Work in progress

Dateiverwaltung (2)

Ute Bormann, TI2

2023-10-13

Inhalt

1. Kern-interne Datenstrukturen zur Dateiverwaltung
2. Der Unix Buffer-Cache

Teil 1:
Kern-interne Datenstrukturen zur Dateiverwaltung

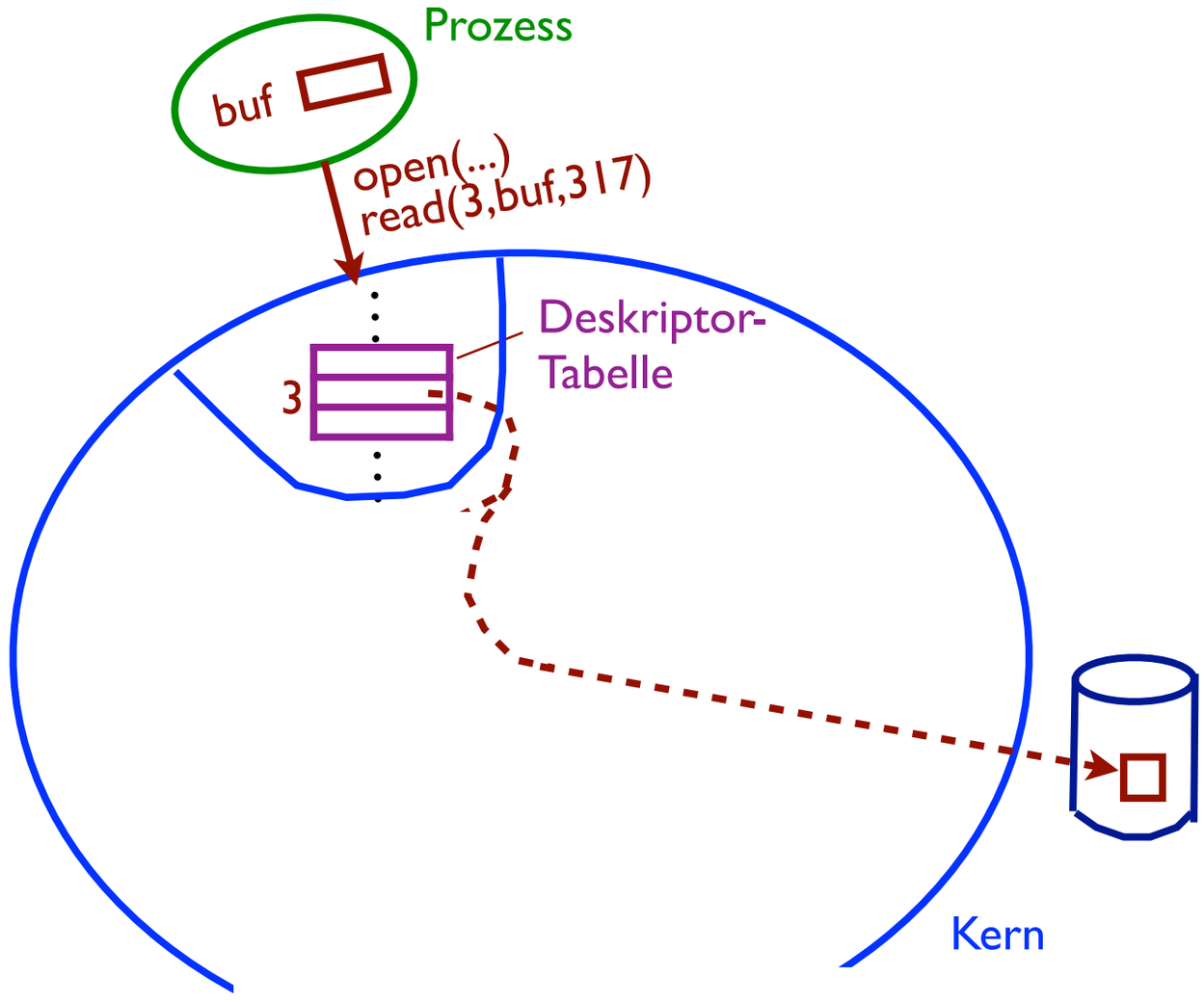
11 Dateiverwaltung (2), Ute Bormann

Kern-interne Datenstrukturen zur Dateiverwaltung

⇒ Verwaltung geöffneter Dateien

Deskriptor-Tabellen (Prozess-spezifisch)

- Eintrag für jeden erhaltenen Deskriptor (indiziert über File Descriptor (fd))
- Verweis auf Eintrag in globaler File-Tabelle



