

Kapitel 1 - Virtualisierung des Prozessors:

Prozess: ein laufendes Programm, was nur im persistenten Speicher als Liste von Befehlen liegt und Betriebssystem überführt es in den laufenden Zustand

Gebrauch: - sollen viele Programme gleichzeitig laufen können und
- Anzahl der Prozesse soll unabhängig von Anzahl physischer Prozessoren sein
- Betriebssystem erzeugt die Illusion vieler existierender virtueller Prozessoren

Darstellung: - modelliert verschiedene Zustände der Maschine /- Was laufendes Programm lesen und verändern kann
- Welche Teile des Rechners wichtig für Ausführung sind (Register, Arbeitsspeicher, persistenter Speicher, etc.)

Prozesszustände: - **Running:** Prozess läuft derzeit auf der CPU. Befehle werden ausgeführt.

- **Ready:** Prozess kann ausgeführt werden, wird aber derzeit nicht ausgeführt.
Aus irgendeinem Grund hat das Betriebssystem den Prozess pausiert.

- **Blocked:** Prozess hat Operation durchgeführt, die ihn in blockierten Zustand gebracht hat bis bestimmtes Ereignis eintritt
->typisches Beispiel ist die Ein- und Ausgabe auf persistenten Speicher.

- **Scheduled:** Überführung eines Prozesses von Ready nach Running

- **Descheduled:** Überführung eines Prozesses von Running nach Ready

-> Nachdem Ereignis eingetreten ist, das den Blocked-Zustand löst, geht der Prozess immer erst in den Ready-Zustand

Timesharing: - Prozess wird eine Zeit lang ausgeführt und dann gestoppt, damit nächster Prozess Rechenzeit bekommt (usw)

-> Rechenzeit auf physischem Prozessor wird auf Prozesse verteilt, welche nacheinander laufen für gewisse Zeit
- erlaubt beliebig viele gleichzeitig laufende Prozesse (virtuell) -> erzeugt Eindruck der Parallelität (noch nicht da)

- **Nachteil:** je mehr Prozesse, desto langsamer

Vom Betriebssystem braucht man Mechanismen:

Zur Implementierung von CPU-Virtualisierung braucht es low-level Mechanik (wird durch Mechanismen erreicht) und smarte Strategien auf hohem Level. -> **Mechanismen:** beinhalten grundlegende Methoden und Protokolle, die notwendige Funktionalität liefern

ContextSwitch: notwendig. Mechanismus für Timesharing -> siehe weiter unten (direktes Ausführen)

ProgrammCounter(PC): Befehlszähler (auch Instruction Pointer, IP) zeigt auf den nächsten auszuführenden Befehl

StackPointer: dient zur Organisation von Informationen des Stacks, z.B. Funktionsparameter, lokale Variablen, Rückgabeadressen

Schnittstellen: - **Create** -> Prozess erstellen, z.B. nach Kommando in Konsole oder Klick auf Icon

- **Destroy** -> erzwungenes Beenden, z.B. wenn Prozess nicht selbständig endet

- **Wait** -> Prozess unterbrechen / anhalten, kann fortgesetzt werden

- **Misc. Control** -> weitere Kontrollmechanismen, z.B. zeitlich befristet zurückstellen

- **Status** -> Rückgabe von Informationen, z.B. Laufzeit oder aktueller Zustand

Datenstrukturen: - Betriebssystem hat verschiedene Datenstrukturen, um relevante Informationen über ablaufende Prozesse zu speichern und in Prozessliste werden alle Prozesse und deren Zustände vorgehalten

xv6 Prozess-Struktur aus OSTEP:

```
1 // the information xv6 tracks about each process including its register context and state
2 struct proc {
3     char *mem; // Start of process memory
4     uint sz; // Size of process memory
5     char *kstack; // Bottom of kernel stack for this process
6     enum proc_state state; // Process state
7     int pid; // Process ID
8     struct proc *parent; // Parent process
9     void *chan; // If !zero, sleeping on chan
10    int killed; // If !zero, has been killed
11    struct file *ofile[NOFILE]; // Open files
12    struct inode *cwd; // Current directory
13    struct context context; // Switch here to run process
14    struct trapframe *tf; // Trap frame for the current interrupt
15};
```

Prozesssteuerung -> Systemaufrufe als Schnittstelle zwischen Betriebssystem und Programmen

Befehle: - fork() -> neuer Prozess wird erstellt, welcher fast vollständige Kopie des aktuellen Prozesses darstellt

-> wird dann in child- und parent-Prozesse unterschieden

- wait() -> parent Prozess wartet auf erfolgreich beenden des child Prozesses

-> Prozess verschiebt die Ausführung seiner restlichen Befehle bis zur Rückgabe aus child

- exec() -> anderer Prozess (der keine inhaltliche Kopie des Originalprozesses ist) wird an bestimmter Stelle ausgeführt

- close() -> schließt file Deskriptor und die die standardmäßig offen sind

- open() -> öffnet Datei gemäß Parametern und nimmt ersten verfügbaren file descriptor

- kill() -> sendet Prozessoren verschiedene Signale (Beenden, Pausieren, ...)

