CRASHKURS · PRÜFUNGSVORBEREITUNG

Einführung in die Informatik 2

Betriebssysteme und Rechnernetze — in einfachen Worten erklärt, Schritt für Schritt, ganz ohne Vorwissen. Zum Üben gibt es das interaktive Lernset (EI2_Lernset.html) mit allen Prüfungsfragen als Karteikarten.

1 · Betriebssystem & Prozesse
2 · Scheduling
3 · Synchronisation
4 · Speicher
5 · Dateien
6 · Ein-/Ausgabe
7 · Netz-Grundlagen
8 · Sicherungsschicht
9 · Vermittlung & IP
10 · TCP
11 · Anwendungsschicht

Inhalt abgeleitet aus den fünf offiziellen Aufgabenblättern (SoSe 2026, Prof. Dr. Djanatliev, TH Ingolstadt, Studiengang KI), den vollständigen Vorlesungsfolien (Kap. 1–11) und der Probeklausur/Klausur SoSe 2023 (Prof. Tiedemann). Laut Professor sind die fünf Blätter die Wissensgrundlage; ergänzt um Folien- und Probeklausurstoff zur vollen Absicherung. Prüfungsformat lt. Modulhandbuch: schriftlich, 90 Minuten, Hilfsmittel: ein handbeschriebenes DIN-A4-Blatt und ein nicht-programmierbarer Taschenrechner.

So benutzt du diesen Crashkurs

Du brauchst **kein Vorwissen**. Arbeite jedes Thema in dieser Reihenfolge durch:

1. **Lies den blauen Einführungskasten** — er sagt dir in zwei Sätzen, worum es geht und warum.
2. **Schau dir Begriffe, Regeln und das Schema an** — das ist genau das, was du in der Klausur anwendest.
3. **Rechne das Beispiel selbst nach** — nicht nur lesen, sondern mit Stift mitmachen.
4. **Übe mit dem Lernset** (Datei **EI2_Lernset.html**, Menüpunkt „Übungskarten“): dort sind alle Prüfungsfragen der Übungsblätter als Karteikarten mit Antworten — jede Karte verlinkt zurück auf das passende Thema in diesem Crashkurs.

Zum Schluss überträgst du die **Formeln und Merksätze** per Hand auf dein DIN-A4-Blatt — das Abschreiben ist schon halbes Lernen.

Inhalt

Gut zu wissen: Die Klausur besteht etwa zur Hälfte aus Betriebssystemen (Themen 1–6) und zur Hälfte aus Rechnernetzen (Themen 7–11). Lass keinen Teil weg. Viele Aufgaben sind reine „Schema-F“-Rechnungen: Wer das Schema kennt, bekommt sichere Punkte.

1 Betriebssystem-Grundlagen & Prozesse

Blatt 1 + 2

Was ein Betriebssystem tut · Kernel- und User-Modus · Prozesse und ihre Zustände

Worum geht es? Ein **Betriebssystem (BS)** ist die Software-Schicht zwischen der Hardware (Prozessor, Speicher, Festplatte) und deinen Programmen. Es sorgt dafür, dass viele Programme gleichzeitig laufen können, ohne sich gegenseitig zu stören. Die Fragen dazu sind meist „Wissensfragen" — kurze, sichere Punkte, wenn du die Antworten auswendig kannst.

Analogie: Stell dir das BS als **Hausverwaltung** eines großen Mietshauses vor. Die Mieter (Programme) wollen Strom, Wasser, Aufzug (Hardware). Die Verwaltung teilt diese knappen Dinge gerecht zu, verhindert Streit, sperrt fremde Wohnungen ab und ist die einzige Stelle, die an die Sicherungskästen darf.

Die vier Grundaufgaben eines Betriebssystems (auswendig!)

1. **Komplexität verbergen (Hardware-Abstraktion):** Es versteckt die komplizierte Hardware hinter einfachen Befehlen (du sagst „Datei speichern", nicht „schreibe auf Sektor 4711").
2. **Ressourcen (Betriebsmittel) verwalten:** Es teilt Prozessor(en), Haupt- und Sekundärspeicher, E/A-Geräte und Rechenzeit unter den Programmen auf.
3. **Schutzstrategien verfolgen:** Es verhindert bei der Ressourcenvergabe, dass ein Programm auf Speicher, Geräte oder Dateien eines anderen unberechtigt zugreift.
4. **Schnittstellen bereitstellen:** Bedienwege für Nutzer und Programme — Benutzerschnittstelle (Shell/CLI bzw. GUI) und Programmierschnittstelle (API, über System Calls).

Wichtige Begriffspaare

Begriff	Einfach erklärt
Multiprogramming	Mehrere Programme liegen gleichzeitig im Speicher; die CPU springt zwischen ihnen, damit sie nie unbeschäftigt ist. *Ziel: CPU gut auslasten.*
Timesharing	Erweiterung davon: mehrere *Benutzer* arbeiten gleichzeitig; jeder bekommt schnell abwechselnd CPU-Zeit und hat den Eindruck, der Rechner gehöre ihm allein. *Ziel: kurze Antwortzeit.*
Programm	Passiv: nur eine Datei auf der Platte (ein „Rezept").
Prozess	Aktiv: ein Programm in Ausführung, mit aktuellem Zustand (Befehlszähler, Speicher, Daten). Das „Kochen nach dem Rezept".
reale Adresse	Echte, feste Position im physischen Hauptspeicher (RAM).
virtuelle Adresse	Adresse aus der „Wunschwelt" des Prozesses; die MMU (Hardware) rechnet sie in eine reale Adresse um. So glaubt jeder Prozess, er hätte den ganzen Speicher für sich.

Kernel-Modus vs. User-Modus

Die CPU kennt zwei Betriebsarten:

- **User-Modus:** Normale Programme laufen hier. Gefährliche Befehle (direkt auf Geräte oder fremden Speicher zugreifen) sind **verboten**.
- **Kernel-Modus:** Nur der BS-Kern läuft hier und darf alles.

Warum sind E/A-Befehle privilegiert (nur Kernel)? Geräte sind *gemeinsam genutzt*. Dürfte jedes Programm direkt darauf zugreifen, käme es zu Chaos, Datenverlust und Sicherheitslücken. Deshalb entscheidet eine zentrale Instanz (der Kern), wer wann darf.

System Call: die einzige „Tür" vom User- in den Kernel-Modus. Das Programm bittet das BS um einen Dienst (z. B. „Datei lesen"). Technisch ist es ein Software-Interrupt: kurz in den Kernel-Modus wechseln, Dienst ausführen, zurück in den User-Modus.

Merksatz: User-Modus = darf wenig (sicher) · Kernel-Modus = darf alles · System Call = die kontrollierte Tür dazwischen.

Prozesszustände: das 5- und das 7-Zustands-Modell

Ein Prozess ist nicht ständig am Rechnen. Er wechselt zwischen Zuständen:

Zustand	Bedeutung
READY (bereit)	Will rechnen, wartet aber, dass die CPU frei wird.
RUNNING (rechnend)	Hat die CPU gerade. Es kann immer nur **einer** rechnen (pro CPU-Kern).
BLOCKED (blockiert)	Wartet auf ein Ereignis (z. B. Festplatte fertig). Kann erst weiter, wenn das Ereignis eintritt.

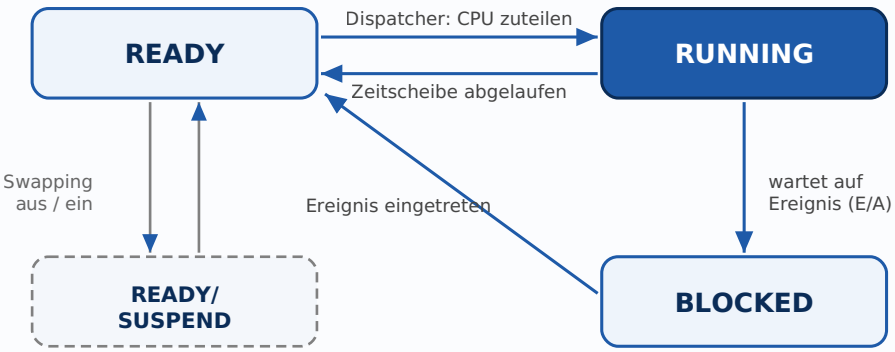
7-Zustands-Modell: Reicht der Hauptspeicher nicht, lagert das BS Prozesse auf die Platte aus (**Swapping**). Es kommen **READY/SUSPEND** und **BLOCKED/SUSPEND** dazu = „ausgelagert“.

Die wichtigsten Übergänge (und ihre Auslöser)

- READY → RUNNING: Dispatcher teilt CPU zu.
- RUNNING → READY: Zeitscheibe abgelaufen (Verdrängung).
- RUNNING → BLOCKED: Prozess wartet auf Ereignis (z. B. E/A).
- BLOCKED → READY: erwartetes Ereignis ist eingetreten (Interrupt „fertig“).
- → SUSPEND: ausgelagert auf Platte (Speicher knapp); zurück durch Einlagern.

Klausur-Klassiker (Blatt 2, Frage 2.2): Eine Ereignisliste ist gegeben, du sollst den Zustand jedes Prozesses zu bestimmten Zeitpunkten nennen. Vorgehen: Tabelle Prozess × Zeit führen. E/A-Zugriff ⇒ **BLOCKED** (Ereignis dazuschreiben!), Interrupt „fertig“ ⇒ READY, ausgelagert ⇒ …/SUSPEND. War ein Prozess blockiert *und* ausgelagert und seine E/A wird fertig, ist er **READY/SUSPEND** — erst nach Einlagern READY.