Kern-interne Datenstrukturen zur Dateiverwaltung

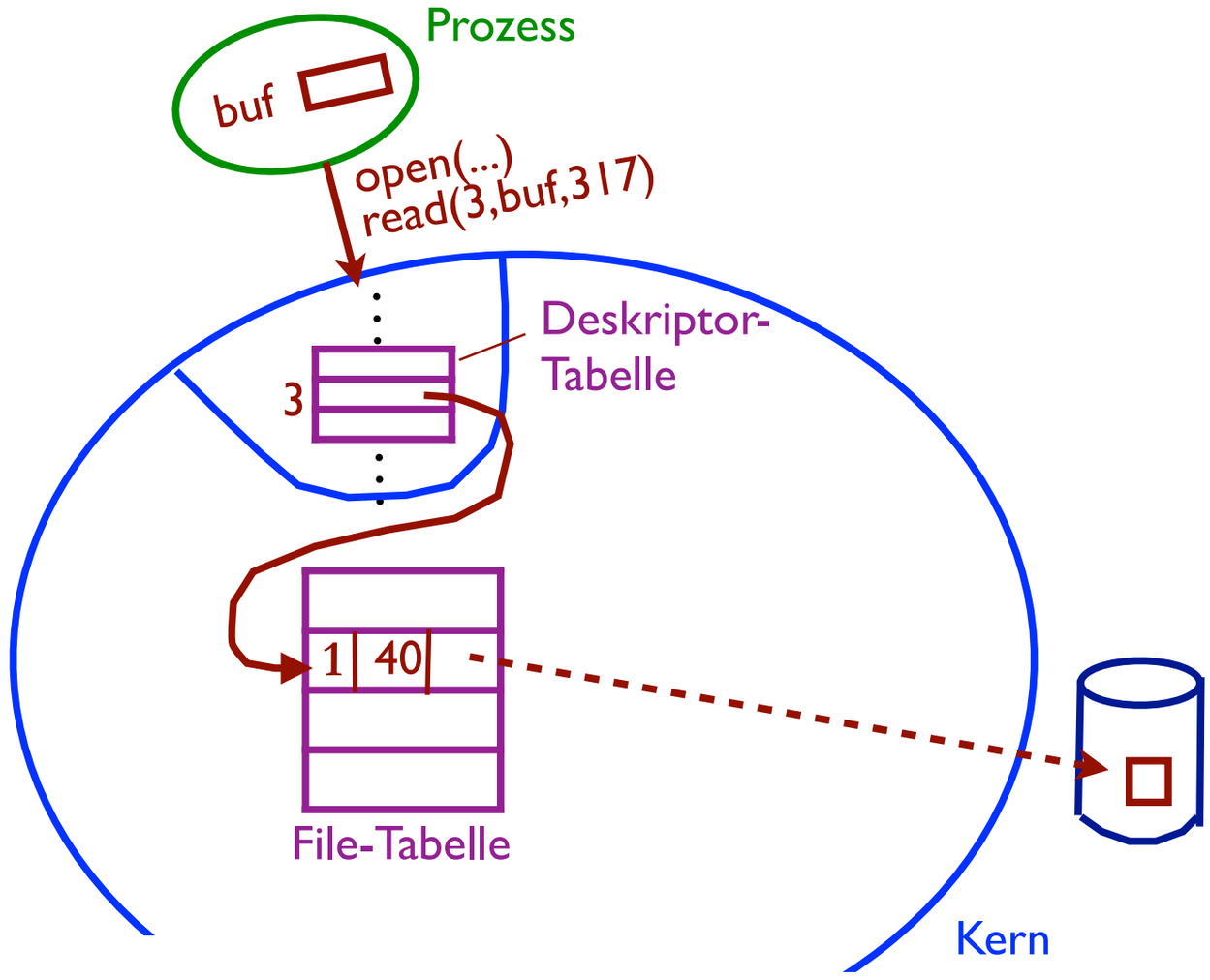
⇒ Verwaltung geöffneter Dateien

Deskriptor-Tabellen (Prozess-spezifisch)

- Eintrag für jeden erhaltenen Deskriptor (indiziert über File Descriptor (fd))
- Verweis auf Eintrag in globaler File-Tabelle

File-Tabelle (System-global)

- Eintrag für jede Datei-Öffnung
- Zähler für Veweise aus Deskriptor-Tabellen (=1, wenn nicht duplizierte Deskriptoren durch fork(), dup())
- Aktuelle Schreib-/Leseposition in Datei
- Veweis auf Eintrag in Inode-Tabelle



Kern-interne Datenstrukturen zur Dateiverwaltung

⇒ Verwaltung geöffneter Dateien

Deskriptor-Tabellen (Prozess-spezifisch)

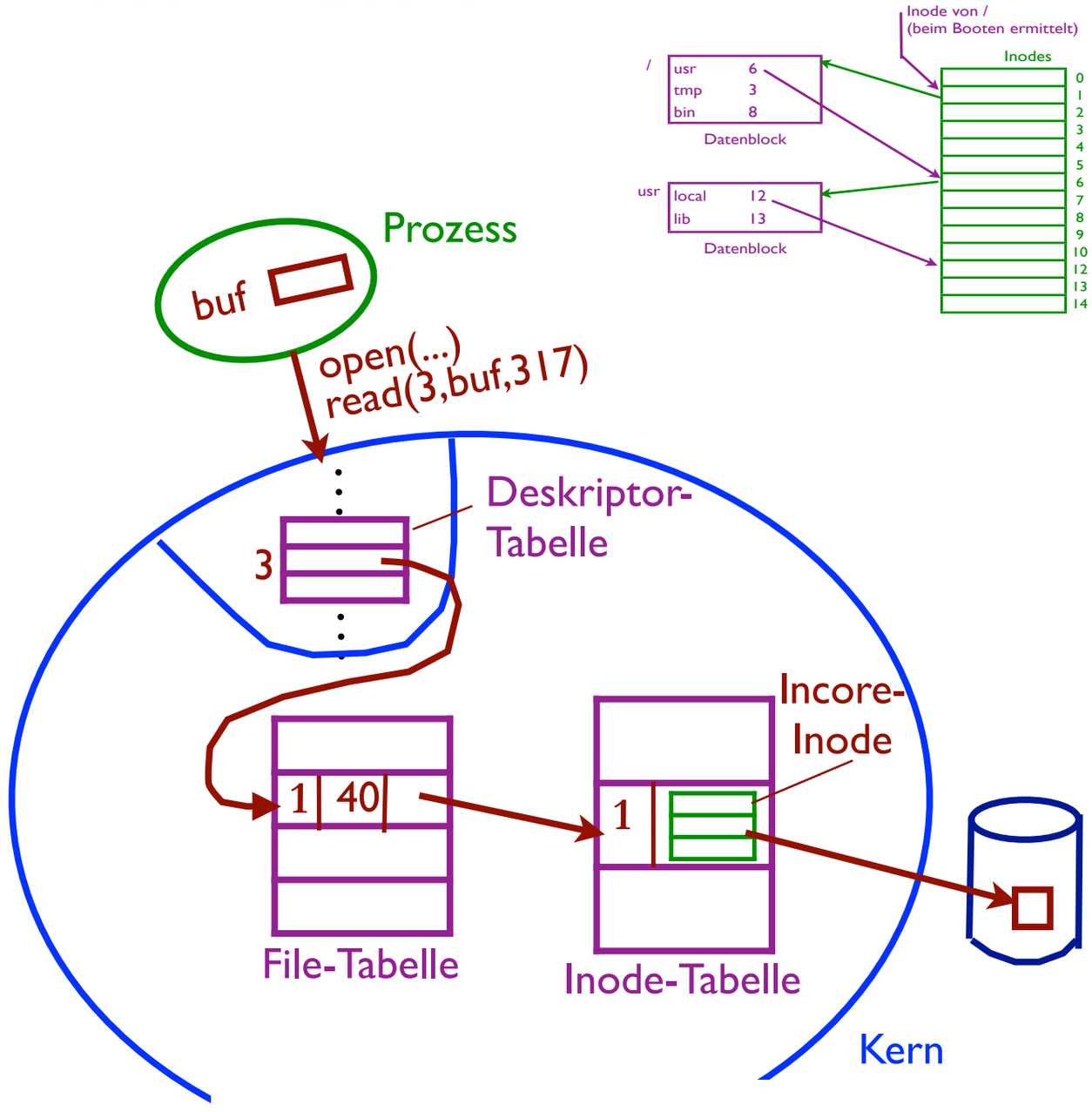
- Eintrag für jeden erhaltenen Deskriptor (indiziert über File Descriptor (fd))
- Verweis auf Eintrag in globaler File-Tabelle

File-Tabelle (System-global)

- Eintrag für jede Datei-Öffnung
- Zähler für Veweise aus Deskriptor-Tabellen (=1, wenn nicht duplizierte Deskriptoren durch fork(), dup())
- Aktuelle Schreib-/Leseposition in Datei
- Veweis auf Eintrag in Inode-Tabelle

Inode-Tabelle (System-global)

- Inode-Kopien der geöffneten Dateien (Incore-Inodes)
  ⇒ schnellerer Zugriff (keine Plattenzugriffe)
- Eintrag: Gerätenummer, Inode-Nummer, Kopie des Platten-Inodes, Zähler für Verweise aus File-Tabelle



