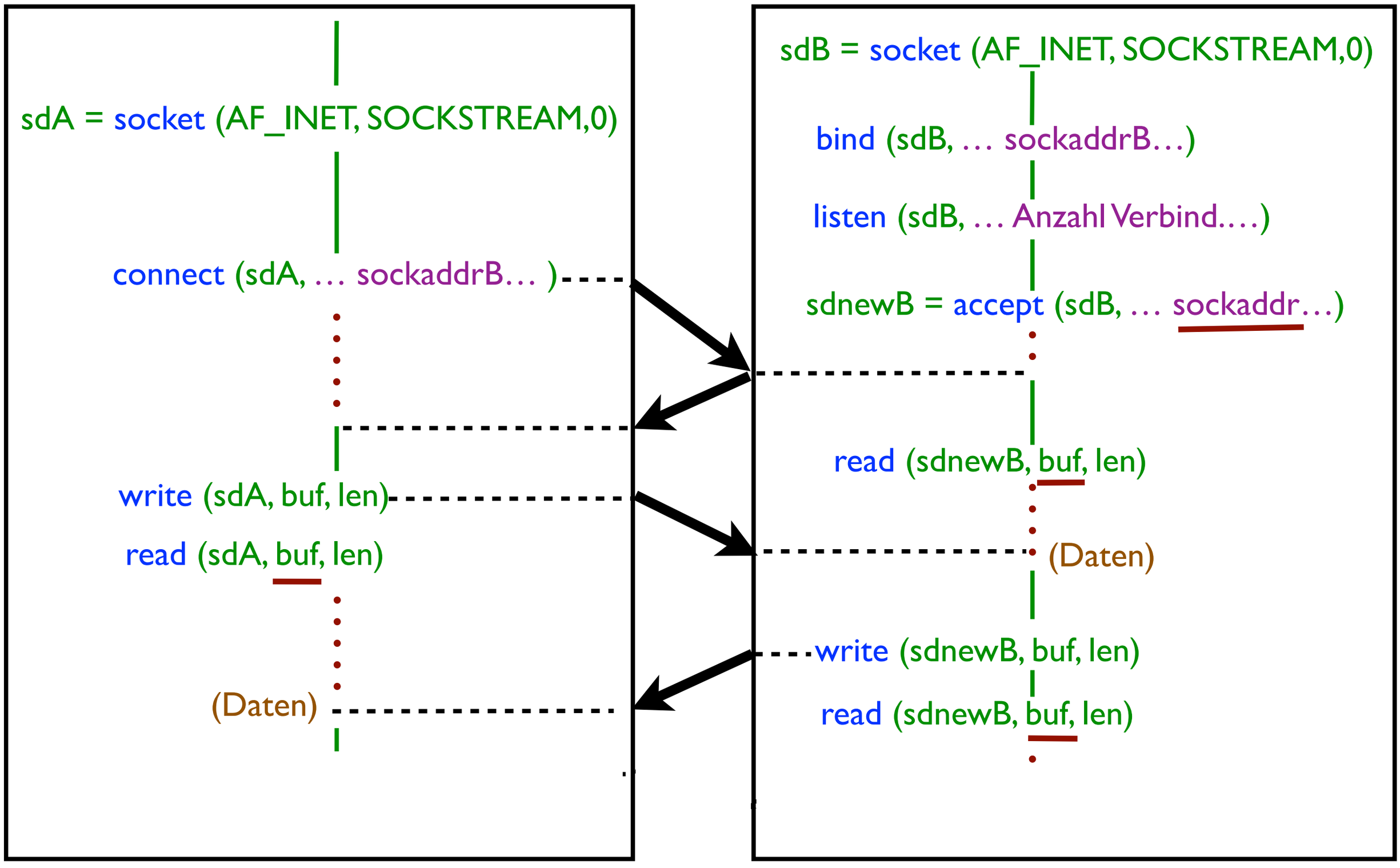
- (Unterstrichene Parameter: Werte werden bei der Ausführung erst ermittelt)

