GBS-Lernzettel

Hauptaufgaben des Betriebssystems: Ressourcenverwaltung, Abstraktion

Arbeitsmodi eines Betriebssystems
- **Benutzermodus (user space):** Ausführung von Benutzer Prozessen im Benutzermodus, Kein direkter Zugriff auf Hardware und Ausführung von privilegierten Befehlen unmöglich
- **Systemmodus (kernel space):** Ausführung des Betriebssystemkerns im Systemmodus, Freie Zugriff auf Hardware und Ausführung von privilegierten Befehlen möglich

Systemaufrufen (Systemcalls)
Aufgabe: Systemcalls bieten dem Benutzermodus die Möglichkeit z.B. auf die Hardware zugreifen zu können
Ablauf: Anwendung ruft im Benutzermodus eine Bibliotheksfunktion auf (z.B. printf); Bibliotheksfunktion ruft den Systemcall auf; Der Systemcall unterbricht den Prozess und wechselt in den Systemmodus, in dem er ausgeführt wird; Nachdem der Systemcall beendet ist wird der Prozess im Usermodus weiter ausgeführt

Monolitische Betriebssystem (z.B. linux/ Unix): der Kernel wird vollständig (inklusive aller Treiber und Subsysteme) im Systemmodus ausgeführt ✅ Weniger Kontextwechsel, enthält generische Treiber
Mikrokern-basierten Betriebssystem (L4)**:** Teile des Kernel (Treiber, Subsysteme) in Benutzermodus auslagern ✅ nur ein kleiner Kernel im Systemmodus ausgeführt (besser wartbar, kleinere TCB, etc.)

Konzept der Linux Prozesshierarchie
- Jeder Prozess hat eine eindeutige ID (PID) die den Prozess identifiziert
- Jeder Prozess (bis aut Init) besitzt einen Elternprozess. Diese Zuordnung wird in der Parent PID (PPID) verwaltet
- Jeder Prozess inkl. Kinder formen eine Prozessgruppe (PGID)

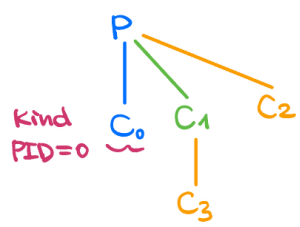
Zombie-prozess: wenn sich Kindprozess vor Elternprozess beendet und Elternprozess nicht auf das Ende des Kindprozesses wartet (getötet, aber bis zum Aufräumen noch im PCB erhalten) Dämonen sind Waisen-Prozesse
Waisen-Prozess: Kindprozess wird weiter (von init adopt) ausgeführt, obwohl sein Elternprozess bereits beendet

Elternprozesse können ihre Kind-Prozesse mit SIGSTOP verhungern lassen; Ein Kind-Prozess kann seinen Elternprozess mit kill() töten. Dämonen können Waisen töten

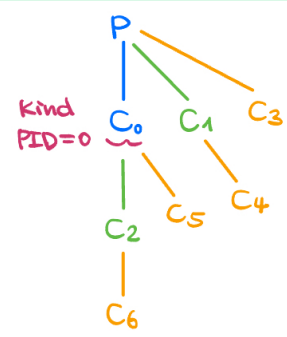
Prozesse besitzen jeweils einen eigenen, isolierten (virtuellen) Adressraum
Threads teilen sich Ressourcen ihres zugehörigen Prozesses (nur Programm Counter, stack, Register-Inhalt nicht)
Um Race Conditions zu vermeiden müssen Threads eines Prozesses **synchronisiert** werden

fork(): Bei Erfolg: Kind-PID im Elternprozess, 0 im Kind; Bei Fehler: -1 im Elternprozess, kein Kind
- jeder Prozess (inkl. Kindprozess) wird bei fork() einen neuen Prozess bzw deren selben Kindprozess erzeugen
- Der Programmierer hat vergessen **break**-Anweisungen am Ende aller switch-**case**-Konstrukte hinzuzufügen, weswegen der **default** case auch vom Parent ausgeführt wird
- **main** selber als ein Thread gezählt auch wenn gar kein fork()

```c
int main(void) {
    pid_t pid = fork();
    switch ( pid ) {
    case -1: return EXIT_FAILURE;
    case 0:  printf("PID=%d\n", pid); break;
    default: printf("PID=%d\n", pid);
             fork(); fork();
    }
    return EXIT_SUCCESS;
}
```




```c
int main(void) {
    pid_t pid = fork();
    switch ( pid ) {
    case -1: return EXIT_FAILURE;
    case 0:  printf("PID=%d\n", pid);
    default: printf("PID=%d\n", pid);
             fork(); fork();
    }
    return EXIT_SUCCESS;
}
```



Scheduler: wählt den nächsten auszuführenden Prozess aus der Menge der rechenbereiten Prozess
- Vorteile von User-Level-Threads gegenüber Kernel-Level-Threads: Schneller Threadwechsel, Schneller Erstellen

Dispatcher: setzt den eigentlichen Kontextwechsel um

Ablauf eines Kontextwechsels - Die Aufgabe wird vom Dispatcher übernommen
1. Ändert den Zustand des rechnenden Prozesses zu wartend oder rechenbereit
2. Sichert den Kontext des zuvor rechnenden Prozesses/ Threads im PCB
3. Lädt den Kontext des rechenbereiten Prozesses/Threads
4. Ändert den Zustand des rechenbereiten Prozesses zu rechnend

Bei Kontextwechsel sollte **Seitentabellenregister** auf die **Adresse** der verwendeten **Seitentabelle** setzen

Für Kontextwechsel benötigte Datenstrukturen

- **Run Queue:** verwaltet Prozesse, die sich im Zustand **rechenwillig** befinden. Der Scheduler entscheidet, welcher dieser Prozesse als nächstes die CPU bekommen soll
- **Wait Queue:** verwaltet Prozesse, die im Zustand **wartend** bzw. **blockiert** sind. Der Dispatcher legt blockierende Prozesse in die Wait Queue

Preemptive Scheduling (unterbrechend): Prozess kann vom Scheduler unterbrochen werden
Non-Preemptive Scheduling (nicht unterbrechend): Der aktuell aufführende Prozess wird so lange ausgeführt bis er terminiert oder blockiert

Wann wird ein unterbrechender Scheduler aktiv ? 1) Wenn ein **neuer Prozess erzeugt** (Der neue Prozess wird nicht automatisch ausgeführt, Der Scheduler bestimmt wer als nächstes ausgeführt wird in Abhängigkeit der Schedulingstrategie); **2)** Prozess **terminiert**; **3)** Prozess **blockiert**; **4) Interrupt** auftritt

Alle Systeme haben **Skalierbarkeit** bei Schedulingstrategie, **Fairness** und **Balance als Optimierungsziele**

Scheduling-Strategien

- **Scheduling in Batch-Systemen (-> Durchsatz, minimale Ausführungszeit, konstante CPU-Belegung)**
  - **First-Come-First-Served (FCFS):** Prozesse werden in Reihenfolge ihrer Ankunftszeiten abgearbeitet
  - **Shortest Job First (SJF):** Auswahl des nächsten Prozesses mit der kürzesten Rechenzeit
  - **Shortest Remaining Time Next (SRTN):** unterbrechend nur bei Eintreffen neuer Prozesse, Ausführungszeit bekannt (praktisch unmöglich), Auswahl des nächsten Prozesses mit der kürzesten verbleibenden Rechenzeit
- **Scheduling in interaktiven Systemen (-> minimale Antwortzeit, Proportionalität)**
  - **Round-Robin Scheduling (RR):** unterbrechend nach Ablauf der Zeitscheibe, jeder Prozess folgen Zeit Quantum, danach CPU-Abgabe (vorzeitige CPU-Freigabe, falls fertig oder blockiert), zyklischer Abarbeitung der Prozesse (entweder Sortierung nach Index oder nach Ankunftszeiten)
    - **Kurze Zeitquanten:** häufigere Kontextwechsel, schlechterer Effizienz, besserer Fairness u. Reaktionszeit
    - **Lange Zeitquanten:** seltenere Kontextwechsel, besserer Effizienz, schlechterer Fairness u. Reaktionszeit
  - **Priority Scheduling:** höher priorisierte Prozesse haben Vorrang
    - **Statische Priorität:** Prioritäten ändern sich nicht, Risiko zu Verhungern
    - **Dynamische Priorität:** Neuberechnung nach z.B Timerinterrupt
- **Scheduling in Echtzeit-Systemen (->Deadlines einhalten: Datenverlust vermeiden,Vorhersagbarkeit)**
  - **Earliest Deadline First (EDF):** Die CPU wird dem Prozess mit der frühsten Deadline zugeordnet
    - **Nicht-unterbrechend:** Prozess so lange ausgeführt bis fertig oder blockiert. Neue Prozesse mit früherer Deadline erst bei nächstem Kontextwechsel berücksichtigt
    - **unterbrechend:** unterbrochen sobald einer mit früherer Deadline rechenwillig reinkommt
  - **Rate-Monotonic Scheduling (RMS):** statische Priorität von Prozessen basierend auf ihrer Periodendauer festgelegt, wobei kürzere Perioden entspricht höhere Prioritäten