Kern-interne Datenstrukturen zur Dateiverwaltung

⇒ Verwaltung geöffneter Dateien

Deskriptor-Tabellen (Prozess-spezifisch)

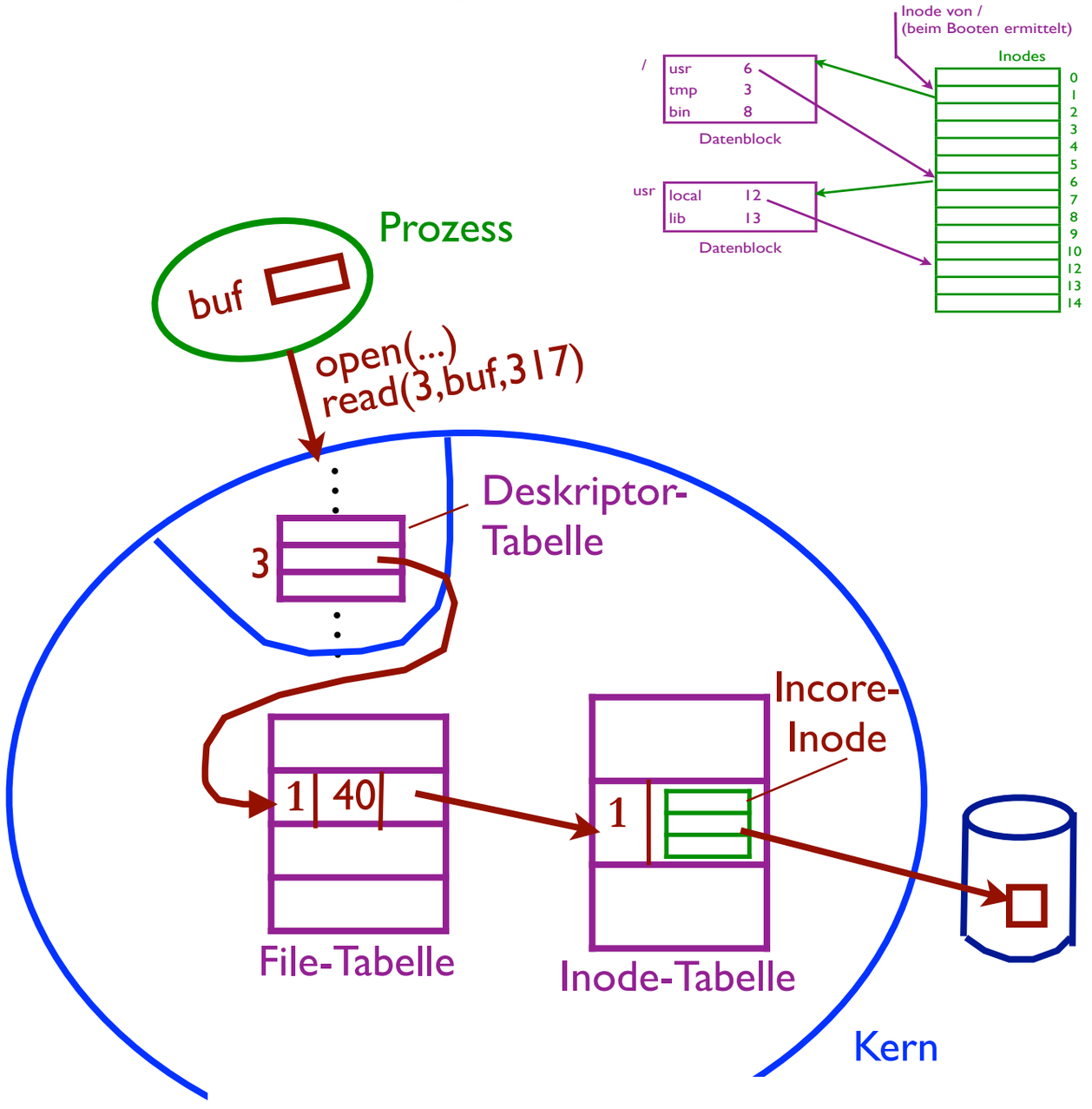
- Eintrag für jeden erhaltenen Deskriptor (indiziert über File Descriptor (fd))
- Verweis auf Eintrag in globaler File-Tabelle

File-Tabelle (System-global)

- Eintrag für jede Datei-Öffnung
- Zähler für Veweise aus Deskriptor-Tabellen (=1, wenn nicht duplizierte Deskriptoren durch fork(), dup())
- Aktuelle Schreib-/Leseposition in Datei
- Veweis auf Eintrag in Inode-Tabelle

Inode-Tabelle (System-global)

- Inode-Kopien der geöffneten Dateien (Incore-Inodes) ⇒ schnellerer Zugriff (keine Plattenzugriffe)
- Eintrag: Gerätenummer, Inode-Nummer, Kopie des Platten-Inodes, Zähler für Verweise aus File-Tabelle



Aktualisierung der Strukturen bei:
open(): Anlegen...
close(): Löschen...
read()/write()/lseek():
aktuelle Position verschieben

Kleine Aufgabe

Worin unterscheiden sich die beiden folgenden Programmauszüge hinsichtlich der eingelesenen Informationen in buf1 und buf2?

A: …

```
pid=fork();
switch (pid) {
  case -1: …error …
  case 0:
    fd=open (“/bla”,..);
    char buf1[100];
    read (fd, buf1, 100);
    break;
  default:
    fd=open (“/bla”,..);
    char buf2[100];
    read (fd, buf2, 100);
}
...
```