Beispiel:



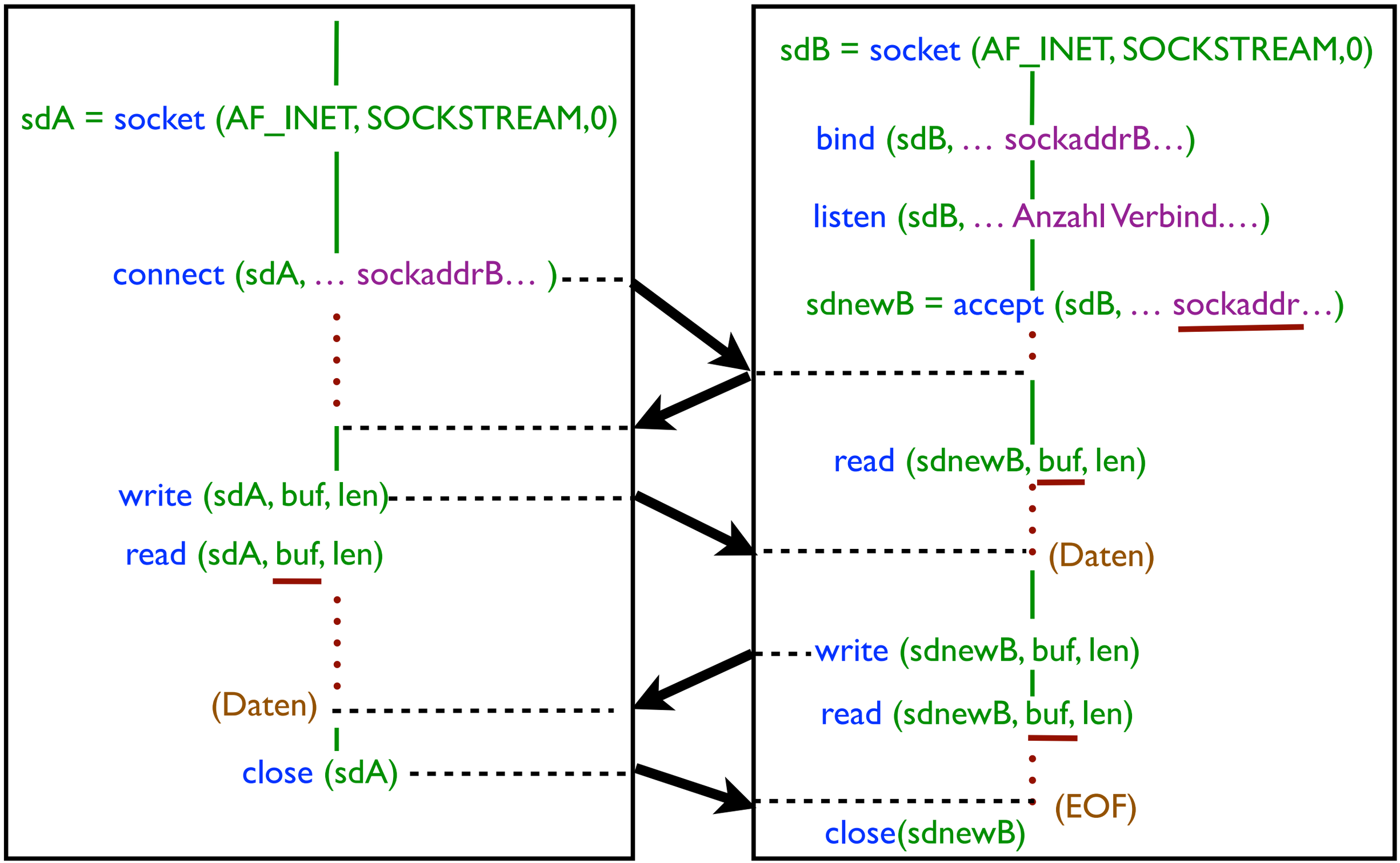
- (Unterstrichene Parameter: Werte werden bei der Ausführung erst ermittelt)