Wartezeit W: Anzahl der Wartezeiten "-"
Verweilzeit V: Alle Zeiteinheiten vom Beginn zum Ende (inkl. "-", "x" sowie blockierte Zeiteinheiten wegen I/O)

RR-Scheduling verteilt CPU-Zeit in festen Zeitscheiben **gleichmäßig** an alle Prozesse - bei vielen Prozesse **ineffizient**
CFS passt die Zuteilung **dynamisch** an, basierend auf **Priorität** u. **Ausführungsbedarf** der Prozess - **effizienter, fairer**

mittels Kommandozeilenbefehls nice
für Prozessen können die Priorität durch einen niceness-Wert dynamisch angepasst werden
beim höheren höheren (niedrigeren) Wert verringern (erhöhen) sich die Prioritäten

Completely-Fair-Scheduler (CFS)

1. Wähle den nächsten rechenwilligen Prozess

-- mit niedrigster virtual runtime vt_i (falls mehrere gleiche vt_i -> mit kleinster index)

2. Berechnen time slice TS_i bei n aktiven Prozessen: $TS_i = TL * \frac{w_i}{\sum_{j=1}^{n} w_j}$

-- die bereits terminierende Prozesse sollen von der Summe ausgeschlossen sein

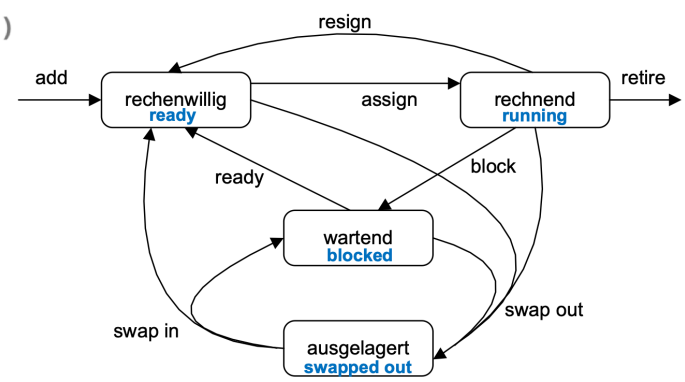
3. Aktualisiere real runtime rt_i mit $rt_{i_new} = rt_{i_old} + TS_i$

-- real runtime rt_i darf die maximale Rechenzeit nicht überschreiten

4. Aktualisiere t in diesem Schritt: $t_{new} = t_{old} + (rt_{new} - rt_{old})$

5. Aktualisiere virtual runtime vt_i mit $vt_{i_new} = vt_{i_old} + \frac{w_0}{w_i}(rt_{i_new} - rt_{i_old})$, **mit** $w_0 = 1024$

--> TS_i und vt_i Werte sollen hier normalerweise aufgerundet werden - z.B. $3,27 -> 4$



Consumer muss warten wenn Buffer **leer**, **Producer** muss warten wenn Buffer **voll**
Kritischer Abschnitt: Prozesse greifen auf gemeinsame Ressourcen zu (down = lock, up = unlock)

```
//Deklarationen
  GF_voll(1);          // Zaehler Zuege in GF, verhindert weniger als 0 Zuege in GF
  GF_frei(1);          // Zaehler freie Plaetze in GF, gegenlaeufig zu GF_voll
  F_voll(3);           // Zaehler Zuege in F, verhindert weniger als 0 Zuege in F
  Baustelle(1);        // Mutex fuer den eingleisigen Abschnitt
  Abschnitt_GF_GH(2);  // Semaphor fuer den Abschnitt GF <-> GH
// ---
```

up: erhöht Counter
-> notify alle die in einem down counter

down: verringert Counter
-> Falls Counter = 0, wartet

```
// Thread
Fahre_in_richtung_F
{
  down(GF_voll);
  down(Abschnitt_GF_GH);

  <Fahre aus GF aus>

  up(GF_frei);

  <Fahre in GH ein>

  up(Abschnitt_GF_GH);
  down(Baustelle);

  <Fahre aus GH aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in F ein>

  up(Baustelle);
  up(F_voll);

}
```

kritischer Abschnitt

```
// Thread
Fahre_in_richtung_GF
{

  down(GF_frei);
  down(F_voll);
  down(Baustelle);

  <Fahre aus F aus>

  <Fahre durch eingleisigen Abschnitt>

  <Fahre in GH ein>

  up(Baustelle);
  down(Abschnitt_GF_GH);

  <Fahre aus GH aus>

  <Fahre in GF ein>

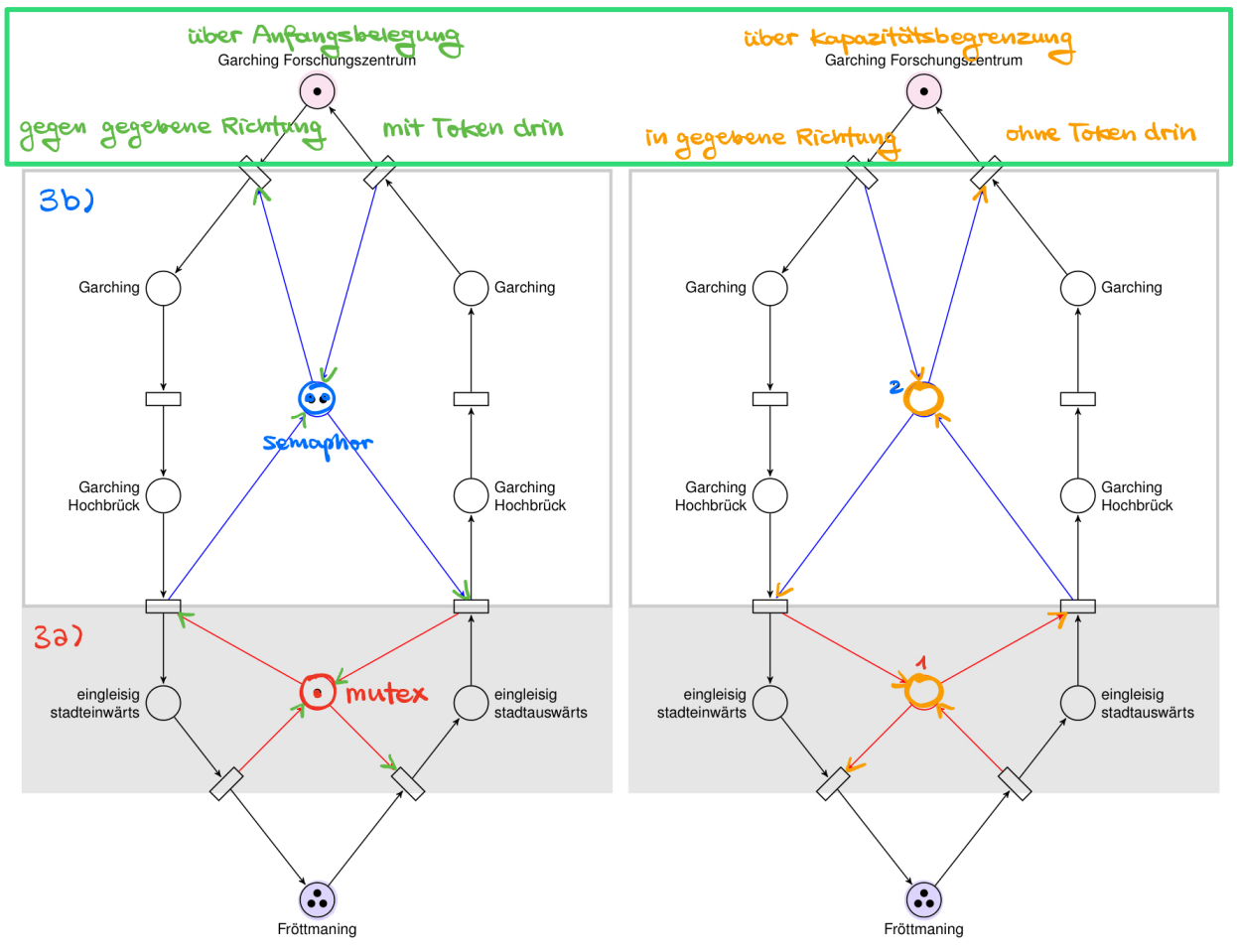
  up(GF_voll);
  up(Abschnitt_GF_GH);

}
```

kritischer Abschnitt

keine anderen down & up innerhalb kritischen Abschnitts erlaubt

Mutex: ermöglicht ausschließlich exklusiven Zugriffe - wenn ein Thread ein mutex sperrt, haben andere Threads keinen Zugriff, bis der mutex freigegeben wird
Semaphore: erlaubt variable Anzahl von Zugriffe - bei einem zählenden Semaphore können mehrere Threads gleichzeitig auf eine Ressourcen zugreifen, bis das Limit der Zählvariable erreicht wird (besitzt Zählvariable, Warteschlange, Maximalwert)