Schaubild — Prozess-Zustandsdiagramm



Kernübergänge des 5-Zustands-Modells; gestrichelt der zusätzliche SUSPEND-Zustand (ausgelagert) im 7-Zustands-Modell. (analog für BLOCKED/SUSPEND)

Einfaches Beispiel — Zustände lesen

P1 startet einen Lesezugriff auf die Festplatte → P1 wird **BLOCKED** (wartet auf „Lesen fertig“). Währenddessen bekommt P2 die CPU → P2 ist **RUNNING**, P3 wartet nur → **READY**. Wenn die Platte meldet „P1 fertig“ (Interrupt), wird P1 wieder **READY** und reiht sich zum Rechnen ein.

Threads — „leichtgewichtige Prozesse“ (Folie 02)

Ein **Thread** ist ein einzelner Ausführungsfaden *innerhalb* eines Prozesses. Ein Prozess kann in mehrere Threads aufgeteilt werden, die **parallel** laufen. Alle Threads eines Prozesses teilen sich dessen Betriebsmittel: Adressraum, geöffnete Dateien, Geräte.

Analogie: Der Prozess ist die Küche (mit allen Geräten); Threads sind mehrere Köche, die gleichzeitig in *derselben* Küche arbeiten.

- Threads erzeugen und zwischen ihnen wechseln ist **schneller** und braucht weniger Aufwand als bei echten Prozessen (kein Adressraumwechsel).
- Der Prozess bleibt **antwortfähig**, auch wenn ein einzelner Thread blockiert.
- Mehrere CPUs → echte Parallelität.
- Kehrseite: das BS schützt Threads **nicht** voreinander (gemeinsamer Speicher → selbst synchronisieren).

2 Scheduling

Welcher Prozess bekommt wann die CPU? · GANTT-Diagramme · Wartezeit & Durchlaufzeit

Worum geht es? Wenn mehrere Prozesse rechnen wollen, aber nur eine CPU da ist, muss das BS eine **Reihenfolge** festlegen. Diese Entscheidung trifft der **Scheduler**. In der Klausur zeichnest du ein GANTT-Diagramm (Zeitbalken) und rechnest die mittlere Wartezeit aus — für dieselbe Prozesstabelle mit verschiedenen Verfahren. Reine Schema-Arbeit.

Analogie: Die CPU ist **eine einzige Kasse** im Supermarkt, die Prozesse sind Kunden. Jedes Verfahren ist eine andere Regel, in welcher Reihenfolge die Kunden drankommen: der Reihe nach, der mit dem kleinsten Einkauf zuerst, oder jeder „eine Minute, dann der Nächste“.

Die Verfahren auf einen Blick

Verfahren	Regel	verdrän-gend?	fair?
FCFS (First Come First Served)	Wer zuerst kommt, mahlt zuerst — Reihenfolge der Ankunft.	nein	ja
SJF / SPN (Shortest Job/Process Next)	Kürzeste Bedienzeit zuerst.	nein	nein
SRTF / SRPT (Shortest Remai-ning Time)	Kürzeste *Restzeit* zuerst.	ja	nein
RR (Round Robin)	Jeder bekommt ein festes Zeitquantum q, dann reihum der Nächste.	ja	ja
Priorität	Höchste Priorität zuerst (kleine Zahl = hohe Priori-tät).	beides	nein*

* Ohne **Aging** (Priorität steigt mit Wartezeit) können niederpriore Prozesse **verhungern** = ewig warten. Mit Aging wird es fair. **Verdrängend (präemptiv)** heißt: ein laufender Prozess kann unterbrochen werden.

Schema in 4 Schritten

1. Zeitachse zeichnen, Ankunftszeiten markieren.
2. An jedem Entscheidungspunkt (Prozessende; bei präemptiv auch jede Ankunft / jedes Quantum-Ende) die Regel anwenden. **Nur schon angekommene Prozesse** betrachten!
3. GANTT-Balken füllen, bis alle fertig sind.
4. Pro Prozess Endzeit ablesen → Wartezeit-Formel → Mittelwert bilden.

Die zwei Formeln

```
Wartezeit = Endzeit − Ankunft − Bedienzeit
```

```
Durchlaufzeit (Turnaround) = Endzeit − An-kunft
```

Mittelwert = Summe ÷ Anzahl der Prozesse. „Bedi-enzeit“ = „Burst“ = wie lange der Prozess rechnen will.

RR-Regel: Ein verdrängter Prozess kommt ans **Ende** der Warteschlange. Kommt im selben Mo-ment ein neuer Prozess an, wird er meist **vor** dem gerade verdrängten eingereiht (Konvention der Vorlesung beachten).

Durchgerechnetes Beispiel

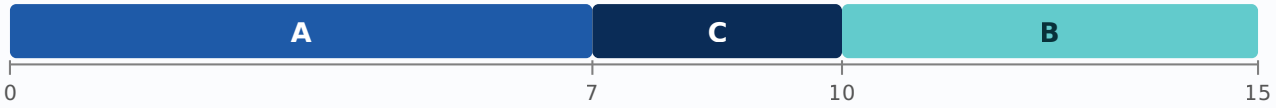
Prozesse: A (Ankunft 0, Bedienzeit 7) · B (2, 5) · C (5, 3)

a) FCFS — nach Ankunft (Reihenfolge A, B, C)

FCFS	A	B	C	⌀
Wartezeit = Ende−Ankunft−Bedienzeit	7−0−7=**0**	12−2−5=**5**	15−5−3=**7**	**4,0**

b) SJF (nicht-verdrängend) — kürzeste Bedienzeit zuerst

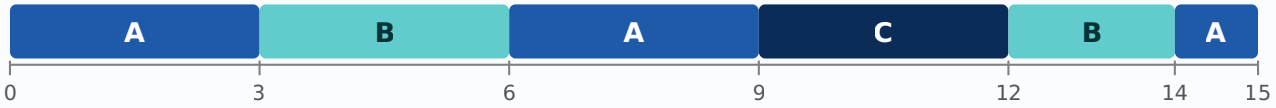
Bei t=0 nur A da → A läuft durch (bis 7). Bei t=7 warten B(5) und C(3) → C ist kürzer → erst C, dann B.



SJF	A	C	B	⌀
Wartezeit	7−0−7=**0**	10−5−3=**2**	15−2−5=**8**	**3,33**

c) Round Robin, q = 3

A 0–3 (Rest 4; B kam bei 2 → [B,A]) · B 3–6 (Rest 2; C kam bei 5 → [A,C,B]) · A 6–9 (Rest 1) · C 9–12 ✓ · B 12–14 ✓ · A 14–15 ✓



RR (q=3)	A	B	C	⌀
Wartezeit	15−0−7=**8**	14−2−5=**7**	12−5−3=**4**	**6,33**

Plausibilitäts-Check: Die Summe aller GANTT-Blöcke = Summe der Bedienzeiten (hier 7+5+3 = 15). Solange immer ein Prozess bereit ist, gibt es keine Lücke. SJF liefert unter den nicht-verdrängenden Verfahren die **kleinste** mittlere Wartezeit — ist dein SJF-Wert größer als FCFS, ist meist die Startphase falsch.

3 Prozess-Synchronisation

Blatt 2 + 3

Semaphore · kritischer Abschnitt · Erzeuger-Verbraucher · Deadlock · Interprozess-Kommunikation

Worum geht es? Wenn mehrere Prozesse **gleichzeitig** auf dasselbe zugreifen (z. B. denselben Speicherbereich), kann Murks entstehen. Synchronisation sorgt für Ordnung: „nur einer gleichzeitig“ oder „erst du, dann ich“. Das Werkzeug dafür heißt **Semaphor**.

Analogie: Ein Semaphor ist wie ein **Korb mit Schlüsseln** vor einer Toilette. **wait/P** = einen Schlüssel nehmen (ist keiner da, musst du warten). **signal/V** = Schlüssel zurücklegen (jetzt darf der Nächste). Ein Korb mit genau 1 Schlüssel = nur einer gleichzeitig (Mutex).

Semaphor: ein Zähler mit zwei Operationen

Operation	Wirkung
wait / P	Zähler um 1 verringern. Ist der Zähler dann unter 0 (bzw. war er 0), **blockiert** der Prozess und wartet.
signal / V	Zähler um 1 erhöhen. Falls jemand wartet, wird einer geweckt.

Anfangswert entscheidet die Rolle: Init **1** = Mutex (genau einer rein) · Init **N** = bis zu N gleichzeitig · Init **0** = „Warte auf Signal“ (Reihenfolge erzwingen).

Drei Begriffe, die oft verwechselt werden

Begriff	Bedeutung
Blockieren	Normales, vorübergehendes Warten (an einem Semaphor / auf E/A). Kein Fehler.
Deadlock (Verklemmung)	Prozesse warten *im Kreis* gegenseitig aufeinander — keiner kommt je weiter.
Starvation (Verhungern)	Ein Prozess kommt nie an die Reihe, obwohl das System weiterläuft.

Kritischer Abschnitt: 3 Korrektheits-Kriterien

- **Gegenseitiger Ausschluss:** nie zwei gleichzeitig drin.
- **Fortschritt:** ist der Abschnitt frei, darf ein Wartender rein.
- **Begrenztes Warten:** niemand wartet unendlich lange.

Reihenfolge erzwingen (Blatt 3, Frage 3.1a)

„opA und opB erst *nachdem* C opC ausgeführt hat.“ C gibt zweimal frei:

```
init(s, 0);   // niemand darf zuerst
A:            B:            C:
 wait(s);      wait(s);      opC;
 opA;          opB;          signal(s);
                             signal(s);
```

So warten A und B, bis C fertig ist und zweimal `signal` gibt. Anfangswert **immer** hinschreiben — gibt Punkte.