Beispiel:



- (Unterstrichene Parameter: Werte werden bei der Ausführung erst ermittelt)

Beispiel:



- (Unterstrichene Parameter: Werte werden bei der Ausführung erst ermittelt)

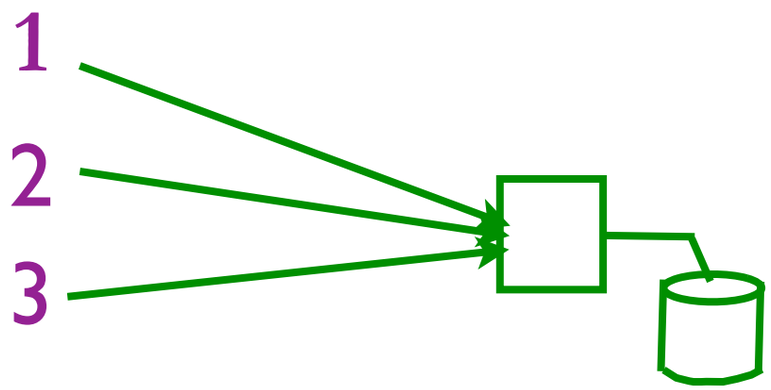
Fragen – Teil 3

- Wie lässt sich der Zugriff auf Sockets in die generische Systemaufrufschnittstelle zum Zugriff auf Dateien einordnen?
- Wie können Sockets adressiert werden?

Teil 4:
Unix select()

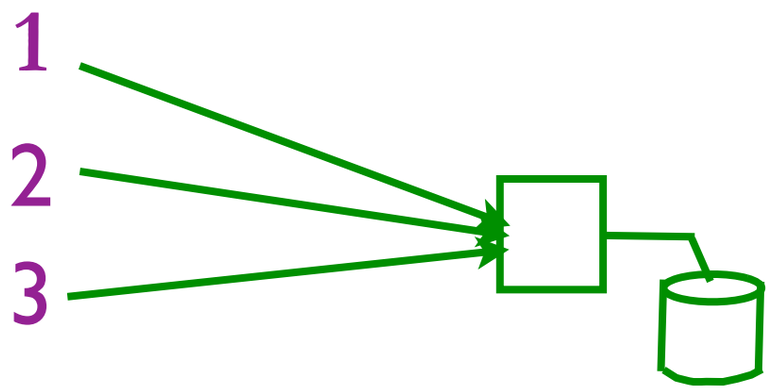
Verwaltung mehrerer Kommunikationsbeziehungen

- Prozess „wartet“ auf mehrere potentielle Ereignisse,
  z.B. Bedienen mehrerer Kommunikationsschnittstellen



Verwaltung mehrerer Kommunikationsbeziehungen

- Prozess „wartet“ auf mehrere potentielle Ereignisse,
  z.B. Bedienen mehrerer Kommunikationsschnittstellen



- Konkreteres Beispiel: Lesen aus mehreren Sockets (+ ggf. Tastatur)

- Deskriptoren reihum abfragen problematisch, da:
  - umständlich
  - verbraucht unnötig Prozessorzeit (busy waiting)
  - read() i.Allg. blockierend
    (warten auf Ereignis 1, während Ereignis 2 eingetreten ist)

⇒ „Sammel-Wartepunkt“ erforderlich

⇒ ähnlich zu guarded commands

- Deskriptoren reihum abfragen problematisch, da:
  - umständlich
  - verbraucht unnötig Prozessorzeit (busy waiting)
  - read() i.Allg. blockierend
    (warten auf Ereignis 1, während Ereignis 2 eingetreten ist)

  ⇒ „Sammel-Wartepunkt“ erforderlich

  ⇒ ähnlich zu guarded commands

- Wdh.: Auf mehrere Eingaben warten in CSP bzw. Erlang:

Alternativanweisung in CSP:

```
[ P1 ? variable1 → anweisung1
[] P2 ? variable2 → anweisung2
[] P3 ? variable3 → anweisung3
]
```

Empfang von Nachrichten in Erlang

```
receive
  MessagePattern1 ->
    Actions1;
  MessagePattern2 ->
    Actions2;
   …
  after Time ->
    TimeOutActions
end
```

- Deskriptoren reihum abfragen problematisch, da:
  - umständlich
  - verbraucht unnötig Prozessorzeit (busy waiting)
  - read() i.Allg. blockierend
    (warten auf Ereignis 1, während Ereignis 2 eingetreten ist)

  ⇒ „Sammel-Wartepunkt“ erforderlich

  ⇒ ähnlich zu guarded commands

- Wdh.: Auf mehrere Eingaben warten in CSP bzw. Erlang:

Alternativanweisung in CSP:

```
[ P1 ? variable1 → anweisung1
[] P2 ? variable2 → anweisung2
[] P3 ? variable3 → anweisung3
]
```

Empfang von Nachrichten in Erlang

```
receive
  MessagePattern1 ->
    Actions1;
  MessagePattern2 ->
    Actions2;
   …
  after Time ->
    TimeOutActions
end
```

- In Unix: Systemaufruf select()
- (Alternative Lösung: Verwendung verschiedener Threads)

- Verwendung von select()

```
n = select (maxfd, readset, writeset, exceptset, timeout)
```

  - `readset:` Warten auf Empfang von Informationen
  - `writeset:` Warten auf Sendebereitschaft
  - `exceptset:` Warten auf „Ausnahmebedingung“
  - `maxfd:` Länge der Bitmasken (muss höchste betrachtete Deskriptornummer umfassen)
  - `timeout:` maximale Wartezeit

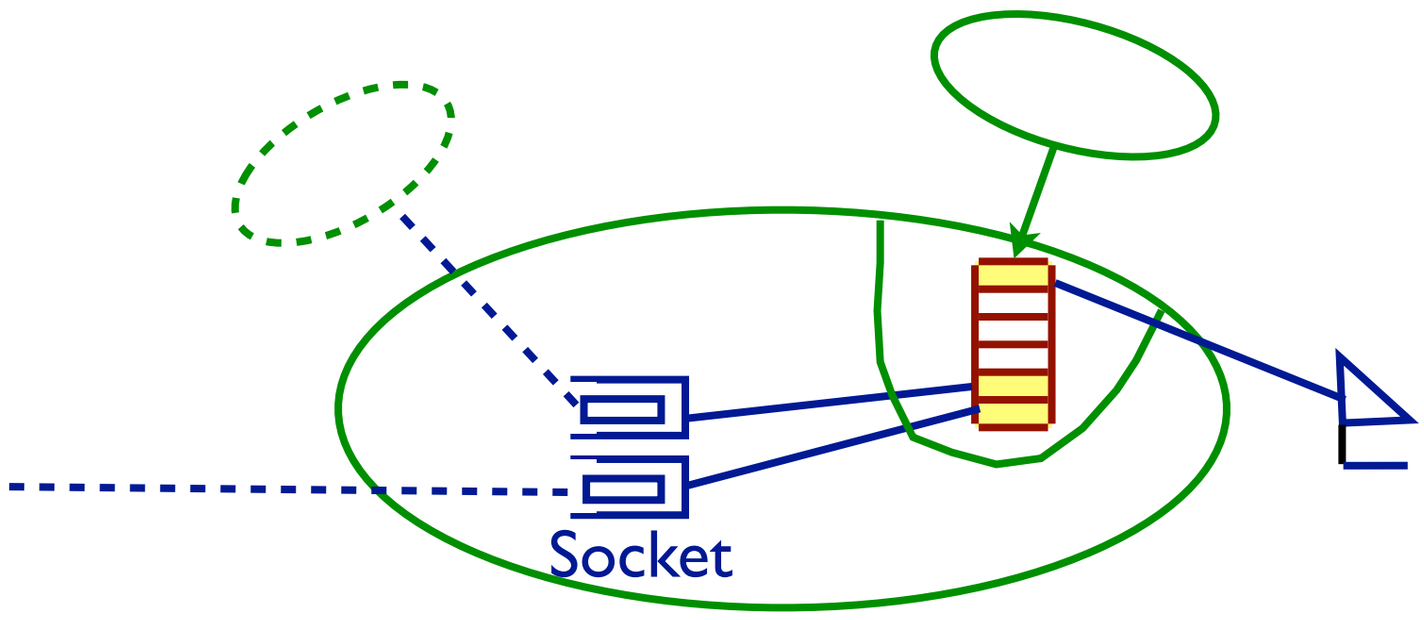
- Angabe der entsprechenden Quellen/Ziele über ihre Deskriptornummer