Lösung ohne Kapazitätsbegrenzung / mit Anfangsbelegung

Lösung ohne Anfangsbelegung / mit Kapazitätsbegrenzung

Abbildung 2: U-Bahn Streckenabschnitt Garching Forschungszentrum ⟺ Fröttmaning

Bedingungen für das Auftreten von Deadlocks

- **Mutual Exclusion:** Die gemeinsam genutzten Ressourcen sind nur exklusiv nutzbar, d.h. eine Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet
- **Hold-and-Wait:** Prozesse, die bereits Ressourcen zugeteilt haben, belegen diese Ressourcen, auch wenn sie noch weitere Ressourcen anfordern
- **No-Preemption:** Zugeteilte Ressourcen können einem Prozess nicht entzogen werden, d.h. die Ressourcennutzung ist nicht unterbrechbar
- **Circular Wait:** Es gibt eine zyklische Kette von min. zwei Prozessen, die jeweils auf eine Ressource warten, die vom nächsten Prozess in der Kette belegt ist

Stelle-Kapazität c: Zahl neben Stelle (default ∞); **Kante-Gewichtung w:** Zahl neben Kante (default 1)
Boolesches Netz: Jede Stelle hat Kapazität 1 und jede Transition hat Gewicht 1
Transition ist **schaltbereit**, wenn alle Vor- und Nachbedingungen erfüllt: Die Stellen aller eingehenden Kanten haben **genügend Tokens**; die Stellen aller ausgehenden Kanten haben **genug freien Platz**
Es schaltet **immer nur eine Transition** zu einer Zeit -> Mehrere Transitionen schalten nie gleichzeitig

Petrinetze-Eigentschaften

- **Erreichbar:** M_n ist von M erreichbar, wenn eine Sequenz von M nach M_n existiert
- **lebendig** -> $\forall M\ \forall t \in T$: t kann in Zukunft wieder Schalten (implizit verklemmungsfrei)

<-> **nicht Lebendig:** ab einem Zeitpunkt/Belegung **irgendeine** Transition nie wieder schalten können wird

- **Verklemmt:** ab einem Zeitpunkt/Belegung **alle** Transitionen nicht mehr schalten können (implizit nicht lebendig)
  - Sofern es im Erreichbarkeitsgraph zum Absturz bzw. Fangzustand bzw. Blatt-Knote kommt

<-> verklemmungsfrei -> $\forall M\ \exists t \in T$: t kann Schalten

- **Fair:** wenn keine Transition verhungern - müssen nie zwischen mehreren schaltbereiten Transitionen entscheiden
  - Verhungern: Ausführung zwar möglich, könnte aber unendlich lange hinausgezögert/ vermieden werden
    - Bsp. es schaltet immer nur t_y, deshalb kann t_x verhungern, obwohl t_x schaltbereit ist

Konstruktion: Erreichbarkeitsgraph -> Petrinetz
jede Transition t_x und deren Zustände Z_{vor} & Z_{nach} betrachten (Kanten-Gewicht = Belegung-Differenz der Zustände)
- Falls Belegung der Stelle s_i im Zustand Z_{vor} größer als im Zustand Z_{nach} - d.h. $Z_{vor}(M(s_i)) > Z_{nach}(M(s_i))$, Kante von Stelle s_i zu t_x
- Falls Belegung der Stelle s_i im Zustand Z_{vor} kleiner als im Zustand Z_{nach} - d.h. $Z_{vor}(M(s_i)) < Z_{nach}(M(s_i))$, Kante nach Stelle s_i aus t_x

Deadlock sind Prozesse blockiert, da sie auf Ressourcen warten, die zugleich von anderen Prozessen blockiert
Deadlock-Detection: erkennt, ob aktuell ein Deadlock vorliegt und beseitigt einen solchen gegebenfalls
Deadlock Prevention: sorgt dafür, dass eine der vier Deadlock Bedingungen nicht erfüllt ist
Deadlock Avoidance: Betriebssystem prüft dynamisch, ob eine Ressourcen-Anforderung erfolgen kann oder nicht (gegebenenfalls werden Ressourcen zurückgehalten, bis es der Fall ist) - dazu kann Bankier-Algorithmus eigesetzt

Livelock: Prozesse geben freiwillig belegte Ressourcen ab, sobald sie merken, dass sie die nächste Ressource nicht belegen können. Sie blockieren nicht, sondern versuchen wieder die erforderlichen Ressourcen zu belegen. Obwohl beide Prozesse weiterhin ausgeführt werden, machen sie keinen Fortschritt
Starvation: Ein Prozess kann verhungern, wenn seine Ressourcenanfrage nie erfüllt wird, da stets andere (höher priorisierte) Prozesse Vorrang haben

das zu einem Erreichbarkeitsgraphen zugehörige Petrinetz ist nicht eindeutig, da es können niemals schaltbereite Transitionen hinzugefügt werden, ohne den Erreichbarkeitsgraphen zu ändern

Implizite Kommunikation über gemeinsame Betriebsmittel: schneller, einfacher z.B. Shared Memory
explizite Kommunikation: Versenden u. Empfangen von Nachrichten - geeignet für Prozesse in disjunkte Adressräume

Signal (schmalbannig, implizit, asynchron): Benachrichtigungsmechanismus
- unterbricht den Kontrollfluss eines Prozesses zu beliebigen Zeitpunkte

Pipes (breitbandig, explizit, asynchron): ein unidirektionaler Strom zwischen zwei Kommunikationspartnern
- im Gegensatz zu unnamed Pipes können **named pipes (FIFOs)** von beliebigen Prozessen verwendet werden, die entsprechende Rechte auf die Pipe-Datei haben
- **Befehl** `grep «Dateiname» «Dateipfad» | Aktion der Linux Kommandozeile`

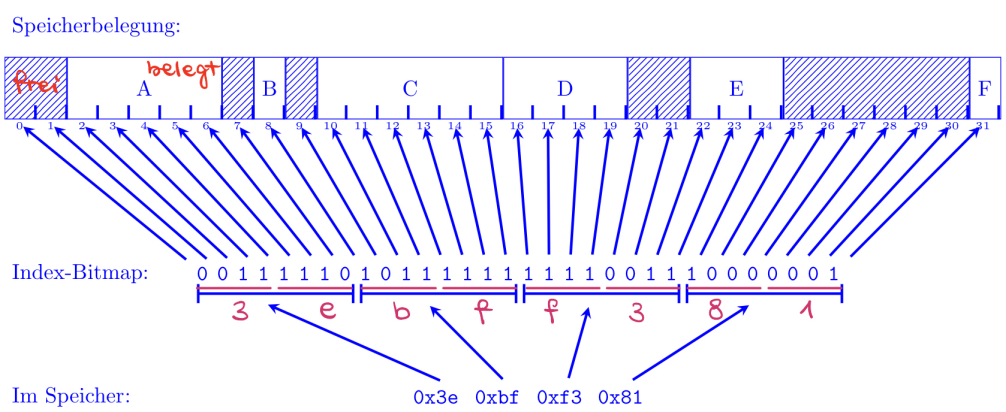
Socket (breitbandig, explizit): bidirektionaler Kommunikationskanal