Erzeuger-Verbraucher (Standard-Baustein fürs A4-Blatt)

```
init(mutex, 1);  // schützt Puffer
init(leer, MAX); // freie Plätze
init(voll, 0);   // belegte Plätze

Erzeuger:           Verbraucher:
 erzeuge(item);      wait(voll);
 wait(leer);         wait(mutex);
 wait(mutex);        entnimm(item);
 lege ab(item);      signal(mutex);
 signal(mutex);      signal(leer);
 signal(voll);       verbrauche(item);
```

Häufigster Fehler: erst `wait(mutex)`, dann `wait(leer/voll)`. Ist der Puffer voll bzw. leer, hält der Prozess den Mutex **und** blockiert → Deadlock. **Regel:** immer zuerst auf den Zählsemaphor (leer/voll) warten, **dann** den Mutex nehmen.

Deadlock-Klassiker (Blatt 2, 2.12): Zwei Abschnitte sperren zwei Locks in **unterschiedlicher** Reihenfolge (m1→m2 vs. m2→m1). Jeder hält ein Lock und wartet aufs andere → Verklemmung. **Lösung:** Locks immer in **derselben** Reihenfolge nehmen.

Interprozess-Kommunikation (IPC) — Blatt 2

	Gemeinsamer Speicher	Nachrichten (Message Passing)
Wie	Prozesse teilen einen Speicherbereich.	Prozesse senden sich Nachrichten (send/receive).
Tempo	schnell	langsamer (Systemaufrufe)
Synchron.	Prozesse müssen selbst sichern	der BS-Kern regelt es
Reichweite	gleicher Rechner	auch über Netz / verteilt

Indirekte Kommunikation (Frage 2.7) = über eine **Mailbox** (Briefkasten), nicht direkt an den anderen Prozess. Antwort c) ist richtig.

Busy Waiting (aktives Warten, Frage 2.10): Der Prozess prüft in einer Schleife dauernd „darf ich schon?“ und **verbrennt dabei CPU-Zeit**. Alternative: sich **blockieren** lassen und die CPU freigeben (kostet zwei Kontextwechsel). Ganz vermeiden lässt sich aktives Warten nicht — irgendwo ganz unten (z. B. beim Sperren sehr kurzer Abschnitte / Spinlocks) ist es sinnvoll.

send/receive (Frage 2.9): „send() nicht-blockierend“ = der Sender macht sofort weiter, egal ob die Nachricht schon abgeholt wurde. „receive() blockierend“ = der Empfänger wartet, bis wirklich eine Nachricht da ist.

Monitor (Folie 02): bequemere Alternative zum „nackten“ Semaphor — ein Sprachkonstrukt, bei dem immer nur **ein** Prozess im Monitor aktiv ist (Mutex automatisch). Auf Bedingungen wird mit `wait`/`signal` an **Bedingungsvariablen** gewartet.

4 Speicherverwaltung

Fit-Strategien · Segmentierung · Paging (Adressumrechnung) · Seitenersetzung

Worum geht es? Der Hauptspeicher (RAM) ist begrenzt und muss unter den Prozessen aufgeteilt werden. Drei Aufgabentypen kommen dran: (1) freie Lücken füllen (First/Best/Worst-Fit), (2) logische in physische Adressen umrechnen (Segmente/Paging), (3) Seitenersetzung Schritt für Schritt durchspielen. Alles reine Rechentechnik mit festem Schema.

Fit-Strategien (Blatt 3, Frage 3.2)

Mehrere freie Lücken, mehrere Prozesse — welche Lücke nimmt welcher Prozess?

Strategie	Regel
First-Fit	Nimm die *erste* Lücke, die groß genug ist. Schnell.
Best-Fit	Nimm die *kleinste* Lücke, die noch passt. Wenig Verschnitt, aber langsam.
Worst-Fit	Nimm die *größte* Lücke. Idee: der Rest bleibt brauchbar groß.

Lücken 100·500·200·300·600 KB; Prozesse 212, 417, 112, 426 KB (in dieser Reihenfolge)

	212	417	112	426
First-Fit	500→Rest288	600→Rest183	288→Rest176	wartet (kein Platz)
Best-Fit	300→Rest88	500→Rest83	200→Rest88	600→Rest174 ✓
Worst-Fit	600→Rest388	500→Rest83	388→Rest276	wartet (kein Platz)

Nur **Best-Fit** bringt alle vier Prozesse unter ⇒ es nutzt den Speicher hier am effizientesten. Wichtig: entstandene **Restlücken** weiterverwenden!

Segmentierung (Frage 3.4)

Logische Adresse = (Segmentnummer s, Offset d). Eine Segmenttabelle liefert je Segment **Basis** und **Länge**.

```
phys. Adresse = Basis[s] + d (nur wenn d < Länge[s])
```

Ist **d ≥ Länge[s]** → unzulässiger Zugriff (Trap). Das ausdrücklich hinschreiben — gibt Punkte.

Paging: Adresse zerlegen (Fragen 3.3 / 3.5)

Der Speicher ist in gleich große **Seiten** (logisch) bzw. **Kacheln/Rahmen** (physisch) geteilt.

```
Seitennr. = ⌊Adresse ÷ Seitengröße⌋ Offset = Adresse mod Seitengröße
```

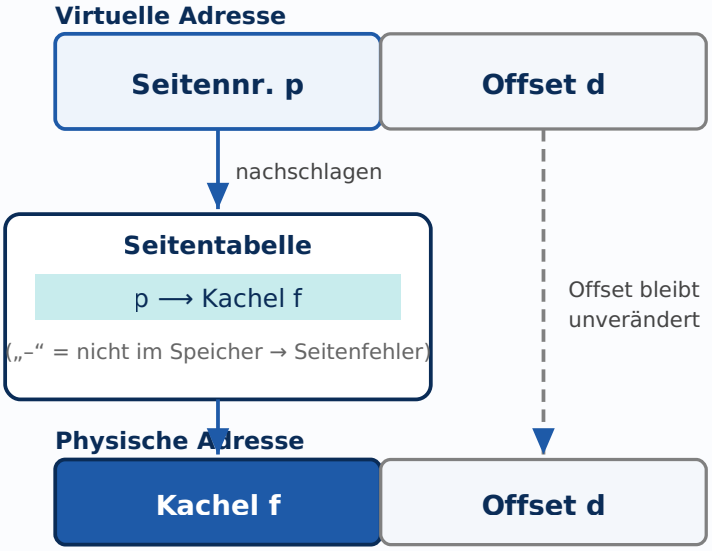
Bei Seitengröße = Zweierpotenz geht es per Bit-Aufteilung: 256 B = 2^8 → die unteren 8 Bit sind der Offset (= 2 Hexziffern), der Rest oben ist die Seitennummer.

Beispiele Adressumrechnung

Seitengröße 1 KB = 1024: **2375** = 2·1024 + 327 → Seite 2, Offset 327. **256** = 0·1024 + 256 → Seite 0, Offset 256.

Hex, 12-Bit-Adresse, Seitengröße 256 B: oberste Hexziffer = Seitennr., untere zwei = Offset. Die Seitentabelle ersetzt nur die oberste Ziffer durch den Rahmen: **9EF → 0EF**. Steht „–“ (nicht im Speicher) → Seitenfehler: Seite in den ersten freien Rahmen der Freiliste laden.

Schaubild — Adressumrechnung beim Paging



Die obere(n) Bit(s) (Seitennummer) werden über die Seitentabelle in eine Kachel übersetzt; der Offset bleibt gleich. phys. Adresse = Kachel · Seitengröße + Offset.

Einfaches Paging vs. virtueller Speicher (Frage 3.6): einfaches Paging = *alle* Seiten eines Prozesses müssen im RAM sein. Virtueller Speicher = es genügt eine *Teilmenge*; fehlende Seiten werden bei Bedarf (Seitenfehler) von der Platte nachgeladen. So laufen Programme, die größer als das RAM sind.

Seitenersetzung (Blatt 3, Frage 3.7)

Ist das RAM voll und eine neue Seite muss rein, welche **Opferseite** fliegt raus?

Strategie	wirft raus …
FIFO	die am längsten geladene Seite (egal wie oft genutzt).
LRU	die am längsten nicht benutzte Seite.
Clock	günstige LRU-Näherung: erste Seite mit Use-Bit 0 (Use-Bit 1 → auf 0 setzen, „zweite Chance“).
Optimal	die am weitesten in der Zukunft gebrauchte Seite. Nur theoretisch (kennt die Zukunft) — Vergleichsmaßstab.

Merkreihenfolge (Fehlerzahl): Optimal ≤ LRU ≈ Clock ≤ FIFO.

Schema

1. Tabelle: Spalten = Referenzen, Zeilen = Kacheln.
2. Seite schon drin? → **Hit** (bei LRU „zuletzt benutzt“ aktualisieren; bei Clock Use-Bit auf 1).
3. Nicht drin? → **Seitenfehler** markieren, Opferseite nach Strategie wählen, ersetzen.
4. Bei Clock zusätzlich die Zeigerposition mitführen.
5. Fehler **erst zählen**, wenn alle Kacheln erstmals belegt sind (Konvention der Musterlösung).

Referenzstring 7,0,1,2,0,3,0,4,2,3,0,3,2 · 3 Kacheln (gezählt nach Erstbelegung)

FIFO: 7 Fehler · LRU: 6 · Clock: 6 · Optimal: 4. Spaltenweise lesen: Hit → Spalte gleich lassen; Fehler → Opferseite ersetzen.

Ergänzung aus der Vorlesung: Buddy-System & TLB

Buddy-System (Folie 03)

Schnelle Speichervergabe mit Blockgrößen in **Zweierpotenzen**. Vorgehen:

1. Anforderung auf die nächste Zweierpotenz aufrunden.
2. Gibt es keinen passenden Block, einen größeren wiederholt **halbieren**; die zwei Hälften heißen **Buddies** (sie unterscheiden sich nur im k-ten Adressbit).
3. Beim Freigeben: ist der Buddy auch frei, beide wieder zu einem größeren Block **verschmelzen**.

\+ sehr schnell, einfaches Verschmelzen · − interne Fragmentierung (Aufrunden verschenkt Platz).