⇒ als Bitmaske ausgelegt

Beispiel: readset

0	1	2	3	4	5=maxfd-1
1	0	0	0	1	1

⇒ Interesse am Lesen von Deskriptor 0, 4 und 5



- Verwendung von select()

```
n = select (maxfd, readset, writeset, exceptset, timeout)
```

  `readset:` Warten auf Empfang von Informationen

  `writeset:` Warten auf Sendebereitschaft

  `exceptset:` Warten auf „Ausnahmebedingung“

  `maxfd:` Länge der Bitmasken (muss höchste betrachtete Deskriptornummer umfassen)

  `timeout:` maximale Wartezeit

- Angabe der entsprechenden Quellen/Ziele über ihre Deskriptornummer

  ⇒ als Bitmaske ausgelegt

Beispiel: readset

0	1	2	3	4	5=maxfd-1
1	0	0	0	1	1

  ⇒ Interesse am Lesen von Deskriptor 0, 4 und 5

- select() prüft, welche Ereignisse eingetreten sind
  (legt sich andernfalls schlafen, bis ein Ereignis eintritt oder Timeout abläuft)

- Rückgabewerte:

  `n:` Anzahl der eingetretenen Ereignisse

readset

0	1	2	3	4	5=maxfd-1
1	0	0	0	0	1

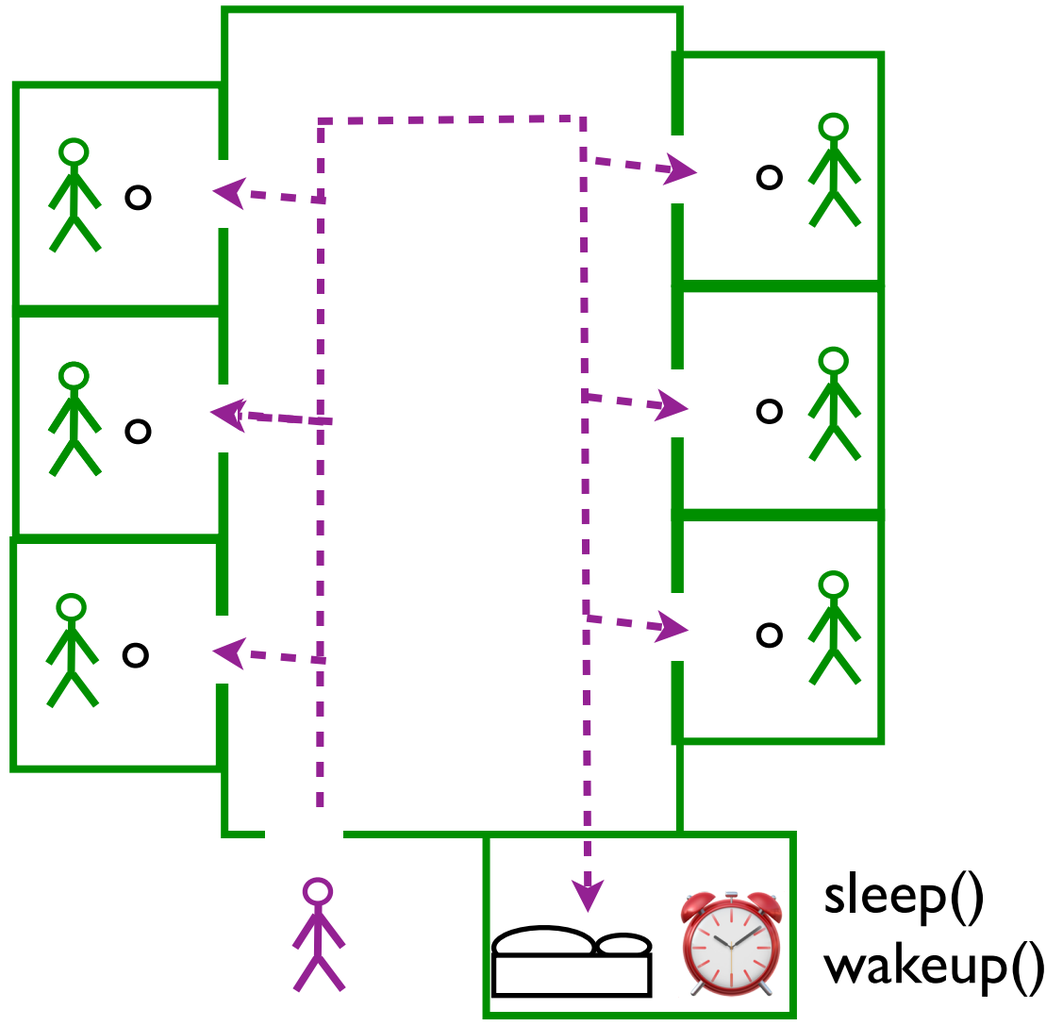
  `readset/writeset/exceptset...` aktualisiert

  ⇒ Lesen von Deskriptor 0 und 5 möglich, nicht von 4

- Implementierung von select()

  - Zunächst überprüfen, ob (ein oder mehrere) Ereignisse vorliegen
    ⇒ sequentiell abfragen
  - Wenn nicht: schlafenlegen

- Korridor-Algorithmus



- Implementierung von select()