Falls zu send()-Zeitpunkt Empfänger noch nicht existiert, send() return Fehler

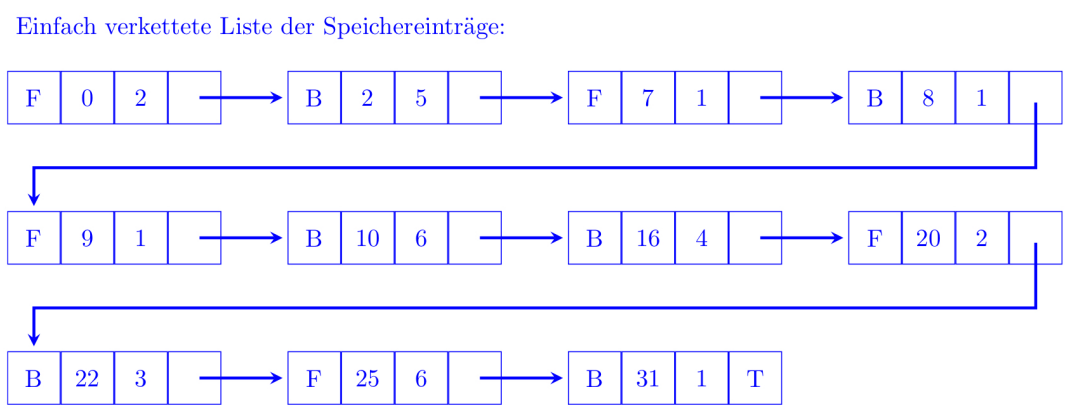
- **Asynchrone Kommunikation:** send() wartet nicht auf eine Bestätigung vom Empfänger, erfordert Puffer
  - Prozess p zwar existiert, aber receive noch nicht bereit, p kann Nachricht erst empfangen wenn receive bereit
- **synchroner Kommunikation:** blockiert Sendevorgang, bis Sender eine Bestätigung vom Empfänger erfolgreich erhalten, Puffer nicht zwingend erfordert
  - Bestätigung wird nach Verarbeitung vom Empfänger automatisch zu sender gesendet, sender braucht keine recevie bereit für Bestätigung

Datenstrukturen für Speicherverwaltung

- **Bitmaps:** Speicher in gleich große Blöcke aufgeteilt und jeder Block mit einem Bit repräsentiert (1 = belegt; 0 = frei)
  - **Problem:** Blockgröße-Wahl **a) zu klein** -> benötigt große Bitmap; **b) zu groß** -> Speicherverschwendung
  - ✅ einfache + schnelle Zugriff auf Speicherblöcke, ❌ Aufwendige Suche nach k aufeinander folgenden Blöcken



- **Verkettete Liste:** Mehrere zusammenhängende Speicherblöcke in Bereichen zusammengefasst & in Listen verwaltet
  - **1) B** = belegt; F = frei; **2)** Anfangsadresse; **3)** Länge; **4)** zeigt auf nächsten Eintrag/ Terminator T (NULL)
  - ✅ flexible Speicheraufteilung, ❌ lineare Suche, Keine feste Verwaltungsstruktur (dynamisch wachsen)



Bei der Fragmentierung handelt es sich um Speicherbereiche, die nicht mehr verwendet werden können

Fragmentierung: ineffiziente Nutzung von Speicherplatz, verursacht durch Aufteilung des Speicherplatzes in kleine, unzusammenhängende Einheiten

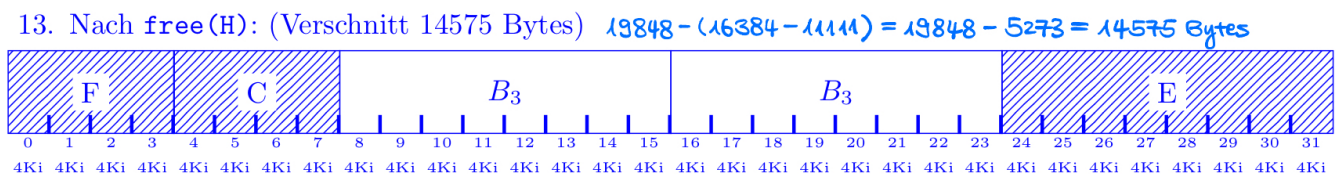
Interne Fragmentierung: Ungenutzte Bereiche eines Blocks (wenn Dateigröße < Blockgröße)

Externe Fragmentierung: Lücken nicht vollständig ausgenutzt - wenn freier Speicher in kleinen, getrennten Lücken existiert, die zusammen ausreichend wären, aber einzeln keine Anfragen erfüllen können.

Buddy-Algorithmus: B_n mit 2^n Blockeinheiten

- **Einfügen:** teilt Speicher durch rekursiven Halbieren, bis kleinstmögliche passend Block für Datenmenge finden
- **Löschen:** Inhalt löschen, Block als frei markieren und ihn mit freien Buddys zu größeren Blöcken verschmelzen

✅ leicht und effizient ❌ die zwei B_3 zwar benachbart, aber nicht Buddy-Paar, kann daher nicht verschmolzen



Belegungsstrategien für Freispeicherverwaltung

- **First-Fit:** Wahl des nächsten freien Speicherbereichs, beginnend beim Anfang des Speichers
- **Next-Fit:** Wahl des nächsten freien Speicherbereichs, beginnend beim letztem vergebenem Bereich

Anfrage	Liste freier Speicherbereiche				
	100 KiB	400 KiB	250 KiB	200 KiB	50 KiB
30 KiB	**70 KiB**	400 KiB	250 KiB	200 KiB	50 KiB
60 KiB	**10 KiB**	400 KiB	250 KiB	200 KiB	50 KiB
120 KiB	10 KiB	**280 KiB**	250 KiB	200 KiB	50 KiB
20 KiB	10 KiB	**260 KiB**	250 KiB	200 KiB	50 KiB
100 KiB	10 KiB	**160 KiB**	250 KiB	200 KiB	50 KiB
10 KiB	**0 KiB**	160 KiB	250 KiB	200 KiB	50 KiB
250 KiB	0 KiB	160 KiB	**0 KiB**	200 KiB	50 KiB

Anfrage	Liste freier Speicherbereiche				
	100 KiB	400 KiB	250 KiB	200 KiB	50 KiB
30 KiB	**70 KiB**	400 KiB	250 KiB	200 KiB	50 KiB
60 KiB	**10 KiB**	400 KiB	250 KiB	200 KiB	50 KiB
120 KiB	10 KiB	**280 KiB**	250 KiB	200 KiB	50 KiB
20 KiB	10 KiB	**260 KiB**	250 KiB	200 KiB	50 KiB
100 KiB	10 KiB	**160 KiB**	250 KiB	200 KiB	50 KiB
10 KiB	10 KiB	**150 KiB**	250 KiB	200 KiB	50 KiB
250 KiB	10 KiB	150 KiB	**0 KiB**	200 KiB	50 KiB

- **Best-Fit:** Wahl des Speicherbereichs mit dem geringsten Verschnitt
- **Worst-Fit:** Wahl des Speicherbereichs mit dem größten Verschnitt

Anfrage	Liste freier Speicherbereiche				
	100 KiB	400 KiB	250 KiB	200 KiB	50 KiB
30 KiB	100 KiB	400 KiB	250 KiB	200 KiB	**20 KiB**
60 KiB	**40 KiB**	400 KiB	250 KiB	200 KiB	20 KiB
120 KiB	40 KiB	400 KiB	250 KiB	**80 KiB**	20 KiB
20 KiB	40 KiB	400 KiB	250 KiB	80 KiB	**0 KiB**
100 KiB	40 KiB	400 KiB	**150 KiB**	80 KiB	0 KiB
10 KiB	**30 KiB**	400 KiB	150 KiB	80 KiB	0 KiB
250 KiB	30 KiB	**150 KiB**	150 KiB	80 KiB	0 KiB

Anfrage	Liste freier Speicherbereiche				
	100 KiB	400 KiB	250 KiB	200 KiB	50 KiB
30 KiB	100 KiB	**370 KiB**	250 KiB	200 KiB	50 KiB
60 KiB	100 KiB	**310 KiB**	250 KiB	200 KiB	50 KiB
120 KiB	100 KiB	**190 KiB**	250 KiB	200 KiB	50 KiB
20 KiB	100 KiB	190 KiB	**230 KiB**	200 KiB	50 KiB
100 KiB	100 KiB	190 KiB	**130 KiB**	200 KiB	50 KiB
10 KiB	100 KiB	190 KiB	130 KiB	**190 KiB**	50 KiB
250 KiB	Kein geeigneter Speicherplatz vorhanden				

Direkter Adressierung: Jede logische Adresse entspricht ihrer physischen Adresse
Basisadressierung: physischen **Zieladresse = Basisadresse + Offset** bzw. logische Adresse
- **Länge** = mindestens **Offset + 1** so groß -> Sonderfall: Offset=0, Länge mindestens 1

Seitenadressierung (Paging-Konzept): Betriebssystem verwaltet eine Seitentabelle, die virtuelle Adressen auf physische Adressen abbildet bzw. Abbilden von Seiten auf Kacheln