TLB — Translation Lookaside Buffer (Folie 03)

Jede Adressumrechnung (virtuell → physisch) müsste eigentlich erst in der Seitentabelle im RAM nachschlagen — ein zusätzlicher Speicherzugriff pro Befehl. Das ist langsam.

Der **TLB** ist ein kleiner, schneller **assoziativer Cache** direkt in der MMU, der die zuletzt benutzten Seiten-→-Kachel-Zuordnungen speichert.

- **TLB-Hit:** Zuordnung im TLB → sofort, kein RAM-Zugriff für die Tabelle.
- **TLB-Miss:** erst in der Seitentabelle nachschlagen, dann in den TLB übernehmen.

Analogie: Spickzettel mit den häufigsten Telefonnummern, statt jedes Mal das ganze Telefonbuch aufzuschlagen.

5 Dateiverwaltung

Blatt 3 + 4

Allokationsverfahren · FAT-Ketten · I-Nodes (max. Dateigröße) · Freispeicher (Bitmap, Freiliste)

Worum geht es? Eine Festplatte ist in viele gleich große **Blöcke** geteilt. Eine Datei besteht aus mehreren Blöcken. Das Dateisystem muss sich merken, *welche* Blöcke zu welcher Datei gehören und welche frei sind. Zwei Rechenklassiker: FAT-Ketten verfolgen und die maximale Dateigröße eines I-Nodes berechnen.

Analogie: Die Platte ist ein Regal mit nummerierten Fächern (Blöcken). Eine Datei ist ein Buch, das auf mehrere Fächer verteilt liegt. Du brauchst ein Inhaltsverzeichnis, das sagt, in welchen Fächern dein Buch steht und in welcher Reihenfolge.

Allokationsverfahren — wie liegen die Blöcke?

Verfahren	Idee	+ / −
Contiguous (zusammenhängend)	Alle Blöcke direkt hintereinander.	+ schneller Direktzugriff · − externe Fragmentierung, Umkopieren beim Wachsen
Linked (verkettet)	Jeder Block zeigt auf den nächsten.	+ wächst/schrumpft frei · − kein schneller wahlfreier Zugriff
Indexed (Indexblock/I-Node)	Ein Indexblock listet alle Blöcke der Datei.	+ wahlfreier Zugriff einfach, flexibel · − Overhead bei winzigen Dateien

Datei schwankt 4 KB–4 MB (Frage 4.2): Indexed ist am besten — die Größe variiert stark und wahlfreier Zugriff ist wahrscheinlich. **Große Blockgröße (Frage 4.1):** gut für große Dateien / sequentiellen Zugriff (weniger Transfers und Zeiger), schlecht bei vielen kleinen Dateien (interne Fragmentierung — der halbleere letzte Block verschwendet Platz).

FAT-Prinzip

Das Verzeichnis nennt den **Startblock**. Eine zentrale Tabelle (FAT) enthält je Block die Nummer des **Nachfolgers** (oder „nil“ = Ende, „free“, „bad“).

```
Blockindex in der Datei = ⌊Byte-Nr ÷ Blockgröße⌋ (ab 0!)
```

Dann ab dem Startblock genau so viele FAT-Schritte weitergehen. Offset = Byte-Nr mod Blockgröße.

Typische Fehler: Bei Byte = Vielfaches der Blockgröße landet man im **nächsten** Block (Offset 0), nicht im vorherigen. Und „bad“/„free“-Einträge sind **keine** Kettenglieder.

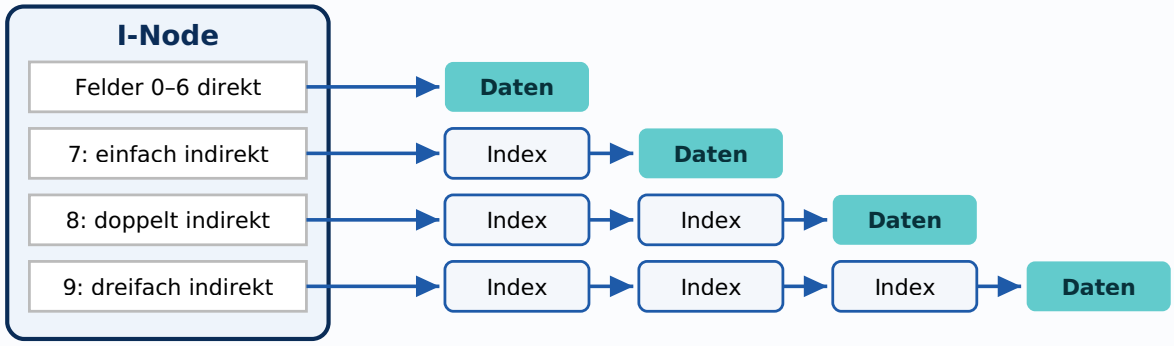
I-Node: maximale Dateigröße (Frage 4.3)

Ein I-Node hat ein paar **direkte** Blockadressen plus indirekte (einfach/doppelt/dreifach). Eine indirekte Adresse zeigt auf einen Block *voller weiterer Adressen*.

```
Adressen pro Block = Blockgröße ÷ Adresslänge
```

Beispiel „linode“: Block 4 KiB, Adresse 2 Byte ⇒ 4096÷2 = **2048** Adressen/Block. 7 direkt + 2048 (einfach) + 2048^2 (doppelt) + 2048^3 (dreifach) Blöcke. Mal Blockgröße = max. Dateigröße (≈ 32 TiB). Schema: erst Adressen/Block, dann Blöcke pro Stufe summieren, dann × Blockgröße.

Schaubild — I-Node mit direkten und indirekten Zeigern



Jede Indirektionsstufe schiebt einen weiteren Block voller Adressen dazwischen — daher wächst die maximale Dateigröße enorm (einfach → doppelt → dreifach indirekt).

Freispeicherverwaltung (Blatt 4)

Bitmap (Frage 4.4)

1 Bit pro Block: 1 = belegt, 0 = frei. Es wird immer der freie Block mit der **niedrigsten** Nummer vergeben (von links). Gelöschte Lücken werden also zuerst wieder gefüllt.

Start: 1111 1110 … nach Datei A (6 Blöcke)

B schreiben (5 Bl.): `1111 1111 1111 0000`
A löschen: `1000 0001 1111 0000`
C schreiben (8 Bl.): `1111 1111 1111 1100`
B löschen: `1111 1110 0000 1100`

Verkettete Freiliste — Zeiger verloren (Frage 4.5)

Ja, rekonstruierbar: Man durchsucht alle Dateien/Verzeichnisse ab der Wurzel, markiert alle benutzten Blöcke; alles Unmarkierte ist frei. Aufwand wächst linear mit der Dateianzahl.

rename vs. Kopieren+Löschen (Frage 3.8): `rename` ändert nur den Verzeichniseintrag (schnell, kein neuer Speicher). Kopieren belegt erst neue Blöcke (kann bei voller Platte scheitern) und ändert Attribute wie die Erstellungszeit.

open()/close() (Frage 3.10): `open()` prüft, ob die Datei existiert und der Prozess Zugriff hat, legt einen Eintrag in der Open-File-Tabelle an und gibt einen Verweis (Handle) zurück. `close()` entfernt diesen Verweis und gibt Ressourcen frei.

Ergänzung aus der Vorlesung: Journaling & Zugriffsrechte

Journaling (Folie 04)

Problem: Stürzt der Rechner *während* einer Schreiboperation ab, kann das Dateisystem inkonsistent werden (halb geschriebene Daten). Lösung: ein **Journal**.

1. Geplante Schreiboperationen zuerst in einen reservierten Bereich (das Journal) **protokollieren**.
2. Zu festen Zeitpunkten das Journal auf den Massenspeicher **durchschreiben (Commit)**.
3. Nach einem Absturz wird das Journal einfach **erneut durchgeschrieben** → konsistenter Zustand.

Analogie: erst auf den Notizzettel schreiben, dann sauber ins Hauptbuch übertragen — bricht etwas ab, hat man den Notizzettel.

Zugriffsrechte & ACL (Folie 04)

Jede Datei trägt in ihren Metadaten (im I-Node) Attribute: Größe, Zeitstempel, Eigentümer und **Zugriffsrechte**.

- Unix-Klassiker: `rwx` (read/write/execute) je für Eigentümer, Gruppe, andere.
- **ACL** (Access Control List): feinere Liste, die einzelnen Nutzern/Gruppen gezielt Rechte gibt.

open() prüft genau diese Rechte, bevor es eine Datei freigibt.

6 Ein-/Ausgabe (E/A)

Pufferung (Doppelpuffer) · Gerätetreiber · interrupt-gesteuerte E/A

Worum geht es? Geräte (Festplatte, Tastatur, Netzwerk) sind viel langsamer als die CPU. Dieser Block ist reine **Verständnisfrage** — keine Rechnung. Drei Antworten reichen: Doppelpuffer, Treiber ohne Neukompilieren, Ablauf einer interrupt-gesteuerten E/A.

Einzel- vs. Doppelpuffer (Frage 4.6)

Ein **Puffer** ist ein Zwischenspeicher. Beim **Einzelpuffer** muss erst gelesen, dann verarbeitet werden — abwechselnd, mit Wartezeiten.

Beim **Doppelpuffer** gibt es zwei: während Puffer 1 *verarbeitet* wird, füllt das Gerät schon Puffer 2. Lesen und Verarbeiten **überlappen** → schneller.

Analogie: Geschirr spülen mit zwei Becken — im einen einweichen, im anderen abtrocknen, ohne Pause.

Randbedingung: Gilt dauerhaft Verarbeitungszeit < Lesezeit (oder umgekehrt), läuft der Vorteil des zweiten Puffers irgendwann leer. Der Doppelpuffer hilft langfristig nur, wenn die Differenz **schwankt** — dann gleicht er die Spitzen aus.