B: …

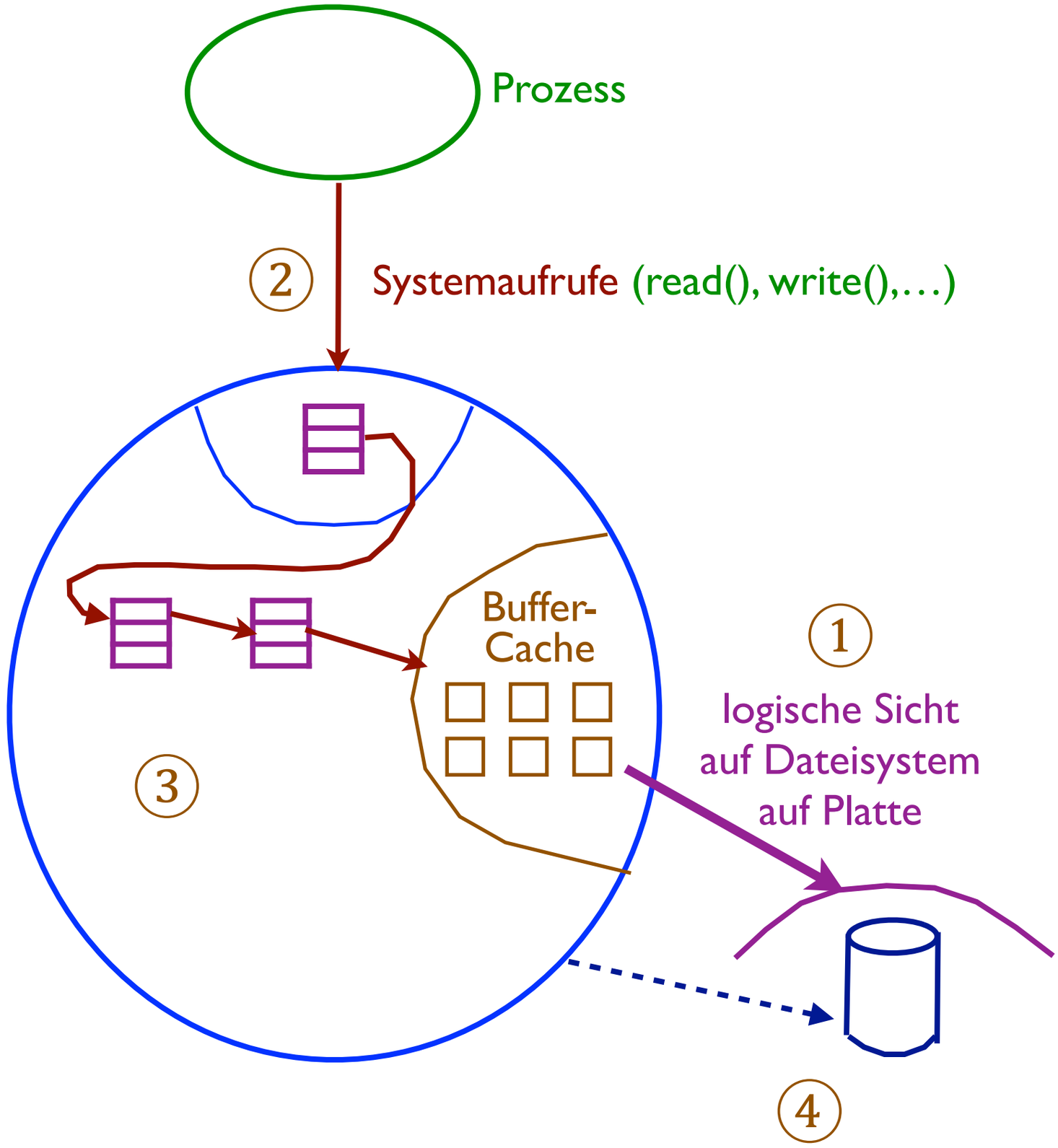
```
fd=open (“/bla”,..);
pid=fork();
switch (pid) {
  case -1: …error …
  case 0:
    char buf1[100];
    read (fd, buf1, 100);
    break;
  default:
    char buf2[100];
    read (fd, buf2, 100);
}
...
```

Fragen – Teil 1

- Wozu werden die folgenden Kern-internen Datenstrukturen verwendet?
  a) Deskriptor-Tabelle
  b) File-Tabelle
  c) Inode-Tabelle

Teil 2:
Der Unix Buffer-Cache

Überblick Dateiverwaltung



Der Unix Buffer-Cache

- Gelesene/zu schreibende Plattenblöcke (Inodes, Inhalt, ...) im Kernadressraum „zwischenpuffern“

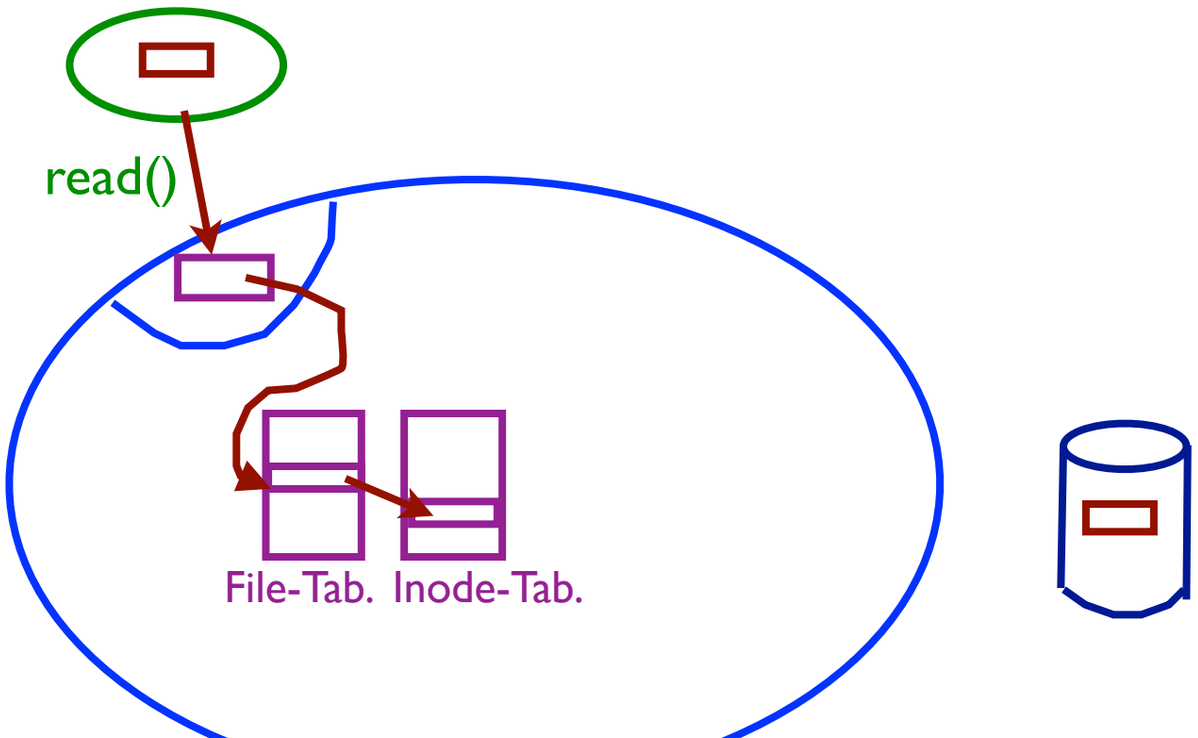
  ⇒ Buffer-Cache

  - Pufferung während Ein-/Ausgabe

    ⇒ vom Prozessadressraum entkoppelt

  - Cache-Funktionalität

    ⇒ Mehrfachzugriff möglich ohne weitere Plattenzugriffe



Der Unix Buffer-Cache

- Gelesene/zu schreibende Plattenblöcke (Inodes, Inhalt, ...) im Kernadressraum „zwischenpuffern“