Größe der Physisch-Adresse = Größe des (Haupt-/ Arbeits-)Speichers
x-bits virtuell-Adresse = x-bits Architektur
Speichergröße <-> Adressbits & Seiten-/Kachelgröße <-> Offsetbits
- Bei 2^X (adressierbaren) Bytes -> X Bits für die Adresse/das Offset

Virtuelle Adresse = [Seitennummer] + [Offset_innerhalb_der_Seite]
Physische Adresse = [Kachelnummer] + [Offset_innerhalb_des_Kachels]
- Bits für virtuelle/ physische Adresse = Bits für Seiten-/ Kachel-Nummer + Bits für Offset
- **Offset** bei beiden **identisch** bzw. **Seitengröße = Kachelgröße**

Seiten-/Kachelanzahl <-> Seiten-/Kachelbits
- Bei 2^Y Seiten/Kacheln -> Y Bits für die Seiten-/ Kachel-Nummer
- Speichergröße ÷ Seiten-/Kachelgröße = Seiten-/Kachelanzahl (<-> Seiten-/Kachelbits)
- Speichergröße ÷ Seiten-/Kachelanzahl = Seiten-/Kachelgröße (<-> Offset-bits)

Binär			Dezimal
Kibi	$1024 = 2^{10}$	$1000 = 10^3$	Kilo
Mebi	2^{20}	10^6	Mega
Gibi	2^{30}	10^9	Giga
Tebi	2^{40}	10^{12}	Tera

Mittels der **einstufige** und **mehrstufige** Übersetzung kann **gleich so viel Speicher adressiert** werden
-> Mehrstufiges Paging spart Speicher - lässt ungenutzte Teile des Adressraums ohne zugehörige Tabellen
a-stufiges Paging mit x bits pro Stufen -> 2^x Einträge pro Stufe und 2^{ax} Pages insgesamt adressiert

physische -> virtuelle: finde entsprechend der Kacheln den **Eintrag in letzt-stufige Pagetable,** und jeweilige Pagetable-Pfad als Seiten in entsprechende Stufe einsetzen

virtuelle -> physische: finde **laut des jeweilige Pagetables** den Eintrag als Kachel (nicht gefunden -Pagefault)

Größe einer Tabelle, die für den zweiten Schritt der Adressübersetzung verwendet wird in Bits: 2^4*4= 16*4=64

Erklärung: Da 4 Bit im zweiten Übersetzungsschritt für Tabelle-Lookup verwendet, hat eine solche Seitentabelle 16 Einträge. Jeder dieser Einträge benötigt genau so viel Platz, wie Bits in der physischen Adresse vorhanden sind, bzw. Kachel-bits (=4)

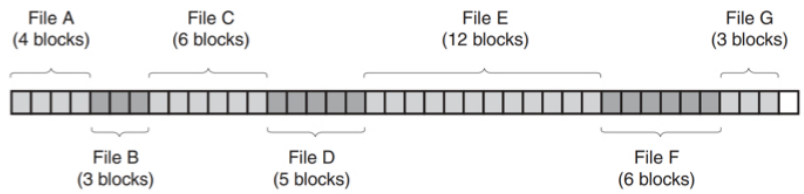
Probleme bei direkter Verwaltung des physischen Arbeitsspeichers: 1) Jedes Programm belegt potentiell den gesamten Hauptspeicher; 2) Adressraum eines Prozesses beschränkt durch Größe des physischen Speichers

Seitenersetzungsstrategien

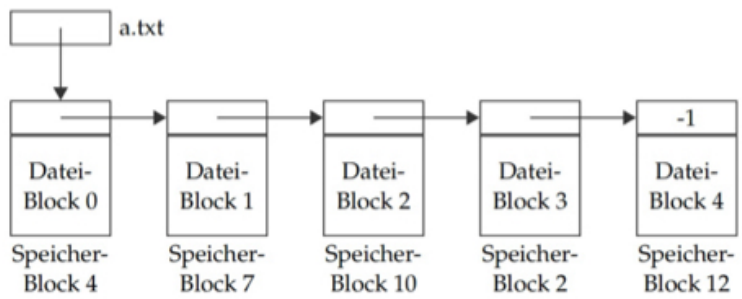
- **FIFO (First-In-First-Out):** Immer die Seite wählen, die am längsten im Speicher gehalten wurde: Kopf der Liste
  - ❌ **Belady-Anomalie:** Erhöhen der Kachelzahl kann sogar zu mehr Seitenfehlern führen (anstatt zu weniger)
- **Clock(FIFO-Second-Chance)** nutzt eine zirkuläre Liste mit Uhrzeiger - ersetzt die erste älteste Seite mit R=0 (unterwegs alle R=1 auf 0 zurücksetzt) und Zeiger zur nächsten Seite, wobei Änderungen bei M=1 zuerst auf Hintergrundspeicher gespeichert bzw. auf 0 setzen) - falls kein Pagefault, Zeiger bleibt unverändert
  - **Lesezugriff:** R-Bit = 1, M-Bit = 0; **Schreibzugriff:** R-Bit = 1, M-Bit = 1
- **Least Recently Used (LRU):** aktuell am längsten nicht genutzte Seite wird bei Pagefault ersetzt - 🔥 letzt-Zugriffszeit
- **Not Frequently Used (NFU):** aktuell am wenigsten genutzte Seite wird bei Pagefault ersetzt - 🔥 Zugriff-Häufigkeit
- **Aging-Verfahren (approximiert LRU):** Timer-Interrupt nach Ablauf des festgelegten Zeitintervalls t (in dem Seitenzugriffe erfasst werden - z.B. 20 ms); dann Zählerstand von A_p **neu berechnen** (**Shift des Zählers A_p um eine Stelle nach rechts** und **Addition des R-Wertes zum am linkesten stehenden Bit** von Zähler A_p

Dateiverwaltung: verwalten die Datei-Inhalte speichernde Blöcke

- **Contiguous Allocation:** Datei wird als zusammenhängende Folge von Blöcken auf Festplatte verwaltet
  - ✅ Einfach zu implementieren und Performanz bei Lesen
  - ❌ externe Fragmentierung wegen gelöschten Bereichen

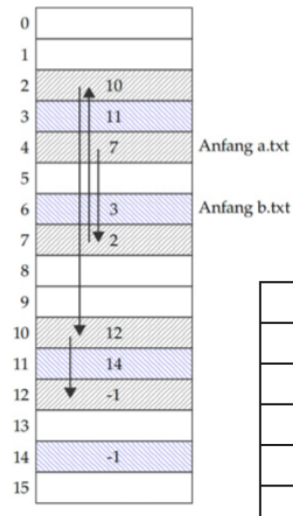


- **Linked List Allocation:** Alle Speicherblöcke werden innerhalb einer verketteten Liste mit Zeiger verwaltet
  - ✅ kein Speicherplatz verloren
  - ❌ Niedrige Performance bei Random-Access, zum Lesen von 1 Block, müssen 2 Blöcke geladen

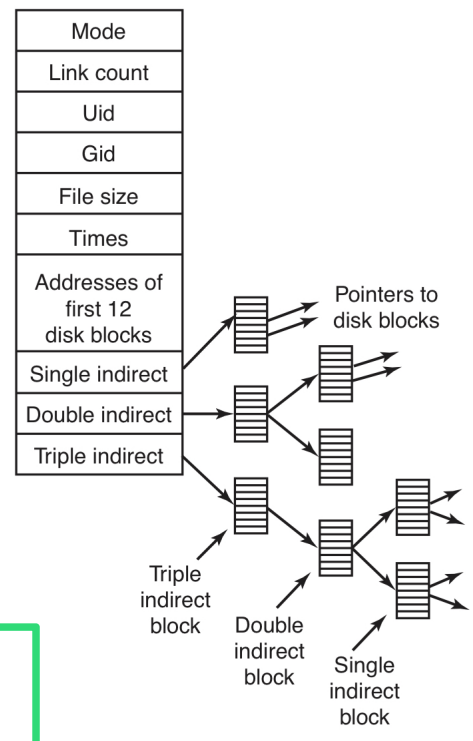


- **Linked List Allocation mit File Allocation Table (FAT):** Zeiger auf nächsten Block wird in Tabelle im Hauptspeicher verwaltet (Spezialfall -1 steht für End of File)

  1 Eintrag pro Block
  - ✅ Kein Verschnitt; Random-Access schneller
  - ❌ zu viel Speicherplatz (FAT muss vollständig im Speicher gehalten)
    - **FAT12 -> 12 bits pro Eintrag**
    - **Geeignet für viele kleine Dateien**



- **i-nodes:** Datei durch Index-Node-Datenstruktur repräsentiert
  - Typ, Dateirechte, Dateigröße, UID, GID, Änderungsdatum, links count, Flags, Adressen der referenzierten Blöcke (single-, double-, triple-indirekt)
  - ✅ es müssen nur die i-nodes der aktuell geöffneten Dateien im Speicher gehalten
  - ❌ Jeder i-node enthält eine fest Anzahl an Blockadressen - begrenzt max. Dateigröße
    - **Geeignet für wenige kleine Dateien u. dynamisch erweiterte Dateien**



Anfangsadresse der n-tes Inode = Anfangsadresse Inodes + (n-1)* InodeGröße
Eintrag in direktDatenblock D_0 = Offsetnummer in Datenblock
Blocknummer im gesamten Dateisystem = Blockgröße * Offsetnummer im Datenblöcke

Aufgabe 2 Vergleich: i-nodes und FAT

Auf einer 16 GiB-Festplatte werden Dateien unter UNIX mittels i-nodes gespeichert. Eine i-node verfüge im Folgenden über 12 direkte Referenzen auf Datenblöcke, einen einfach indirekten Verweis und einen zweifach indirekten Verweis. Die Blockgröße betrage 1024 Bytes und die Adresslänge 32 Bit.

a)* Wie viele Blöcke Verwaltungsaufwand benötigt eine 250 KiB-Datei?