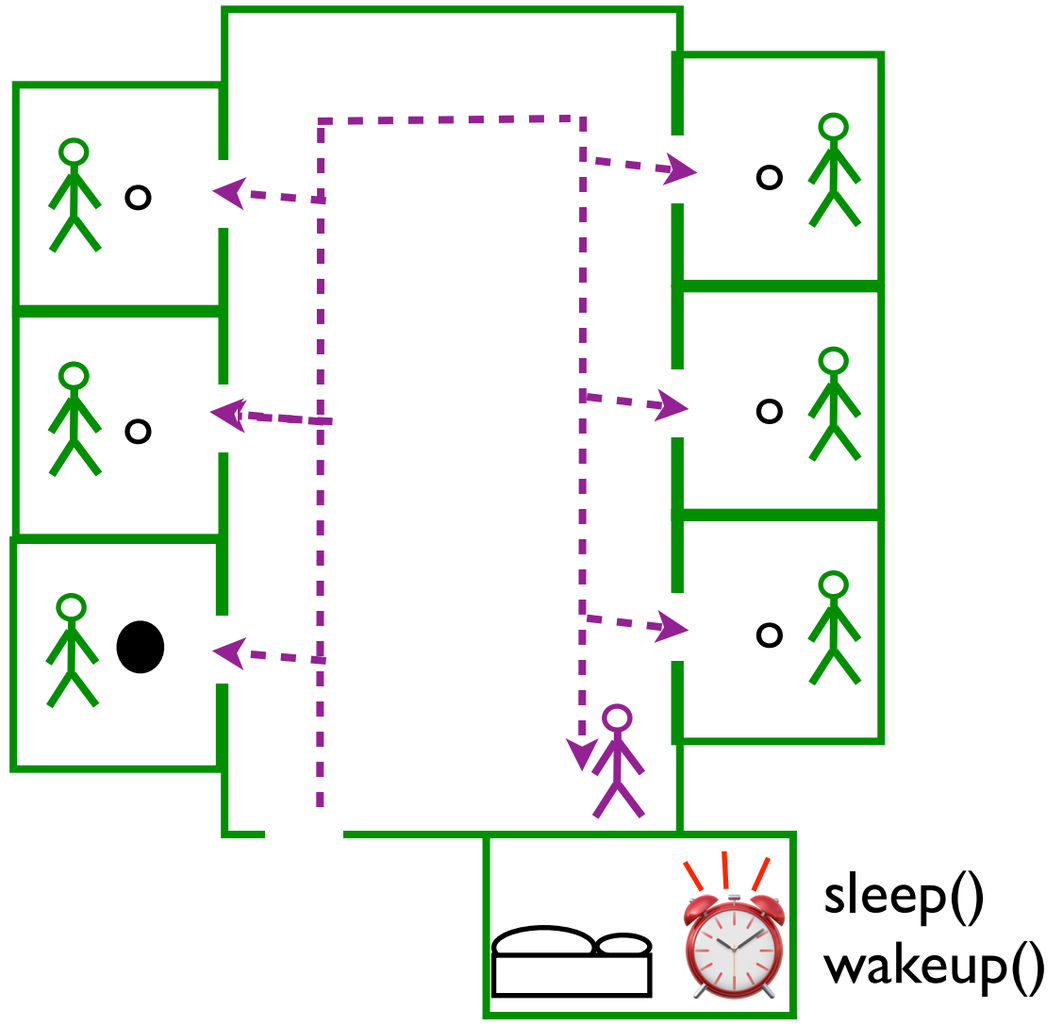
  - Zunächst überprüfen, ob (ein oder mehrere) Ereignisse vorliegen
    ⇒ sequentiell abfragen
  - Wenn nicht: schlafenlegen

    ⇒ Wann?

    ⇒ Wenn letzte Quelle überprüft, kann erste bereits „Auftrag“ haben

    ⇒ „Lost Wakeup“ vermeiden

  - Korridor-Algorithmus



- Implementierung von select()