Treiber ohne Neukompilieren (Frage 4.7)

Ein **Treiber** ist das Programm, das ein bestimmtes Gerät ansteuert. Er wird vom Hersteller geliefert und als **dynamische Bibliothek** nachgeladen. Bei der Installation trägt das BS in eine nach Gerätenummern sortierte Tabelle Zeiger auf `open/close/read/write` ein.

Ein System Call mit der Gerätenummer findet darüber die richtige Funktion. Das BS selbst bleibt **unverändert** — kein Neu-Übersetzen nötig.

Abgrenzung: **Polling** = CPU fragt aktiv in einer Schleife ab (Busy Waiting, verschwendet Zyklen). **Interrupt** = das Gerät meldet sich selbst, wenn es fertig ist. **DMA** = ein Controller schiebt ganze Blöcke direkt in den Speicher, die CPU wird nur am Ende einmal unterbrochen.

Interrupt-gesteuerte E/A: der Ablauf (Frage 4.8)

1. Das Programm löst die E/A per **System Call** aus; der aufrufende Prozess wird **blockiert**; der Treiber startet das Gerät.
2. Das Gerät arbeitet selbstständig; die CPU kann inzwischen andere Prozesse rechnen lassen.
3. Gerät fertig → es löst einen **Interrupt** aus und unterbricht die CPU.
4. Die CPU sichert ihren aktuellen Zustand und springt über den Interrupt-Vektor in die **Interrupt-Service-Routine (ISR)**.
5. Die ISR verarbeitet das Ereignis und setzt den wartenden Prozess zurück auf **READY**.
6. **Return from Interrupt** → die CPU macht dort weiter, wo sie unterbrochen wurde.

Kernidee: Die CPU wartet **nicht** aktiv auf das langsame Gerät, sondern wird erst „angeklingelt“, wenn das Ergebnis da ist. Das ist effizienter als Polling.

7 Rechnernetze: Grundlagen

Schichtenmodelle (OSI / TCP-IP) · Header · Leitungs- vs. Paketvermittlung · „Bernie“

Worum geht es? Hier beginnt Teil 2. Netze sind in **Schichten** aufgebaut — jede Schicht hat eine klare Aufgabe und nutzt die Dienste der Schicht darunter. Das macht das Ganze beherrschbar. Die Grundlagenaufgaben fragen Schichtenverständnis, Header-Overhead und einen Vergleich der Vermittlungsarten ab.

Analogie: Einen Brief verschicken. Du schreibst den Inhalt (Anwendung), steckst ihn in einen Umschlag mit Adresse (Vermittlung), die Post transportiert ihn physisch (Bitübertragung). Jede Ebene fügt ihre eigene „Hülle“ hinzu — genau das macht ein **Header**.

ISO/OSI (7 Schichten) und TCP/IP (4 Schichten)

OSI	Aufgabe	TCP/IP & Beispiele
7 Anwendung 6 Darstellung 5 Sitzung	Dienste für Programme; Codierung, Dialog.	**Anwendungsschicht** — HTTP, DNS, FTP, SMTP
4 Transport	Ende-zu-Ende-Verbindung zwischen zwei Programmen.	**Transportschicht** — TCP, UDP
3 Vermittlung	Wegfindung quer durchs Netz (Routing).	**Internetschicht** — IP, ICMP
2 Sicherung 1 Bitübertragung	Übertragung über eine Leitung; Fehlererkennung, Medienzugriff.	**Netzzugangsschicht** — Ethernet, WLAN, CRC

Wozu Header? (Kapselung)

Beim Senden fügt **jede** Schicht ihren eigenen Header (Steuerinfos: Adressen, Nummern, Prüfsummen) vorne an; beim Empfänger entfernt die passende Schicht ihn wieder. So „spricht“ jede Schicht mit ihrer Partnerschicht der Gegenseite.

```
Header-Anteil = n·h / (n·h + M)
```

n Schichten, h Byte Header je Schicht, M Byte Nutzdaten (Frage 4.11).

Schichten-Spielregeln (Frage 4.10)

Eine Schicht darf nur mit (a) der direkt darunter und (b) logisch mit der gleichen Schicht der Gegenseite reden. Schichten überspringen oder „quer durchtelefonieren“ ist ein **Verstoß** (Klausurtyp: Verstöße in einer Geschichte markieren).

Leitungs- vs. Paketvermittlung (Frage 4.12)

Leitungsvermittlung	Paketvermittlung
Fester Kanal wird vorab aufgebaut (Telefon).	Nachricht in Pakete zerlegt; Router speichern & leiten weiter.
+ konstante Rate, − Aufbauzeit, Kanal bei Pausen ungenutzt.	+ flexible Auslastung, kein Aufbau; − variable Verzögerung, Pufferung.

```
T_Leitung = s + x/b + k·d T_Paket = x/b + (k−1)·p/b + k·d
```

x Bit Nachricht, k Teilstrecken, b bit/s, d Laufzeit je Strecke, s Aufbauzeit, p Paketgröße.

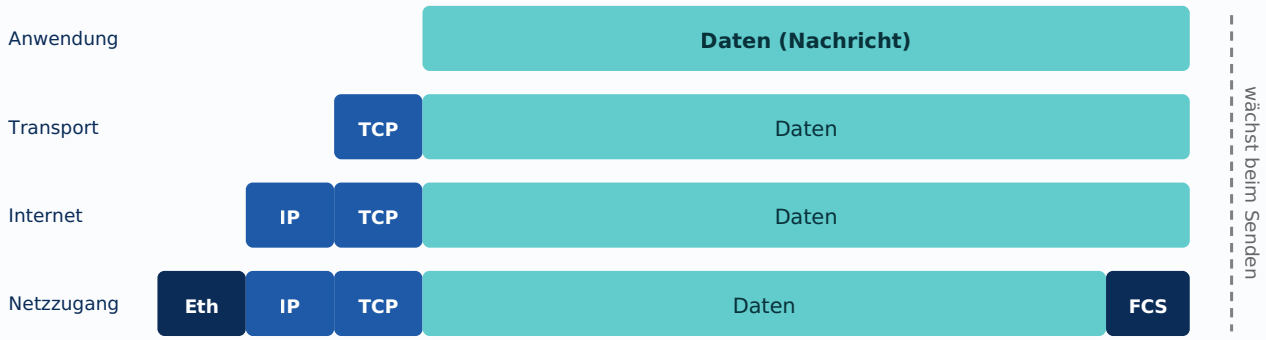
```
Paket schneller ⇔ (k−1)·p/b < s
```

„Bernie der Bernhardiner“ (Frage 4.9) — Datenrate per Hund

3 Bänder × 7 GiB = $3 \cdot 7 \cdot 2^{30} \cdot 8 \approx 1{,}8 \cdot 10^{11}$ bit. Eine 150-Mbit/s-Leitung braucht dafür ≈ 1200 s. Bernie läuft 18 km/h = 5 m/s → in 1200 s schafft er 6000 m. **Unter ~6 km ist der Hund schneller** als die Leitung. Skalierung: Geschwindigkeit ×2 oder Bandkapazität ×2 → Grenze ×2; Leitungsrate ×2 →

Grenze ÷2.
Lehre: Für riesige Datenmengen über kurze Distanz schlägt physischer Transport jede Leitung.

Schaubild — Kapselung: jede Schicht fügt ihren Header hinzu



Beim Senden hüllt jede Schicht die Daten in ihren eigenen Header (Sicherungsschicht zusätzlich einen Trailer/FCS). Beim Empfänger wird Schicht für Schicht wieder ausgepackt.

Baud vs. Bitrate (Probeklausur 5c)

Die **Schrittgeschwindigkeit** (in Baud) sagt, wie viele Signalschritte pro Sekunde gesendet werden. Die **Bitrate** hängt zusätzlich davon ab, wie viele **Zustände** ein Signalschritt unterscheiden kann — je mehr Zustände, desto mehr Bits pro Schritt.

```
Bitrate = Schrittgeschwindigkeit × log₂(Anzahl Signalzustände)
```

Oktonäres Signal, 12 kBd

Oktonär = 8 Zustände → $\log_2(8) = 3$ Bit pro Schritt.
Bitrate = 12 000 × 3 = **36 kbit/s**.

Binär ×1 · ternär ×$\log_2 3 \approx 1{,}58$ · quaternär ×2 · oktonär ×3.

8 Sicherungsschicht

Stop-and-Wait · Go-back-N · CRC · CSMA/CD & Backoff · Manchester · Prüfsumme

Worum geht es? Die Sicherungsschicht macht aus einer unzuverlässigen Leitung eine **zuverlässige Strecke**: Rahmen bilden, Fehler erkennen (CRC, Prüfsumme), Tempo abstimmen (Fenster), und bei geteiltem Medium den Zugriff regeln (CSMA/CD). Vieles ist formelbasiert.

Stop-and-Wait: Kanalausnutzung (Frage 5.6)

Sende einen Rahmen, dann **warte** auf die Bestätigung, erst dann den nächsten. Viel Leerlauf, wenn die Leitung lang ist.

```
U = T_Frame / (T_Frame + 2·T_Prop)
```

T_Frame = L/b (Sendezeit), T_Prop = Signallaufzeit. Für U ≥ 50 %: T_Frame ≥ 2·T_Prop, also L ≥ 2·T_Prop·b.

b = 4 kbit/s, T_Prop = 20 ms

L ≥ 2·0,02 s·4000 bit/s = **160 bit**. Ab 160 Bit Rahmengröße ist die Ausnutzung ≥ 50 %.