1. 250 KiB Nutzdaten benötigen 250 Datenblöcke à 1 KiB.
2. 12 Blöcke werden direkt über die i-node adressiert. Diese liegt (neben anderen i-nodes) ebenfalls auf einem Datenblock. $250 - 12 = 238$ Blöcke übrig
3. Die restlichen 238 Blöcke werden mittels single-indirect Verweisen verwaltet. Pro Datenblock sind hierbei jeweils
$$\frac{1024 \text{ Byte}}{32 \text{ bit}} = 256 > 238 \Rightarrow \text{noch ein Datenblock}$$
Verweise möglich.
4. Insgesamt sind also 2 Blöcke Verwaltungsaufwand und 250 Blöcke für Nutzdaten nötig.

$16\,\text{GiB} = 16 \cdot 2^{30} = 2^4 \cdot 2^{30} = 2^{34}$ Bytes

b)* Welchen Umfang würde eine FAT (File Allocation Table) für die obige Festplatte haben? Verwenden Sie die kleinstmögliche Anzahl von Bits für die Blocknummern.

$$\#\,\text{Blöcke} = \frac{2^{34}\,\text{Bytes}}{2^{10}\,\text{Bytes/Block}} = 2^{24}$$

Die File Allocation Table ist ein Array, das für jeden Block der Festplatte einen Eintrag besitzt. Die Festplatte hat 2^{24} Blöcke der Größe 1 KiB. Daraus folgt, dass die Tabelle 2^{24} Einträge der Größe 24 bit (3 Byte) verwalten muss. ⇒ Die FAT ist $2^{24} \cdot 24$ bit = $2^{24} \cdot 3$ Bytes = 48 MiB groß.

Konzept "everything is a file"- unter Unix-Systemen unterstützt Dateitypen
- **Dateien** (Binärdatei, Textdatei) beinhalten Benutzerinformationen - **Symbol "-"** 8
- **Verzeichnisse** sind Systemdateien zur Verwaltung des Dateisystems - **Symbol "d"** 4
- **Character Special** Files: modellieren serielle I/O Geräte wie Terminal, Drucker - sequentiell zugreifen
- **Block Special Files:** modellieren Massenspeicher - Festplatten, SSD, USB-Speicher - Zugriff auf beliebigAdresse
- **Socket und Named Pipe**

Ob A in eine Datei file.txt schreiben darf: erst schauen ob A auf die **Verzeichnisse** der Datei zugreifen darf (= ob x bit bei Verzeihnis gesetzt). falls nicht dann darf nicht in file.txt schreiben. falls ja dann schauen ob A User oder in Group oder alle anderen enthalten somit deren Zugriffsecht überprüfen

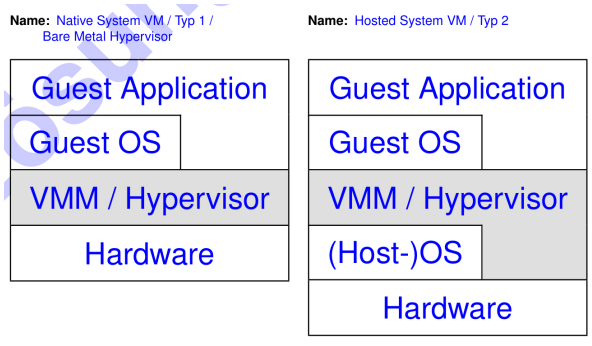
Zugriffsrechte bei reguläre Datei/ Verzeichnisse
read (r, 4): Inhalt lesen/ kopieren; Namen gespeicherter Dateien auflisten
write (w,2): Inhalt hinzufügen/ verändern/ löschen; (nur wenn x auch gesetzt ist) gespeicherte Dateien erstellen / verschieben /umbenennen/ löschen d.h. -w- selber erlaubt keinen Zugriff auf das Verzeichnis
execute (x,1): Als Programm ausführen; Zugriff auf Metadaten von bereits existierende Dateien - Nutzen der Dateien im Verzeichnis entsprechend dieser Datei-Rechte lesen/ bearbeiten/ ausführen

Hardlinks: Das Verzeichnis verweist direkt auf die (erste) i-node der referenzierten Datei
- dekrementiert Link-Counter bei Löschen eines Eintrags, wobei Daten erst bei Link-Counter=0 gelöscht werden

Softlink (Symbolische Links): erzeugt eine zusätzliche Datei durch Angabe des Pfadnamens als Datei-Verweis
- Bei Löschen/ Verschiebung der Zieldatei wird Link invalide; ohne den Link-Count zu beeinflussen **relativer Pfad**

Der Name einer Datei ist **in den Daten des Inodes gespeichert**, der das Verzeichnis darstellt, in dem die Datei referenziert wird. Dies liegt daran, dass eine Datei als Hardlink in mehreren Verzeichnissen mit unterschiedlichen Namen verlinkt sein kann.
Wenn Berechtigungen in den Directory-Entries statt in I-Nodes gespeichert werden, könnten Hardlinks auf dieselbe Datei **unterschiedliche** Berechtigungen haben, was individuelle Zugriffsrechte je nach Verzeichnispfad ermöglicht

Virtualisierungsmöglichkeiten
- **Virtual Machine/ Hypervisor:** virtualisiert eines ganzen Systems inkl. Prozessor, Speicher und IO -> Overhead
  - **Typ I Hypervisor** (Native System VM) läuft direkt auf der Hardware
  - **Typ II Hypervisor** (Hosted System VM) als Teil des Betriebssystems ausgeführt
- **Container:** virtualisiert nur den Userspace (-> ermöglicht Gruppierung und Ausführung mehrerer Prozesse in isolierten Umgebungen, wobei sie sich denselben Host-Betriebssystem-Kern teilen) -> geringer Overhead



Process Virtual Machines virtualisieren das **Application Binary Interface (ABI)**
System Virtual Machines virtualisieren die **Instruction Set Architecture (ISA)**

Bestandteile von Linux Container

- **Namespaces:** Bieten per Namespace eine isolierte Sicht auf globale Ressourcen
  - **User-Namespaces:** Isolieren Nutzer- u. Gruppen-IDs auf einem System, erlauben separate Vergaben von Rechten
    - Dateien, auf deren Nutzer nicht gemappt wird, gehören in User-Namespce **nobody**
  - **Mount-Namespaces:** Isolieren die Mountpoints im virtual file system (VFS) - verschiedene Sicht auf Verzeichnisstruktur des Dateisystems (Änderung im gemeinsam genutzten Dateisystem sind daher in allen darauf zugreifende Namespaces sichtbar -> völlig unabhängige Ansichten durch Partitionen/ temporäre Dateisystem tmpfs)
  - **PID-Namespaces:** Isolieren die Prozesse und PIDs
  - **Network-Namespaces:** Isolieren die Netzwerkstacks (Interfaces, Adressen, Firewall etc.)
  - **UTS-Namespaces:** (UTS = Unix Time Share) Isolieren Host- und Domainname des Hosts
- **Control Groups:** Systemressourcen werden für Prozessgruppen allokiert/ limitiert und überwacht
  - verwaltet CPU-Zeit, CPU-Kerne, Speicher, Prozess-Anzahl, IO-Zugriff
- **SECCOMP:** Ein auf den BPF basierter Mechanismus, um Systemcalls für Prozesse einzuschränken