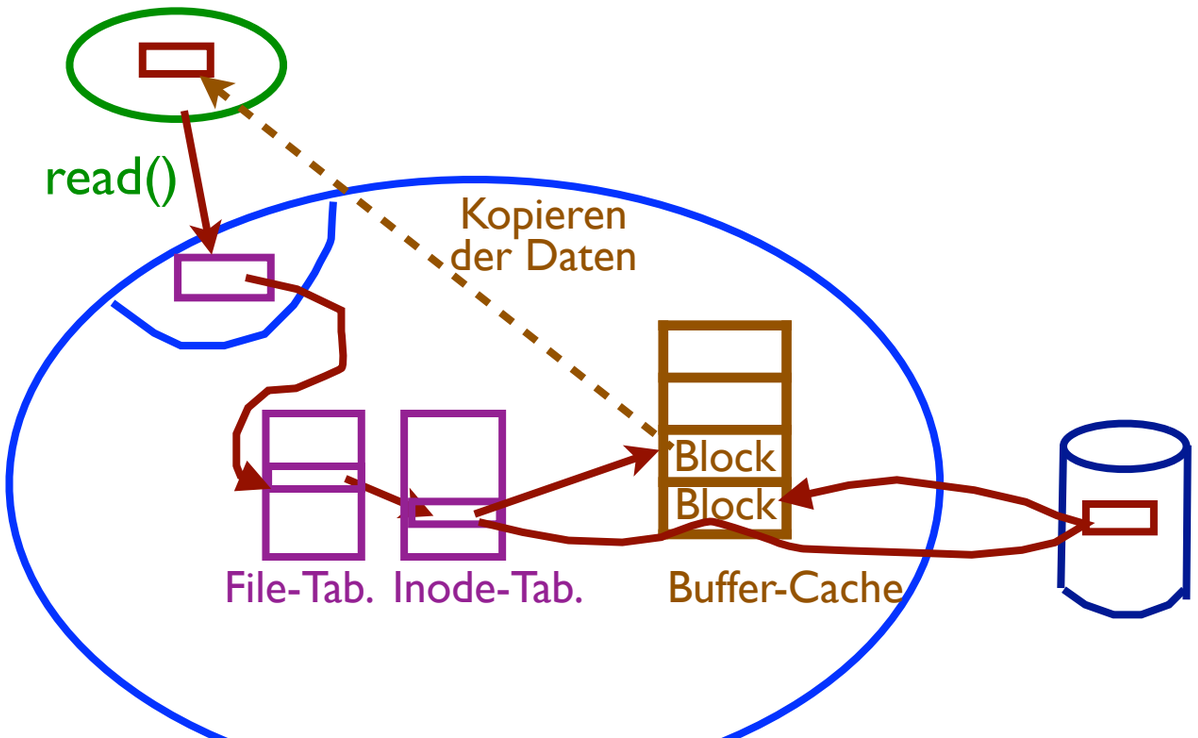
  ⇒ Buffer-Cache

  - Pufferung während Ein-/Ausgabe

    ⇒ vom Prozessadressraum entkoppelt

  - Cache-Funktionalität

    ⇒ Mehrfachzugriff möglich ohne weitere Plattenzugriffe



Der Unix Buffer-Cache

- Gelesene/zu schreibende Plattenblöcke (Inodes, Inhalt, ...) im Kernadressraum „zwischenpuffern“

  ⇒ Buffer-Cache

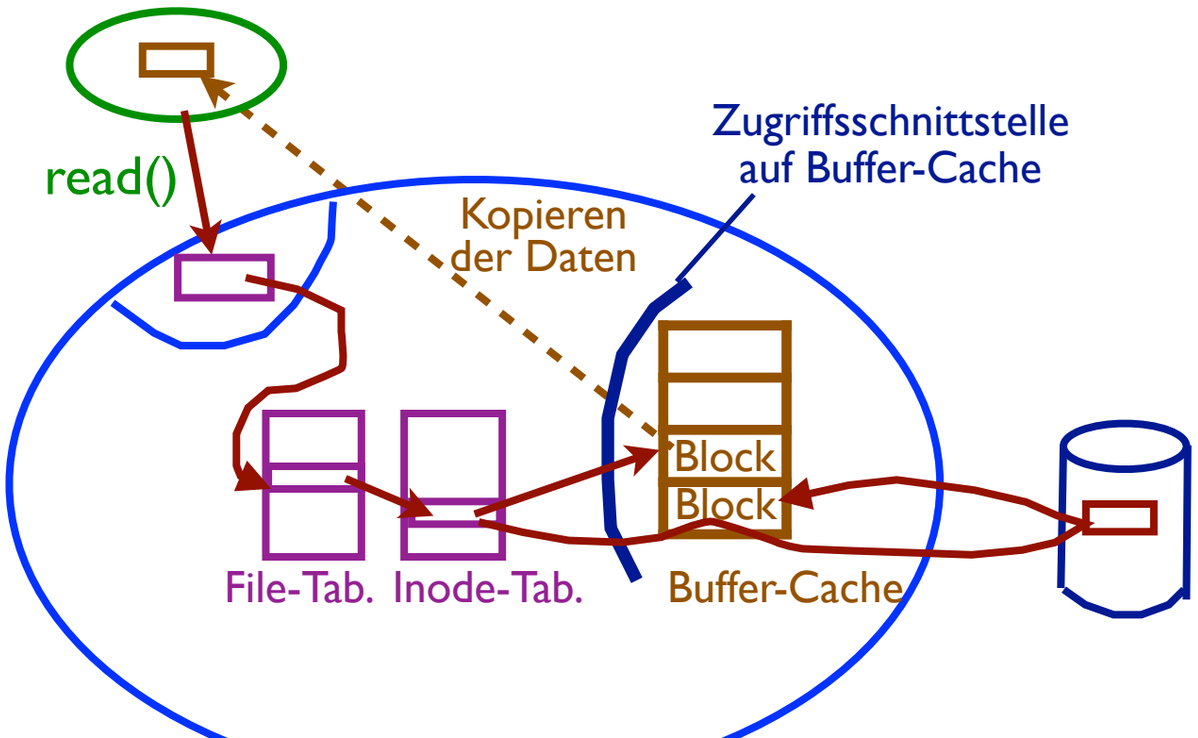
  - Pufferung während Ein-/Ausgabe

    ⇒ vom Prozessadressraum entkoppelt

  - Cache-Funktionalität

    ⇒ Mehrfachzugriff möglich ohne weitere Plattenzugriffe

- Kerninterne Realisierung von read()/write(): Zugriff über Schnittstelle zu Buffer-Cache:



Zugriffsoperationen auf Buffer-Cache

Puffer-Belegung

bread (dev, blkno) Block lesen ⇒ bei read(), z.T. write()

getblk (dev, blkno) Puffer reservieren (später unter (dev,blkno) ablegen)
⇒ bei write()

⇒ liefert Buffer Pointer (bp)

Zugriffsoperationen auf Buffer-Cache

Puffer-Belegung

bread (dev, blkno) Block lesen ⇒ bei read(), z.T. write()

getblk (dev, blkno) Puffer reservieren (später unter (dev,blkno) ablegen)
⇒ bei write()

⇒ liefert Buffer Pointer (bp)

Puffer-Freigabe

brelse (bp) Puffer freigeben ⇒ bei read()

bwrite (bp) Block schreiben ⇒ sofort auf Platte retten, bei write()