Go-back-N / Gleitfenster (Frage 5.1)

- **n-Bit-Seq.-Nummern** → Werte 0…2^n-1; max. Sendefenster = **2^n-1** (bei 3 Bit: höchstens 7).
- Bei Lücke verwirft der Empfänger die Folgerahmen; der Sender **wiederholt alle ab dem fehlenden** („go back n“).
- **Piggybacking:** die Bestätigung (ACK) reist im Datenrahmen der Gegenrichtung mit; eigene ACK-Rahmen nur, wenn keine Daten anstehen. **NAK** (negativ) sofort senden.
- **Weg-Zeit-Diagramm:** schräge Pfeile (Laufzeit!), jeden Rahmen mit Seq-Nr. und ACK-Feld beschriften.

CRC — Fehlererkennung durch Polynomdivision (Frage 5.7)

Idee: An die Nachricht werden Prüfbits angehängt, sodass das Ganze durch ein vereinbartes **Generatorpolynom** teilbar ist. Beim Empfänger: Rest 0 → ok, Rest ≠ 0 → Fehler.

1. Grad des Generators = r → r Nullen anhängen.
2. **Modulo-2-Division** (XOR statt Minus, kein Übertrag) durch den Generator.
3. Der Rest (r Bit) ersetzt die angehängten Nullen → das ist die gesendete Folge.
4. Empfänger teilt erneut: Rest 0 = kein erkennbarer Fehler.

10011101, Generator x^3+1 = 1001

r = 3 → 10011101**000** ÷ 1001 (mod 2) → Rest **100**.
Gesendet: **10011101100**.
Beim Empfänger Rest ≠ 0 → Fehler erkannt.

Nicht erkennbar sind genau die Fehlermuster, die selbst ein Vielfaches des Generators sind. **Rechentrick:** führende 1 → Generator XOR-en; führende 0 → eine Stelle weiterrücken. Kein „größer/kleiner“-Vergleich nötig.

Medienzugriff & Codierung

CSMA/CD + Binary Exponential Backoff (Frage 5.8)

Bei Ethernet hören Stationen vor dem Senden ab (Carrier Sense). Bei einer Kollision warten beide eine **zufällige** Zeit, ehe sie es erneut versuchen.

```
nach n-ter Kollision: k gleichverteilt aus {0,…,2ⁿ−1} → 2ⁿ Werte
```

Nach der 5. Kollision: $k \in \{0,\ldots,31\}$ → 32 Werte → $P(k=4) = \mathbf{1/32}$. Wartezeit = k Slotzeiten.

Häufigster Fehler: $\{0\ldots2^n-1\}$ enthält $\mathbf{2^n}$ Werte, nicht 2^n-1.

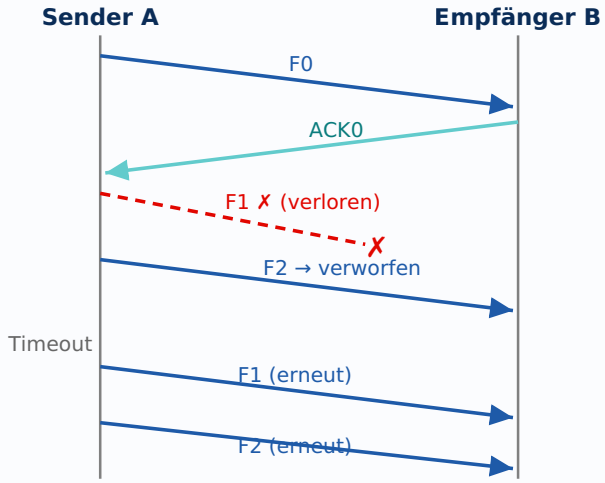
Hidden Terminal (Frage 5.10)

Funkstationen hören sich nur teilweise. Sendet A an B, können andere Stationen senden, die B **nicht** stören (außerhalb von B's Reichweite). Antwort bestimmst du, indem du je Station prüfst: Liegt sie in Reichweite des Empfängers? Wenn nein, darf sie parallel senden.

Manchester-Codierung (Frage 5.11)

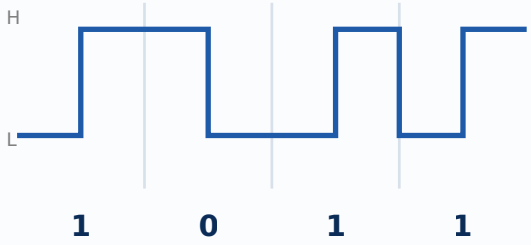
Jedes Bit hat eine **Flanke** in der Bitmitte: in der Vorlesungskonvention z. B. **Low→High = 1**, **High→Low = 0**. Bitfolge ablesen = Flankenrichtung je Bit notieren.

Schaubild — Go-back-N (Weg-Zeit)



Geht ein Rahmen verloren, verwirft B alle Folgerahmen; A sendet ab dem fehlenden alles erneut.

Schaubild — Manchester-Codierung



Flanke in der Bitmitte: Low→High = 1, High→Low = 0. Hier ergibt sich 1011.

Internet-Prüfsumme (Frage 5.12): Bytes paarweise zu 16-Bit-Wörtern zusammenfassen, alle Wörter mit **Einerkomplement-Addition** summieren (Überlauf vorne wieder addieren), dann das Ergebnis **bitweise invertieren**. Für „Networking“ (10 Byte = 5 Wörter) ebenso vorgehen.

Ergänzung aus der Vorlesung: Fehlererkennung/-korrektur & Framing (Folie 10)

Erkennen vs. Korrigieren

- **Parität** (Querparität/VRC): ein Prüfbit pro Zeichen macht die Anzahl der Einsen gerade/ungerade — erkennt **einzelne** Bitfehler, kann sie aber nicht orten.
- **CRC**: erkennt ganze Fehlerbündel (siehe oben), korrigiert nicht.
- **Fehlerkorrektur** (z. B. Hamming): genug Prüfbits, um den Fehler nicht nur zu erkennen, sondern die fehlerhafte Stelle zu **finden und umzudrehen**. Faustregel über die **Hamming-Distanz** d: $d-1$ Fehler erkennbar, $\lfloor (d-1)/2 \rfloor$ korrigierbar.

Rahmenbildung (Framing)

Aus dem Bitstrom müssen **Rahmengrenzen** erkennbar sein. Verfahren u. a.:

- feste Länge / Längenfeld im Header;
- **Byte-/Bit-Stuffing**: ein Sonderzeichen markiert Anfang/Ende; kommt das Muster in den Daten vor, wird ein Füllzeichen/-bit eingefügt, das der Empfänger wieder entfernt.

Beispiel-Rahmenformat: IEEE 802.3 (Ethernet) mit Präambel, Adressen, Längen-/Typfeld, Nutzdaten und FCS (CRC).

9 Vermittlungsschicht & IP

IP-Subnetting (VLSM) · CIDR-Routing (Longest-Prefix) · Dijkstra · Link-State-Routing

Worum geht es? Diese Schicht findet den **Weg** eines Pakets quer durchs Internet. Drei Aufgabentypen: einen Adressblock in Subnetze zerschneiden (Subnetting), anhand einer Routingtabelle entscheiden, wohin ein Paket geht (CIDR), und kürzeste Wege berechnen (Dijkstra / Link-State). Subnetting ist die ergiebigste Rechenaufgabe.

Analogie: Eine IP-Adresse ist wie eine Postanschrift mit Stadt+Hausnummer. Das **Präfix** (/24 usw.) sagt, wie viele vordere Bits die „Stadt" (das Netz) bilden; der Rest ist die „Hausnummer" (der Host). Subnetting = eine große Stadt in Bezirke aufteilen.

IP-Subnetting / VLSM (Frage 5.4)

Ein /24-Block hat 256 Adressen. Pro Subnetz gehen **2 Adressen** verloren (Netz- und Broadcast-Adresse).

```
nutzbare Hosts = 2^(32−Präfix) − 2
```

Präfix	/25	/26	/27	/28	/29	/30
Adressen	128	64	32	16	8	4
nutzbar	126	62	30	14	6	2

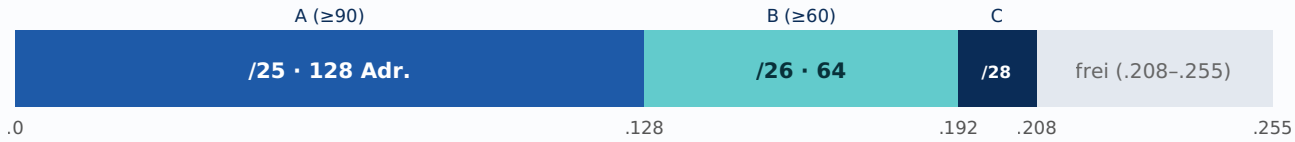
Schema (VLSM)

1. Bedarf je Subnetz **+2** → auf nächste Zweierpotenz aufrunden → Präfix.
2. **Größtes Subnetz zuerst** platzieren.
3. Der Reihe nach vergeben; jede Subnetzadresse muss ein **Vielfaches** ihrer Blockgröße sein (Ausrichtung!).
4. Kontrolle: lückenlos? überlappungsfrei? Ausrichtung korrekt?

223.1.17.0/24 für ≥90, ≥60, ≥12 Hosts

223.1.17.0/25 (126 Hosts ≥ 90) · 223.1.17.128/26 (62 ≥ 60) · 223.1.17.192/28 (14 ≥ 12). Größtes zuerst!

Schaubild — Aufteilung des /24-Blocks (256 Adressen)



Größtes Subnetz zuerst, jede Subnetzadresse ist ein Vielfaches der Blockgröße (Ausrichtung). Die Blockbreite verdoppelt sich mit jeder kleineren Präfixzahl.

CIDR-Routing & Longest-Prefix-Matching (Frage 5.5)

Ein Router vergleicht die Zieladresse mit allen Tabelleneinträgen. Es gewinnt der **spezifischste** (längste) passende Präfix. Passt keiner → **default**.

Vorgehen: Zieladresse und Eintragspräfix in Binär vergleichen — stimmen die ersten (Präfix-)Bits überein, passt der Eintrag. Beispiel 135.46.63.10 fällt in 135.46.60.0/22 (Interface 1), weil /22 nur die ersten 22 Bit prüft.