Bestandteile von E/A-System-Struktur

- **User-Level-Software:** Bibliotheksroutinen für E/A, Spooling
- **Geräteunabhängige Software:** führt Berechtigungsprüfung aus, um zu ermitteln, ob der Nutzer das Gerät nutzen darf
- **Geräte-Treiber:** Übersetzen von geräteunabhängigen auf gerätespezifische Operationen; Interrupt Handling; Melden von Gerätefehlern; Initialisieren der Hardware, berechnet Track und Sektor bei einem Festplattenzugriff, beschreibt den Register des Geräts
- **Interrupt-Handler:** Geräte-Rückmeldungen behandeln
- **Geräte-Controller:** Bestandteil der Geräte-Hardware, Steuerung der beweglichen Hardware (z.B. Lesekopf), Interrupt melden, Daten puffern, Kommunikation mit CPU

Unterbrechungsbehandlung bei Beendigung einer E/A Aktivität

1. Gerät meldet über ein Signal die Beendigung einer E/A-Aktivität
2. Interrupt-Controller-Chip erkennt das Signal, informiert die CPU
3. BS unterbricht aktuellen Prozess, wechselt in Kernel-Modus
4. Führt die Unterbrechungsbehandlungsroutine aus
5. Handler signalisiert Beendigung der Ausführung der Unterbrechungsbehandlungsroutine

Hardware-Timer (in Hardware implementiert): generiert Interrupt wenn der Timer abgelaufen ist
- Die Anzahl von gleichzeitig gesetzten Hardtimern wird **durch Hardware limitiert**

Soft-Timer (in Software implementiert): reduzieren die Anzahl der durch Hardware-Timer verursachten Interrupts, indem sie vor dem Verlassen des Kernel-Modus überprüft und abgearbeitet werden, wodurch sie weniger Einfluss auf die Systemleistung haben

Virtualisierungstechnik

- **Linux Container:** nutzt Linux cgroups und namespace zur Abschottung
- **Hardware-assisted virtualization:** Benutzung von Intel VT-x, AMD-V etc.

- **CPU-Virtualisierung**
  - **Para Virtualization:** Dedizierte Gerätetreiber kommunizieren mit dem Hypervisor durch sog. Hypercalls
  - **Binary Translation:** Hypervisor interpretiert/ emuliert (Teile des) Binärcodes der Gast-VM in Software
    -> Hypervisor im **System/ Kernel Mode** ausgeführt vs Gast-Betriebssystem im **User Mode** ausgeführt
- **Speichervisualisirung -> für Adressübersetzung von GMs**
  - **Shadow Page Tables (SPT)**
  - **Second Level Address Translation (SLAT)**

zur Kommunikation/ Datenaustausch mit anderen Geräten

- **Programmed I/O:** CPU kopiert und wartet synchron darauf, dass das Gerät die jeweiligen Aktionen ausführt
- **Interrupt-driven I/O:** CPU initiiert das Lesen/Schreiben eines Datum, und das Gerät meldet asynchron, sobald die Aktion abgeschlossen wurde
- **Direct Memory Access (DMA):** CPU konfiguriert DMA-Controller, der interagiert mit dem Gerät, stößt den Transfer an; Abschluss signalisiert ein Interrupt (✅ Mit DMA kann ein Geräte-Controller asynchron im Hintergrund auf Speicher zugreifeN ✅ DMA entlastet die CPU)
- **Port-mapped I/O**
- **Memory-Mapped I/O**

Hexdump - z.B. 32-bit Architektur

```
00000000  cf 08 31 91 15 dd f3 31  c5 96 db 99 3d e6 e2 00
00000010  00 00 00 0f b9 0b f4 a0  10 00 00 00 01 d5 6c e0
00000020  2e 30 67 fb 3c 66 64 8d  65 99 10 85 7e 7b a8 8e
00000030  74 11 aa a7 00 00 00 10  9a 86 e8 44 98 b0 a4 78
00000040  3b 70 79 92 9f ef 7c 56  41 4d 1c 07 fe ab 6f b5
```

```c
&Adressierung; *Pointer-Deferenzierung
struct element {char character;};

int main() {
    struct element example = {'A'};
    struct element *ptr = &example;
    printf(example.number) // Direkter Zugriff
    printf(ptr->number)  // Indirekter Zugriff
}
```

Falls auf **Little-Endian** Architektur -> Inhalt von Adresse 0x04 auslesen: 0x31 f3 dd 15
Falls auf **Big-Endian** Architektur -> Inhalt von Adresse 0x04 auslesen: 0x15 dd f3 31
Für **char** und **string** braucht man **keine Umwandlung** in Endian - einfach von links nach rechts lesen
Sofern man ein **Null-Byte 00** lesen, ist der **String zu Ende gekommen**

Bei der **Pointerarithmetik** bestimmt der **Datentyp**, auf den der Pointer zeigt, die **Sprungweite**: Ein short* springt um 2 Bytes, ein int* um 4 Bytes, auch wenn **beide Pointer selbst auf einem 32-Bit-System 4 Bytes groß** sind

```c
// sizeof(str) = strlen(str) + 1 (Platz für terminierende null-Byte)
char str[] = "tux";
size_t len = strlen(str); // strlen(str) = 3  -> wie viele chars hat der String (ohne null-Byte)
size_t size = sizeof(str);// sizeof(str) = 4  -> wie viel Speicherplatz dafür allokiert sind (mit null-Byte)
```

scanf schreibt zusätzlich ein abschließendes NULL byte. char input[100] -> scanf ("%99s", input); wäre korrekt
memset(out, '0', len): füllt den `out`-gezeigte Speicherbereich, mit dem '0' über eine Länge von den Bytes
char buf[3]={0}; alle drei Elementen von Array buf werden mit 0 initialisiert
fgets(buf, 3, stdin); liest 2 Zeichen von stdin ins Array buf und fügt am Ende der Eingabe automatisch ein Nullzeichen ('\0') hinzu

```c
void fct(char *s, const char *t) {
   while(*s++ = *t++) ;
}
```

strcpy() = so lange von t nach s kopiert, bis t auf '\0' (Stringende erreicht) zeigt

```c
1  intVar = arrayXYZ[8];
2  intVar = *(arrayXYZ+8);
3  intVar = *(int *) ((void *) arrayXYZ+(8*sizeof(int)));
```

intVar = 8 [arrayXYZ]

keinen Unterschied zwischen den ~~drei~~ vier Anweisungen
Alle greifen auf das 9. Element des Arrays zu

double result[n]; liegt auf Stack der Funktion und ist nach dem return nicht mehr gültig
-> **double result = malloc(n*sizeof(double));** besser (malloc() ist kein Systemcall)
nach **malloc()** muss man dann prüfen ob es erfolgreich bzw. ob result != NULL
Erst wenn der Speicherbereich nicht mehr benutzt wird, sollte mittels **free()** freigegeben - sonst **Memory Leak**
Falls user after free oder double free -> **undefined behavior**

malloc(), calloc() reserviert auf **Heap**, alloca() reserviert auf **Stack**

- Allokationen auf dem Stack sind wesentlich performanter als auf Heap, da nur der Stack-Pointer nach oben verschoben werden muss, um neuen Speicher zu allozieren)

int (*(*foo) (void)) []; foo is a pointer to function with no arguments returning pointer to array of int

Padding

1. Padding wird hinzugefügt, sodass jedes Feld auf einer Adresse beginnt, die ein Vielfaches seines Alignments ist
2. Am Ende wird Padding eingefügt, bis Gesamtgröße der Struktur ebenfalls ein Vielfaches des größten Alignments

Schutzmaßnahmen gegen Buffer-Overflow Angriffe

- **Stack-Canary:** zufällig generiert, vor der Return-Adresse auf Stack platziert, wird bevor die Funktion return überprüft
- **Data Execution Prevention (DEP):** Der Stack wird vom Betriebssystem durch das NX-Bit als non-executable markiert und verhindert somit die Ausführung vom einem potenziell eingeschleusten Shellcode auf dem Stack
- **Address Space Layout Randomization (ASLR)** randomisiert die Adresse des Stacks und erschwert dem Angreifer diesen zu finden

Anwendungsfälle für **Batchsysteme**: rechenintensive Aufgaben, wissenschaftliche Simulation durchführen
- Render-Farm für Animationsfilme, Supercomputer, Datenbank-Backup, Finanztransaktion-Verarbeitung

Anwendungsfälle für **interaktive Systeme**: erfordert sofortige Reaktion auf Benutzereingaben
- Smartphones, Getränkeautomaten-Controller, interaktive Kiosksysteme und Videospiele, online-Chat