Zugriffsoperationen auf Buffer-Cache

Puffer-Belegung

bread (dev, blkno) Block lesen ⇒ bei read(), z.T. write()

getblk (dev, blkno) Puffer reservieren (später unter (dev,blkno) ablegen)
⇒ bei write()

⇒ liefert Buffer Pointer (bp)

Puffer-Freigabe

brelse (bp) Puffer freigeben ⇒ bei read()

bwrite (bp) Block schreiben ⇒ sofort auf Platte retten, bei write()

bdwrite (bp) Block schreiben ⇒ erst bei neuer Belegung auf Platte retten
(ggf. vorher durch sync())

Zugriffsoperationen auf Buffer-Cache

Puffer-Belegung

bread (dev, blkno) Block lesen ⇒ bei read(), z.T. write()

getblk (dev, blkno) Puffer reservieren (später unter (dev,blkno) ablegen)
⇒ bei write()

⇒ liefert Buffer Pointer (bp)

Puffer-Freigabe

brelse (bp) Puffer freigeben ⇒ bei read()

bwrite (bp) Block schreiben ⇒ sofort auf Platte retten, bei write()

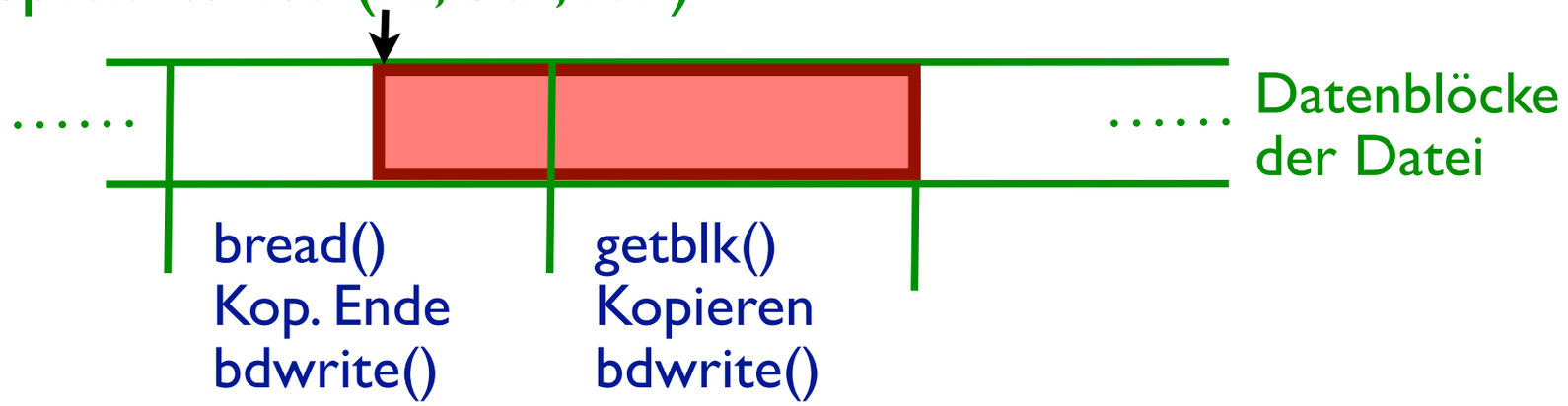
bdwrite (bp) Block schreiben ⇒ erst bei neuer Belegung auf Platte retten (ggf. vorher durch sync())

- Für alle Plattenblöcke des Dateisystems genutzt:
  - Datenblöcke (bei read(), write(),...)
  - Inodeblöcke (bei open(), close(),...)
  - Indirektblöcke (ggf. bei read(), write(), um Nummer des Datenblocks zu ermitteln)

Realisierung read()/write()

1) Belegung der erforderlichen Puffer für Schreib-/Lesebereich
2) Umkopieren: User-Adressraum ⟷Buffer-Cache
3) Freigabe der Puffer

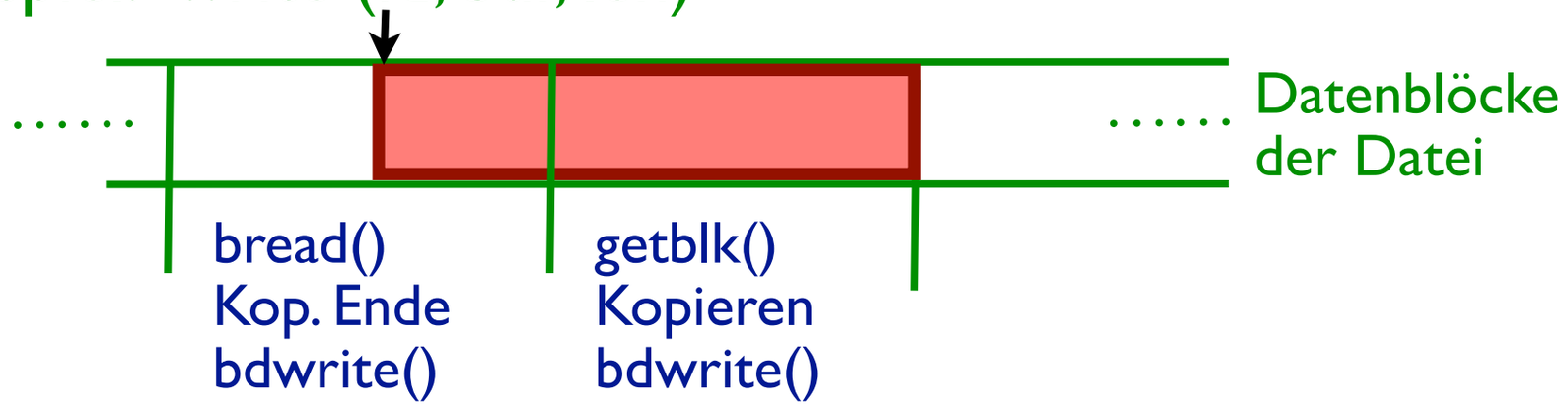
Beispiel: write (fd, buf, len)