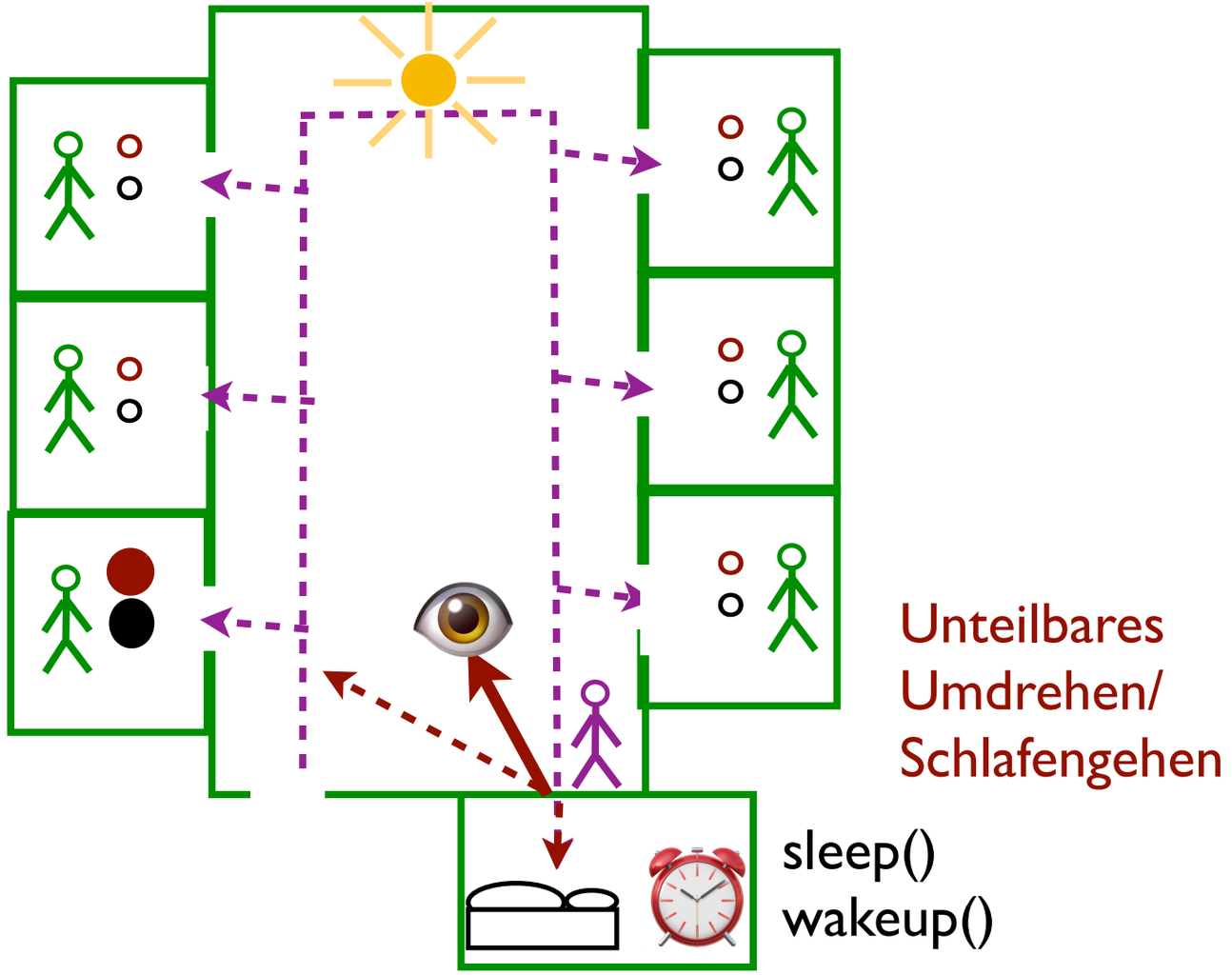
  - Zunächst überprüfen, ob (ein oder mehrere) Ereignisse vorliegen
    ⇒ sequentiell abfragen
  - Wenn nicht: schlafenlegen

    ⇒ Wann?

    ⇒ Wenn letzte Quelle überprüft, kann erste bereits „Auftrag“ haben

    ⇒ „Lost Wakeup“ vermeiden

- Korridor-Algorithmus



Fragen – Teil 4

- Wie arbeitet der *Korridor-Algorithmus* zum Überwachen mehrerer Eingabequellen in etwa?

Zusammenfassung

- Exkurs: Software Transactional Memory (STM)
- Interprozesskommunikation in Unix
  - Über gemeinsamen Adressraum

    ⇒ Synchronisaton über Locking, Semaphore, ...

  - Über gemeinsame Dateien

    ⇒ Synchronisation über File Locking

  - Über Nachrichtenkanäle (asynchroner Nachrichtenaustausch)

    ⇒ implizite Synchronisation

    - Pipes
    - Named Pipes (FIFOs)
    - Sockets

  - Verwaltung mehrerer Kommunikationsbeziehungen (select())

Interprozesskommunikation in Unix – Fragen

1. Was ist *STM (Software Transactional Memory)*?
2. Worin unterscheiden sich die Eigenschaften der folgenden Unix-Mechanismen zur Interprozesskommunikation:
   a) *Pipes*,
   b) *Named Pipes*,
   c) *Sockets*?
3. Wie lässt sich der Zugriff auf Sockets in die generische Systemaufrufschnittstelle zum Zugriff auf Dateien einordnen?
4. Wie können Sockets adressiert werden?
5. Wie arbeitet der *Korridor-Algorithmus* zum Überwachen mehrerer Eingabequellen in etwa?