Kürzeste Wege: Dijkstra & Link-State (Fragen 5.2 / 5.3)

Dijkstra (Frage 5.2)

1. Startknoten Distanz 0, alle anderen ∞.
2. Den unbesuchten Knoten mit der **kleinsten** Distanz wählen, als „fertig“ markieren.
3. Seine Nachbarn aktualisieren: ist „aktuelle Distanz + Kante“ kleiner, übernimm sie (Vorgänger merken).
4. Wiederholen, bis alle fertig sind. Ergebnis: kürzeste Distanz + Pfad zu jedem Knoten.

Link-State-Routing / LSA (Frage 5.3)

- Jeder Router misst die Kosten zu seinen **direkten Nachbarn** und flutet sie als **LSA** (Link State Advertisement) an alle (Flooding; Sequenznummer + Alter verhindern Schleifen).
- **LSA-Inhalt:** Absender, Liste (Nachbar, Kosten), Sequenznummer, Alter/TTL.
- Aus allen LSAs baut jeder Router die **Distanzmatrix** = vollständige Topologie.
- Daraus die **Routingtabelle** (dst / next-hop / distance) — in der Klausur oft „durch Hinsehen“.

Ergänzung aus der Vorlesung: IP-Bausteine & Distanzvektor (Folien 08/09)

Rund um IP

Baustein	Wofür
TTL	Lebenszeit-Zähler im IP-Header; jeder Router −1, bei 0 wird das Paket verworfen (verhindert ewige Kreise).
Fragmentierung	Paket größer als die MTU einer Strecke → in kleinere Stücke zerlegt (IPv6: macht das nur der Sender).
ARP	findet zur IP-Adresse die zugehörige MAC-Adresse im LAN.
DHCP	weist Hosts automatisch IP-Adresse, Maske, Gateway, DNS zu.
NAT	übersetzt viele private Adressen auf eine öffentliche → spart Adressen.
ICMP	Fehler-/Diagnosemeldungen (ping, „Ziel nicht erreichbar“, TTL abgelaufen).
IPv6	128-Bit-Adressen (8 Hex-Gruppen) statt 32 Bit → praktisch unbegrenzt viele Adressen.

Distanzvektor vs. Link-State

Distanzvektor (RIP)	Link-State (OSPF)
Jeder Router kennt nur seine **Nachbarn** und tauscht mit ihnen seine **ganze Distanztabelle** aus (Bellman-Ford).	Jeder Router flutet seine **Linkkosten** an alle und berechnet selbst die ganze Topologie (Dijkstra).
einfach, aber **Count-to-Infinity**: schlechte Nachrichten verbreiten sich langsam.	schnellere Konvergenz, mehr Aufwand/Speicher.

Hierarchie: Das Internet ist in **Autonome Systeme (AS)** unterteilt; innerhalb eines AS läuft z. B. RIP/OSPF, zwischen den AS **BGP**.

10 Transportschicht: TCP

Sendefenster & Window Scaling · Slow Start · RTT-Schätzung (Jacobson) · TCP Reno

Worum geht es? TCP sorgt für eine zuverlässige Verbindung zwischen zwei Programmen: nichts geht verloren, alles kommt in der richtigen Reihenfolge an, und der Sender passt sein Tempo an. Die Aufgaben sind kleine Rechnungen — mit wenigen Formeln sicher lösbar.

Analogie: Das **Sendefenster** ist wie die Anzahl Pakete, die ein Versandlager losschicken darf, *bevor* die erste Empfangsbestätigung zurückkommt. Je länger der Postweg (RTT), desto mehr muss „unterwegs“ sein dürfen, damit der Laster nie stillsteht.

SeqNo/AckNo-Tabellen: Auf-, Daten- und Abbau (Probeklausur 7)

Dieser Aufgabentyp ist mit **zwei Regeln** komplett lösbar:

```
Regel 1: SYN und FIN verbrauchen je 1 Sequenznummer (zählen wie 1 Byte). Regel 2: AckNo = empfangene SeqNo + Nutzdatenlänge (+1 bei SYN/FIN) = nächstes erwartetes Byte
```

a) Verbindungsaufbau (3-Wege-Handshake)

Nr	SYN	ACK	FIN	Len	SeqNo	AckNo
1	1	0	0	0	500	—
2	1	1	0	0	**1000**	**501**
3	0	1	0	0	**501**	**1001**

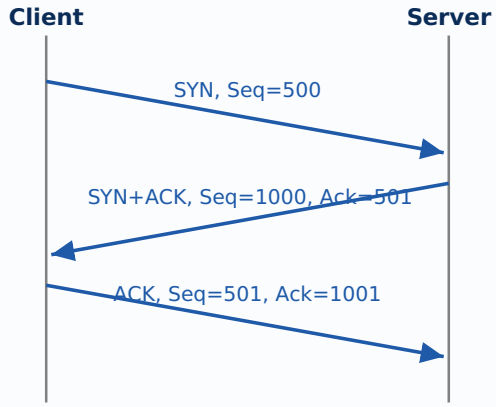
Nr 2 = SYN+ACK des Servers, bestätigt 500+1=501. Nr 3 = ACK des Clients, eigene SeqNo 501, bestätigt 1000+1=1001.

b) Datenphase (Beispiel)

Nr	Len	SeqNo	AckNo
4	50	501	1001
5	0	1001	**551**
6	100	551	1001
7	0	1001	**651**

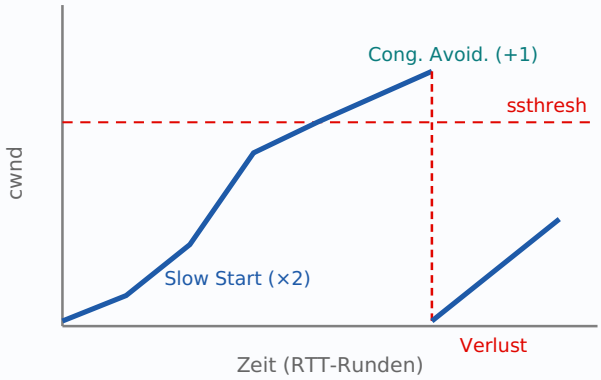
Abbau (c): gleiches Schema mit FIN-Flag; FIN zählt wie 1 Byte, das abschließende ACK bestätigt FIN-SeqNo + 1.

Schaubild — 3-Wege-Handshake



SYN/FIN zählen wie 1 Byte; AckNo = empfangene SeqNo + 1.

Schaubild — Slow Start & Congestion Avoidance



Erst exponentielles Verdoppeln, ab ssthresh nur noch +1 pro Runde; bei Verlust Einbruch.

Sendefenster & Window Scaling (Frage 4.14)

```
Fenster für Dauersenden = RTT × Datenrate
```

Satellit: 800 ms × 24 Mbit/s = 19,2 Mbit = **2,4 MByte**.

Problem: Das TCP-Fensterfeld ist nur 16 Bit → max. $2^{16}-1$ = 65 535 Byte. Viel zu klein.

Lösung: Window-Scale-Option (Faktor 2^n, n ≤ 14). Hier n = 6, denn $2^{16}\cdot 2^6 = 2^{22} \geq 2{,}4\cdot 10^6$.

Jacobson-RTT-Schätzung (Frage 4.15)

```
RTT_neu = α · RTT_alt + (1−α) · M
```

M = gemessener Wert. Beispiel α = 0,9, Start 30 ms, Messungen 26/32/24: → 29,6 → 29,84 → 29,256 ms. Immer den **zuletzt geschätzten** Wert einsetzen.

Slow Start (Frage 4.16)

Das Überlastungsfenster startet bei **1 MSS** und **verdoppelt** sich pro RTT-Runde (1, 2, 4, 8, …) bis zur Schwelle **ssthresh**; danach „Congestion Avoidance“: nur noch +1 pro Runde.

RTT 10 ms, Empfangsfenster 24 KB, MSS 2 KB

Runden: 2 → 4 → 8 → 16 KiB. Runde 5 wäre 32 KiB > 24 KB → volles Fenster nach **4 Runden = 40 ms**.

TCP Reno: Verlust erkennen (Frage 4.17)

- **Slow Start** erkennt man am **Verdoppeln** pro Runde; **Congestion Avoidance** am **linearen +1**.
- **Triple Duplicate ACK** (3 gleiche ACKs): leichter Verlust → ssthresh = Fenster/2, Fenster = ssthresh (Fast Recovery), *kein* kompletter Neustart.
- **Timeout:** schwerer Verlust → ssthresh = Fenster/2, Fenster springt auf **1** (neuer Slow Start).

Diagramm lesen: Sägezahn-Kurve. Steiler (exponentieller) Anstieg = Slow Start. Sanfter (linearer) Anstieg = Congestion Avoidance. Absturz auf 1 = Timeout; halber Einbruch = Triple Duplicate ACK.

Ergänzung aus der Vorlesung: UDP, Ports & zwei Kontrollarten (Folie 07)

TCP vs. UDP

TCP	UDP
verbindungsorientiert; Fehler-, Fluss- und Überlastkontrolle; Reihenfolge garantiert.	verbindungslos; keine Kontrollmechanismen; Reihenfolge nicht garantiert; nur Prüfsumme.
zuverlässig, aber mehr Overhead.	schlank & schnell → Streaming, Echtzeit, DNS.

Ports & Sockets

Die IP-Adresse adressiert den **Rechner**, der **Port** das einzelne Programm darauf. Ein **Socket** = IP + Port. So weiß das Betriebssystem, welcher Anwendung ein ankommendes Segment gehört (Multiplexing/Demultiplexing).