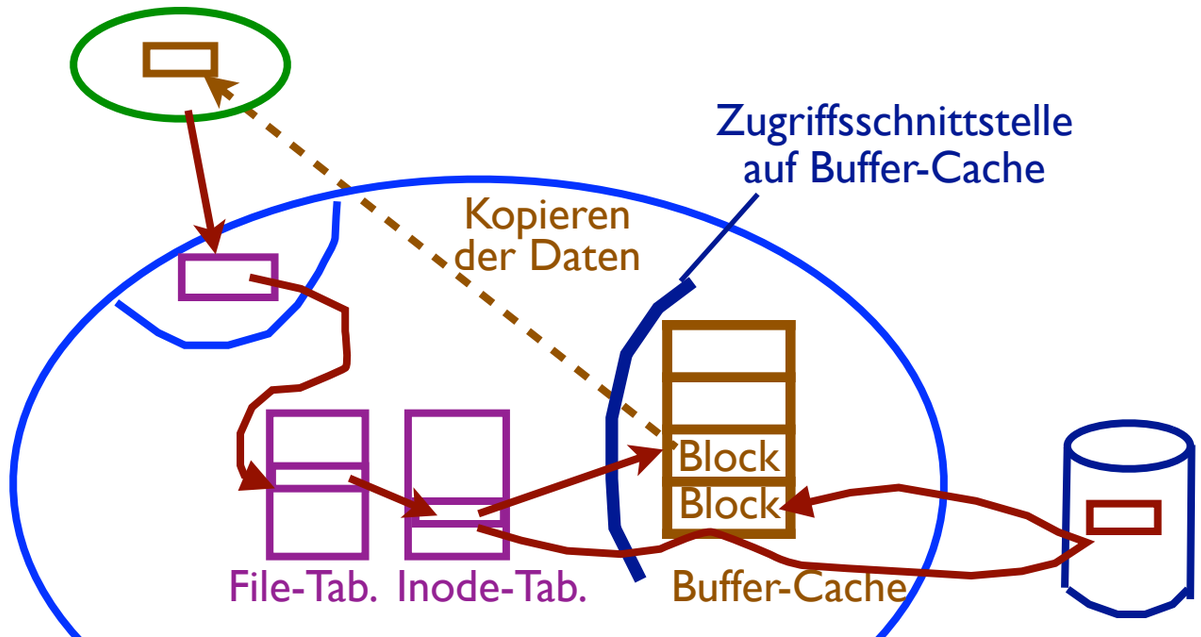
Realisierung read()/write()

1) Belegung der erforderlichen Puffer für Schreib-/Lesebereich
2) Umkopieren: User-Adressraum ⟷Buffer-Cache
3) Freigabe der Puffer

Beispiel: write (fd, buf, len)



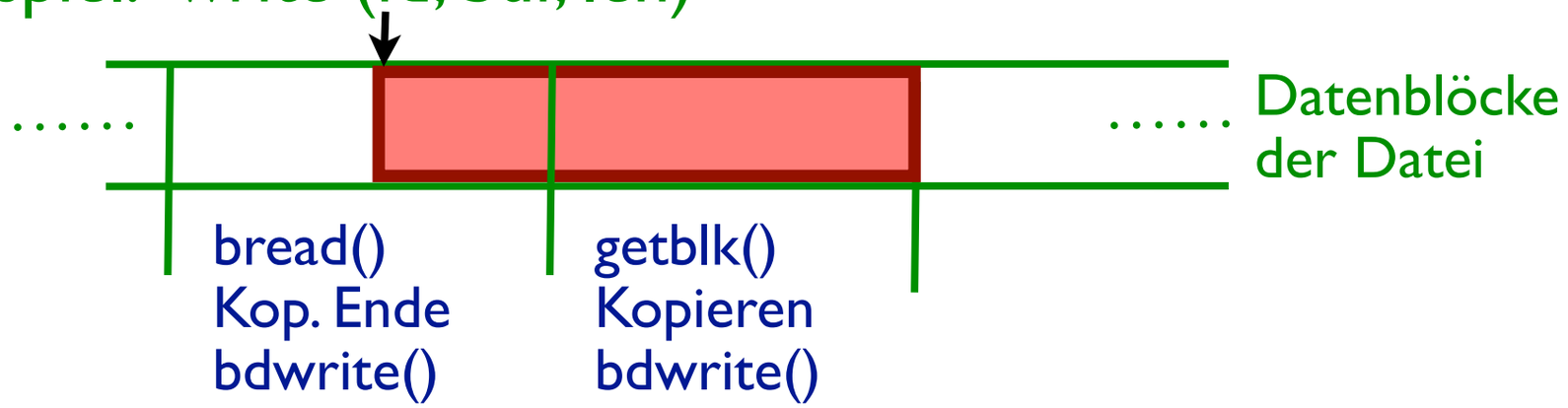
- Weitere Schreib-/Leseoperationen auf diesem Bereich?
  ⇒ keine Plattenzugriffe nötig, wenn Puffer noch nicht wieder belegt (Inhalt unverändert)
  ⇒ Reclaim (Cache-Funktionalität)



Realisierung read()/write()

1) Belegung der erforderlichen Puffer für Schreib-/Lesebereich
2) Umkopieren: User-Adressraum ⟷Buffer-Cache
3) Freigabe der Puffer

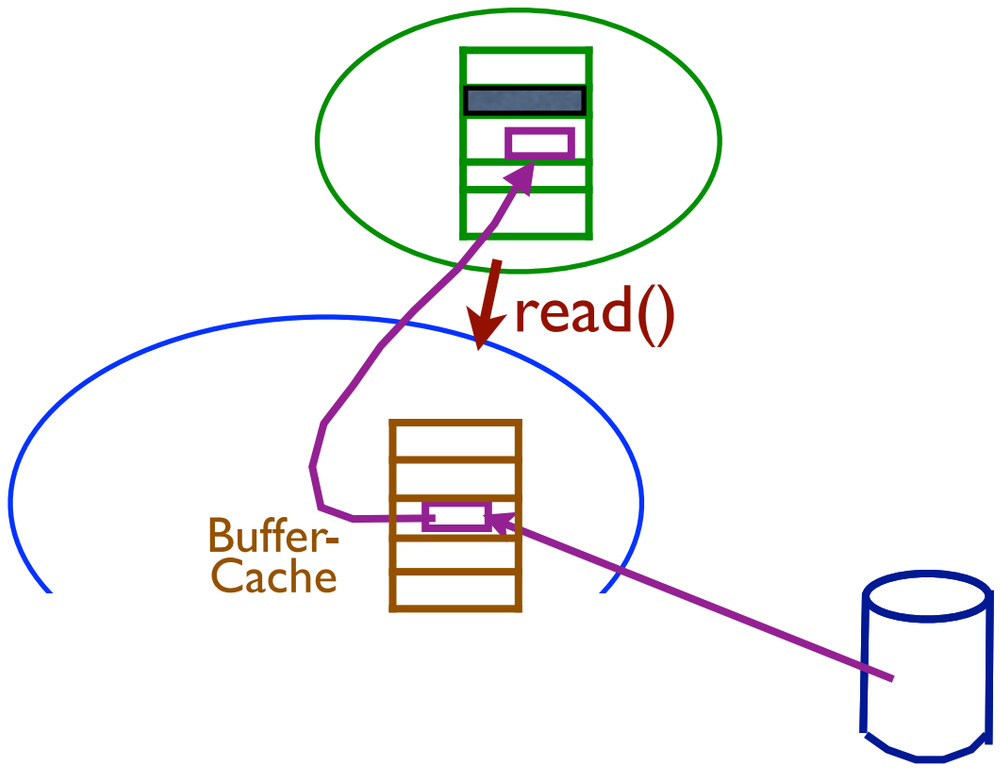
Beispiel: write (fd, buf, len)



- Weitere Schreib-/Leseoperationen auf diesem Bereich?
  ⇒ keine Plattenzugriffe nötig, wenn Puffer noch nicht wieder belegt (Inhalt unverändert)
  ⇒ Reclaim (Cache-Funktionalität)
- Neu-Belegung von Puffern (klassische Form)
  ⇒ Verdrängungsalgorithmus ⇒ z.B. FIFO / Least-Recently-Used (LRU)
  (heute eher über normales Paging realisiert)