Anwendungsfälle für **Echtzeit-Systeme**: erfordert sofortige Reaktion auf Ereignisse in festgelegten Zeitraum
- Automobilindustrie, industrielle Automatisierung (Steuerung von Produktionslinien), medizinische Geräte (Herzschrittmacher)

Inhalt von Prozess Control Block (PCB): enthält alle für Kontextwechsel relevant Information u. Datenstruktur
- Prozesspriorität, Prozesszustand, Process ID (PID), Process Group ID (PGID), Parent Process ID (PPID), Stack Pointer (SP), Programm Counter (PC)

Inhalt von Block Group in ext Dateisystem: Padding, Superblock (speichert Anzahl und Größe der i-nodes und Blöcke), Block Group Descriptor Table, Block Allocation Bitmap, Inode Allocation Bitmap, Inodes, Datenblöcke (direkt/ indirekt adressiert)
- Festplattenpartitionen werden in Blockgruppen aufgeteilt

Systemcalls unter Unix für Dateisystem
open(), close(), read(), write(), chmod(), chown(), mkdir(), rmdir(), link(), rename()

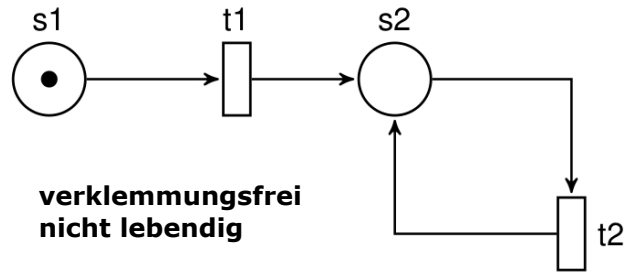
Systemcalls unter Unix für Prozess
fork(), exec(), wait(), exit(), kill(), getpid(), getppid(), nice(), setuid(), getgid()

Systemcalls unter Unix für Threads
pthread_create(), pthread_exit(), pthread_join(), pthread_mutex_lock(), pthread_cond_wait()

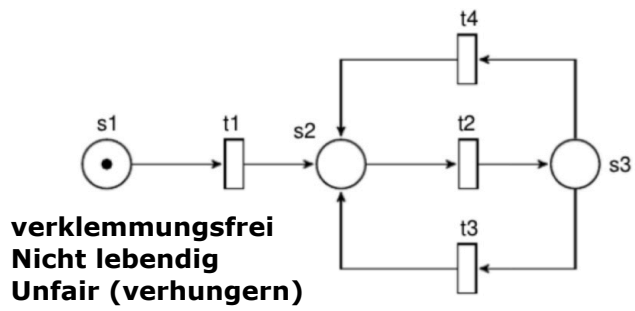
Systemcalls unter Unix für Kommunikation
pipe(), mkfifo(), **socket():** Öffnen eines neuen Sockets, **bind():** Binden des Sockets an einen Port, **listen():** Warten auf eine neue eingehende Verbindung, **accept():** Öffnen der eingehenden Verbindung als eigner File Descriptor, **connect():** Verbinden des Sockets zu einem Server, send(), recv(), **close():** Schließen der Datei
- **Server:** socket, bind, listen, accept, close (pro Client Socket), close (Server Socket)
- **Client:** socket, connect, close

SIGSEGV Segmentation Fault: schreibende Zugriff auf read-only Daten, Buffer Overflow, Stack Overflow, Heap Overflow, Zugriff auf nicht existierenden Speicher, von der Adresse 0 (wo für Null Pointer reserviert) lesen

Vom Programm (z.B. in C) zum ausführbaren Code
- **Compiler:** Übersetzer von C in Assembler
- **Assembler:** Übersetzer von Maschinensprache in die ISA
- **Linker:** bindet einzelne Module + liefert ausführbare Datei (Executable)
- **Loader:** Laden der gebundenen Objekte in den Speicher



RAID 0 (Striping) tragen nicht zur Erhöhung der Datensicherheit bei
RAID 1 (Mirroring)
RAID 0+1 (Striping+Mirroring)
RAID 2 nutzen Striping auf Bitebene
RAID 3 (Striping auf Bit-Ebene mit zusätzliche Platte für Paritätsinfo)
RAID 4 ((Striping von Blöcken mit zusätzliche Platte für Paritätsinfo)
RAID 5 (Striping von Blöcken und Verteilung der Paritätsblöcke)



Der Von-Neumann-Flaschenhals bezeichnet die zunehmende Diskrepanz der Geschwindigkeiten von Prozessor und Hauptspeicher. Dem Problem begegnet man üblicher Weise durch Einführung einer Speicherhierarchie

Swapping ermöglicht es, Arbeitsspeicher auf die Festplatte auszulagern, wenn der Speicherbedarf (=virtuelle Adressraum) die Kapazität des physischen Speichers übersteigt ✅ Speicherkapazität größer ❌ aber langsamer

cd / ins Wurzelverzeichnis (Root) wechseln
cd ./ ins aktuelle Verzeichnis wechseln
cd ../ ins übergeordnet Verzeichnis wechseln

```
int fd = open(argv[1], O-RDONLY)
size_t filesize = get_file_size(fd)
char *buffer = malloc(filesize)
read(fd, buffer, filesize)
write(fd, buffer, filesize)
close(fd)
free(buffer)
```

bekannte Dateisysteme
- **NTFS** (New Technology File System)
- **FAT32** (File Allocation Table 32)
- **exFAT** (Extended File Allocation Table)
- **EXT4** (Fourth Extended Filesystem)
- **HFS+** (Hierarchical File System Plus)
- **APFS** (Apple File System)

Gute Kryptographische Hashfunktionen

- Kryptographische Hashfunktionen sind **Einwegfunktionen**
- Die Ergebnisse kryptographischer Hashfunktionen sind **gleichverteilt** und **unabhängig vom Input**
- **Zwei Inputs** kryptographischer Hashfunktionen können **denselben Output** liefern
- Kryptographische Hashfunktionen sind **praktisch unumkehrbar**

durch Einsatz von Zertifikaten werden sichergestellt

- Das ein öffentlicher Schlüssel noch aktuell ist
- Das ein öffentlicher Schlüssel einer Instanz zugeordnet werden kann

wenn ein Prozess mit dem Warten auf ein **I/O-Event fertig**, wird dann **rechenwillig**

Clock und **LRU** erzeugen **wenige Pagefaults** bei **hoher zeitlicher Lokalität** von Seitenanfragen

Journaling schützt vor **Dateninkohärenzen** durch **Stromversorgung-Ausfall** und **Betriebssystem-Absturz**

TSL (Test-and-Set Lock) wird auf Hardware-Ebene zur Synchronisation von kritischen Abschnitten verwendet

SIGKILL und **SIGSTOP**: Die Signale können vom empfangenden Prozess nicht abgefangen oder blockiert werden

Wie kann das Dining Philosophers Problem von Dijkstra gelöst werden?

- Durch das Außerkraftsetzen einer der vier Deadlock-Bedingungen
- Durch das Einfügen der Möglichkeit, dass Philosophen anderen, welche nur ein Essstäbchen haben, dieses entziehen können
- Durch das Einfügen eines Tokens. Nur der Philosoph, der diesen Token aktuell besitzt, darf versuchen sich beide Essstäbchen zu nehmen, wenn diese verfügbar sind

Virtual File System (VFS)

- VFS stellt eine Schnittstelle zu den einzelnen auf der Maschine genutzten Dateisystemen
- Nutzer erkennen auf den ersten Blick keinen Unterschied zwischen auf der Maschine verwendeten Dateisysteme

Prozessadressraum in einem mit Paging verwalteten Speichersystem

- dient unter anderem dem Schutz unterschiedlicher Prozesse voreinander
- wird in Seiten eingeteilt

Lösu

c) Welche Aussagen treffen zu, wenn die Basisblockgröße bei dem Buddy-Allocator Algorithmus verdoppelt wird:

- [x] Bei gleichen Speicheranfragen bleibt der belegte Speicherplatz im Vergleich zu kleineren Basisblöcken gleich oder wird größer.
- [x] Es werden weniger Freibereichslisten benötigt.
- [x] Der Verwaltungsaufwand nimmt ab.
- [ ] Es werden gleich viele Freibereichslisten benötigt.
- [x] Der interne Verschnitt wird größer oder bleibt gleich.

a) Welche Prozesse können sich temporär blockieren?

Prozess A	Prozess B	Prozess C
1. fordert R an	1. fordert S an	1. fordert T an
2. fordert S an	2. fordert T an	2. fordert R an
3. gibt R frei	3. gibt S frei	3. gibt T frei
4. gibt S frei	4. gibt T frei	4. gibt R frei
[x] A und B	[x] A und C	[x] B und C