Flusskontrolle ≠ Überlastkontrolle

- **Flusskontrolle:** schützt den **Empfänger** vor Überlauf — er teilt mit seinem Empfangsfenster mit, wie viel er noch aufnehmen kann.
- **Überlast-/Staukontrolle (Congestion):** schützt das **Netz** vor Überlastung — Slow Start, Congestion Avoidance, Reaktion auf Verluste (siehe oben).

Merksatz: Das tatsächliche Sendefenster ist das **Minimum** aus Empfangsfenster (Fluss) und Überlastfenster (Congestion).

11 Anwendungsschicht

Blatt 4 + 1

DNS (rekursiv/iterativ) · Protokolle FTP, Telnet, SNMP · Shell & Batch-Scripting

Worum geht es? Die oberste Schicht — hier leben die Programme, die du direkt benutzt. Prüfungsrelevant: wie aus einem Namen wie `www.thi.de` eine IP-Adresse wird (DNS) und wozu einige bekannte Protokolle dienen. Dazu die praktische Shell-Übung aus Blatt 1.

DNS — vom Namen zur IP-Adresse (Frage 4.18)

Namensbestandteile von www.thi.de

- **www** = Hostname
- **thi** = Second-Level-Domain
- **de** = Top-Level-Domain (TLD)
- **thi.de** = Domänenname der Zone

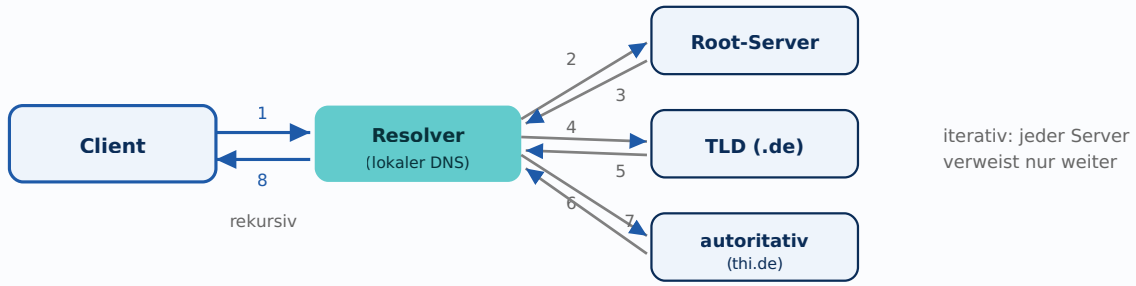
Analogie: DNS ist das **Telefonbuch** des Internets: du kennst den Namen, willst aber die Nummer (IP).

rekursiv vs. iterativ

rekursiv	iterativ
Client fragt seinen Resolver; **dieser** besorgt die finale Antwort und übernimmt die ganze Arbeit.	Der Fragende hangelt sich selbst weiter: Root → TLD (de) → autoritativer Server (thi.de); jeder verweist nur weiter.

Klausurtyp: DNS-Nachrichten als nummerierte Pfeile in eine Topologie einzeichnen. Typisch bei leeren Caches: Client →(rekursiv)→ Resolver →(iterativ)→ Root → TLD → autoritativer Server → Antworten rückwärts. Jede Anfrage kostet eine Umlaufzeit.

Schaubild — DNS-Auflösung (rekursiv + iterativ)



Client → Resolver ist rekursiv (1, 8): der Resolver liefert die fertige Antwort. Resolver → Root → TLD → autoritativ ist iterativ (2–7): jeder Server schickt nur den nächsten Verweis zurück.

Protokollzwecke (Frage 4.13)

Protokoll	Zweck
FTP	Dateiübertragung zwischen zwei Rechnern über TCP/IP.
Telnet	Interaktives Login auf einem fernen Rechner; *unverschlüsselt* (heute durch SSH ersetzt).
SNMP	Abfrage und Verwaltung von Netzwerkkomponenten (Netzwerkmanagement).

HTTP — wie eine Webseite geladen wird (Folie 06)

HTTP ist das Protokoll, mit dem der Browser Webseiten vom Server holt (Anfrage → Antwort). Es ist **zustandslos**: der Server merkt sich von sich aus nichts zwischen zwei Anfragen.

HTTP/1.0	HTTP/1.1
nicht-persistent: für jedes Objekt (HTML, jedes Bild) eine eigene TCP-Verbindung auf- und abbauen.	**persistent**: eine Verbindung für alle Objekte; optional **Pipelining** (mehrere Anfragen ohne auf jede Antwort zu warten).

Cookies: weil HTTP zustandslos ist, legt der Server eine kleine Kennung (Cookie) beim Client ab; bei jeder weiteren Anfrage schickt der Browser sie mit → der Server erkennt den Nutzer wieder (Login, Warenkorb).

Cache & Proxy: bereits geladene Objekte werden zwischengespeichert (Client-Cache oder Proxy-Server) → schneller, weniger Last. Steuerung per `If-Modified-Since` (Antwort 304 = noch gültig) und `Cache-Control: max-age=…`.

Ladezeit-Rechnung (Probeklausur 8)

Seite = 1 HTML + 4 Grafiken = **5 Objekte**. DNS-Lookup = RTT_ns; je Objekt 1 RTT_web; TCP-Aufbau T_setup, -Abbau T_term. Keine parallelen Verbindungen.

```
a) HTTP/1.0 (nicht-persistent): T = RTT_ns + 5·(T_setup + RTT_web + T_term) b) HTTP/1.1 (persistent): T = RTT_ns + T_setup + 5·RTT_web + T_term
```

```
Ersparnis = 4·(T_setup + T_term)
```

Persistent spart die 4 zusätzlichen Auf-/Abbauten; die 5 Anfrage-RTTs bleiben (ohne Pipelining) gleich. **Herleitung zeigen** — sie gibt die Punkte.

E-Mail-Protokolle (Folie 06)

Protokoll	Rolle
SMTP (Port 25)	**Versand** / Weiterleitung von Mails (Push). Textbasiert, über TCP, mit Verbindungsaufbau. Legt die Mail in der Ziel-Mailbox ab.
POP3 (Port 110)	**Abholen** vom Server auf den Client (meist Herunterladen + Löschen). Einfach.
IMAP4	**Abholen mit Serverhaltung**: Mails bleiben auf dem Server in Ordnern; mehrere Clients greifen auf denselben Bestand zu; zentrale Datensicherung.

> **Merksatz:** SMTP **schiebt** Mails hin (Versand), POP3/IMAP **holen** sie ab. POP3 = herunterladen, IMAP = auf dem Server lassen.

Praktische Übung: Shell & Batch (Blatt 1, Aufgabe 1.7)

Die **Shell** ist die Texteingabe, mit der man dem BS direkt Befehle gibt. Ein **Batchscript** (.bat) ist eine Datei mit mehreren solchen Befehlen, die nacheinander ausgeführt werden.

- `echo Text` gibt Text aus. `echo Text > datei` schreibt ihn in eine Datei.
- `@echo off` als erste Zeile unterdrückt das Anzeigen der Befehle selbst — nur die Ausgabe erscheint.
- Parameter im Script: `%1`, `%2` stehen für die mitgegebenen Argumente.

Umbenenn-Script: `renscr abc xyz`

```
@echo off
if "%2"=="" (echo Zu wenig Parameter &
exit /b 1)
ren *.%1 *.%2
```

Die `if`-Zeile fängt fehlende Parameter ab; `ren *.%1 *.%2` benennt alle Dateien mit Endung %1 in %2 um. Mit Endung .bat nicht arbeiten, damit sich das Script nicht selbst umbenennt.

★ Prüfungstaktik
90 Minuten · ein handbeschriebenes DIN-A4-Blatt · nicht-programmierbarer Taschenrechner

Zeitmanagement (90 Minuten)

- Klausur etwa **hälftig** Betriebssysteme (Themen 1–6) und Rechnernetze (7–11). Keinen Teil weglassen.
- Erst **alle Wissensfragen** einsammeln (4 OS-Aufgaben, Header, Protokolle, DNS) — schnelle, sichere Punkte.
- Dann die **Rechenaufgaben** mit festem Schema (Scheduling, Paging, Subnetting, TCP-Tabellen).
- Pro Aufgabe grob **1 Minute je Punkt**; bleibst du hängen, markieren und weiter.
- **Lösungsweg immer hinschreiben** — die Herleitung gibt Punkte, nicht nur das Endergebnis.

Häufige Punktebringer

- Bei Adress-/Paging-Aufgaben: **Trap/unzulässig** ausdrücklich vermerken, wenn Offset ≥ Länge.
- Bei Semaphoren: **Anfangswerte** immer angeben.
- Bei Subnetting: **Ausrichtung** prüfen (Subnetzadresse = Vielfaches der Blockgröße).
- Bei BEB/CSMA: $\{0 \ldots 2^n - 1\}$ hat $\mathbf{2^n}$ Werte.
- Seitenfehler erst **nach** der Erstbelegung zählen.

Aufs A4-Blatt gehört (Merksätze)

```
Wartezeit = Endzeit − Ankunft − Bedienzeit
Durchlaufzeit = Endzeit − Ankunft Seite=⌊Adr/Seitengr⌋, Offset=Adr mod Seitengr
nutzbare Hosts = 2^(32−Präfix) − 2 Adressen/Block = Blockgröße ÷ Adresslänge U = T_Frame /(T_Frame + 2·T_Prop) max. Fenster (n-Bit) = 2ⁿ−1 Bitrate = Schritt × log₂(Zustände) RTT = α·RTT_alt + (1−α)·M Fenster = RTT × Datenrate AckNo = SeqNo + Länge (+1 bei SYN/FIN) HTTP/1.0: RTT_ns + 5·(T_setup+RTT_web+T_term)
```

Reihenfolge der Verfahren (Seitenfehler): Optimal ≤ LRU ≈ Clock ≤ FIFO.

Vor der Abgabe: GANTT-Summe = Summe der Bedienzeiten? Subnetze überlappungsfrei? Bei jedem „Erläutern Sie“ wirklich eine Begründung geschrieben?