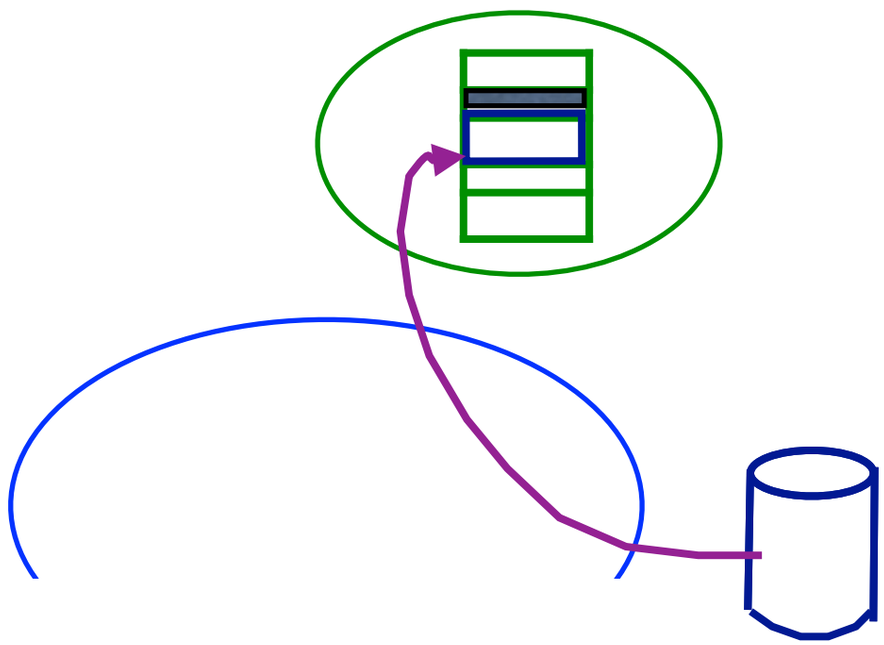
Virtueller Adressraum vs. Dateien

- Virtueller Adressraum enthält Text-, Daten-, Stacksegment(e) des Prozesses (initial: a.out-Datei, brk())
- Zugriff auf weitere (langlebige) Informationen (Dateien) über read()/write()



Zugriff auf Datensegment	Zugriff auf Datei (read()/write())
Bei Mehrfachzugriff zugehörige Prozess-Page u.U. bereits im Hauptspeicher enthalten	Bei Mehrfachzugriff entspr. Buffer-Cache-Block u.U. im Hauptspeicher enthalten
Dann Adressumwandlung direkt durch MMU	–
Wenn (noch) nicht im Hauptspeicher: Page Fault ⇒ dann Kontextwechsel in Kern	Zugriff über Systemaufruf nötig ⇒ immer Kontextwechsel in Kern
Wenn kein Reclaim möglich ⇒ Plattenzugriff	Wenn kein Reclaim möglich ⇒ Plattenzugriff
	Kopieren von Buffer-Cache-Block in User-Adressraum

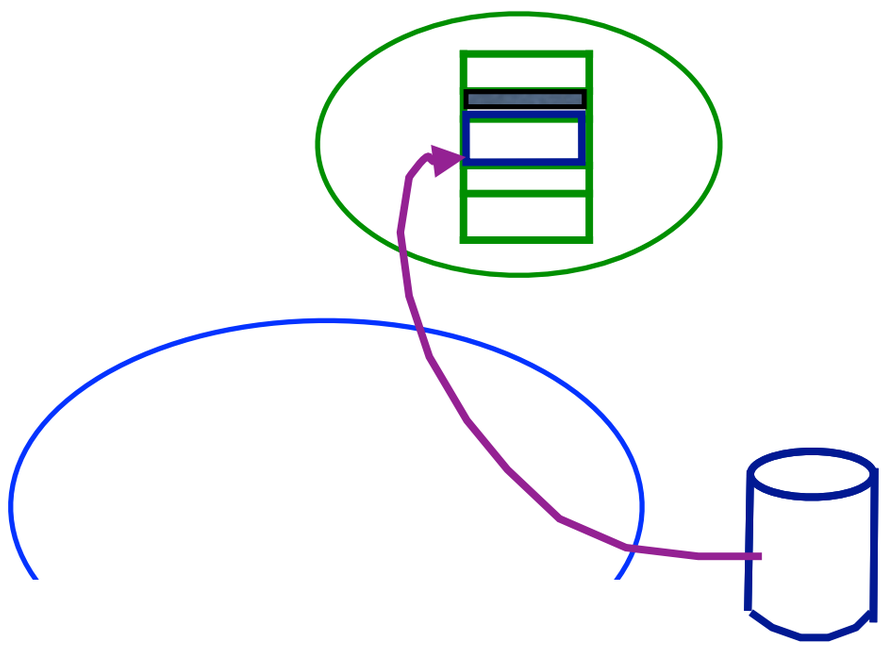
- Alternativ: Dateien in virtuellen Adressraum des Prozesses einblenden:
  mmap (addr, len, prot, flags, fd, off)



- len Bytes ab Position off (Pagegrenze!) von Datei fd
  ab Adresse addr (Pagegrenze!) in Prozessadressraum einblenden;
- prot ⇒ erlaubte Operationen: lesen/schreiben/...
- flags ⇒ Kopie/Original

⇒ „Direkter“ Zugriff über Paging (evtl. Plattenzugriff erforderlich)

- Alternativ: Dateien in virtuellen Adressraum des Prozesses einblenden:
  mmap (addr, len, prot, flags, fd, off)



  - len Bytes ab Position off (Pagegrenze!) von Datei fd
    ab Adresse addr (Pagegrenze!) in Prozessadressraum einblenden;
  - prot ⇒ erlaubte Operationen: lesen/schreiben/...
  - flags ⇒ Kopie/Original

  ⇒ „Direkter“ Zugriff über Paging (evtl. Plattenzugriff erforderlich)

- Gibt auch read()-Varianten mit automatischem „mmap“
- Heute z.T. blockweises Lesen über Bibliothek

Zusammenfassung

- Kerninterne Datenstrukturen zur Dateiverwaltung:
  - Prozess-spezifische Deskriptor-Tabellen
  - System-globale File-Tabelle
  - System-globale Inode-Tabelle
- Unix Buffer-Cache
- Virtuelle Adressräume vs. Dateien

Dateiverwaltung 2 – Fragen

1. Wozu werden die folgenden Kern-internen Datenstrukturen verwendet?
   a) Deskriptor-Tabelle
   b) File-Tabelle
   c) Inode-Tabelle

2. Welche Aufgaben hat der *Buffer Cache* in Unix?

3. Welche Vorteile bietet es, Dateien mit dem Unix-Systemaufruf `mmap()` in den virtuellen Adressraum eines Prozesses abzubilden?