-> Signale stellen eine Infrastruktur dar, um durch externe Ereignisse Einfluss auf Prozesse nehmen zu können.

Direktes Ausführen: - Programm läuft direkt auf dem Prozessor

- dazu wird Prozessoreintrag in Prozessorliste erstellt, Speicher zugelassen und Programmcode geladen

-> dann wird an Stelle gesprungen, um Ausführung zu starten

- D.A. ist Ausgangspunkt um Kontrolle über CPU zu behalten, während Prozess läuft

- Programme laufen direkt auf CPU (ist schnell und effizient) + Ein-/Ausgaben bzw. Zugriff auf (mehr)Speicher

Problem: sind Operationen, die nicht uneingeschränkt benutzt werden dürfen. Jeder Prozess darf nicht auf alles zugreifen.

Modi: - **User Mode:** stellt begrenzten Umfang an Operat. bereit + Ein-/Ausgaben führen zu Exceptions. BetrSys beendet Prozesse

- **Kernel Mode:** alle Operationen erlaubt (Bsp. Ein-/Ausgabe) -> Komponenten des Betriebssystems laufen in diesem Modus

Flexibler Rechtevergabe durch Verschlussmechanismen: Um Systemaufruf auszuführen, muss Programm speziellen Trap-Befehl

Ausführen -> dieser bewirkt Übergang von User Mode in Kernel Mode -> System führt dort gewünschte Operation aus (wenn erlaubt) und springt dann zurück zum Programm im User Mode

-> Beim Wechsel muss durch Hardware sichergestellt werden, dass irgendwann in den Ursprungszustand zurückgesprungen werden kann (Speicherinhalte, Registerzustände, etc.). -> Solche Informationen landen auf dem **Kernel Stack**

Trap-Tabelle: - wird angelegt beim Booten des Systems + beinhalten Infos (als Code) für Hardware darüber, was bei versch. besonderen Ereignissen passieren soll -> Zu diesen Ereignissen gehören neben Interrupts auch Systemaufrufe

Eingeschränkte direkte Ausführen (Trap-Table-Anlage): - hat 2 Phasen -> Bootzeit und Laufzeit

- Betriebssystem muss die Hardware über den Ort der Trap Handler informieren

- Hardware merkt sich diese Informationen bis zum nächsten Neustart des Systems

- Systemaufrufe werden üblicherweise über eine eindeutige Nummer identifiziert

- Programm-Code platziert benötigte Nummer im Register u. Betriebssystem prüft deren Gültigkeit (Sicherheitsmechanis.)

Prozesswechsel: **Situation:** Programme laufen direkt auf CPU, dann läuft Betriebssystem per Definition gerade nicht

Problem: gibt keine Möglichkeit eine Aktion auszuführen, wenn gerade keine Rechenzeit zur Verfügung steht

-> muss Wege geben die Kontrolle zurück zu erhalten, um Prozesse zu wechseln (2 Ansätze)

Ansätze: **Kooperativ(Abwarten):** - Betriebssystem vertraut darauf, dass laufende Prozesse sich vernünftig verhalten.

- wird angenommen, dass lange laufende Prozesse in regelmäßigen Abständen die Kontrolle zurückgeben

- inPraxis: viele Prozesse haben Systemaufrufe (I/O, ...) -> gibt yield()-Aufruf, um Kontrolle freiwillig zurückzugeben

- (Programm geben Kontrolle zurück, wenn sie etwas illegales tun (Trap zum Betriebssystem)

Imperativ(Kontrollübernahme): - Betriebssystem braucht Funktionen, um Prozesse unterbrechen zu können.

- typisches Bsp ist zeitbasierter Unterbrechungsmechanismus, der Prozesse nach gewisser Zeit anhält

Problem: Programm bleibt durch Fehler in in Unendlicher Schleife hängen -> Lsg: **Kontrollmechanismen** (Bsp.: Zeit)

-> **Zeit:** - Zeitgeber wird so programmiert, dass alle paar Millisekunden ein Interrupt ausgelöst wird

- Prozesse werden angehalten und der Interrupt Handler des Betriebssystems läuft

- spezieller Code für Hardware zur Boot-Zeit konfiguriert

-> Betriebssystem erhält Kontrolle nach gewisser Zeit zurück, wenn Timer gestartet wurde

-> aber es gibt privilegierte Operationen die Timer deaktivieren können

Anmerk: - Hardware hat bei Interrupt Verantwortung dafür, Zustand so zu speichern, dass Prozess später fortgesetzt werden kann

- Wenn Scheduler entscheidet, den Prozess zu wechseln, wird ein **Context Switch** durchgeführt

Context-Switch: Betriebssystem sichert zunächst Registerinhalte des aktuell laufenden Prozesses in Datenstruktur (Kernel Stack)

-> dann lädt es die entsprechenden Einträge für nächsten Prozess aus dieser Datenstruktur

-> so stellt Betriebssystem sicher, dass nach Rückgabe aus der Trap, der neue Prozess ausgeführt wird

SwitchSchritte: 1-Inhalte des allgemeinen Registers sichern 2-Programm Counter (Befehlszähler) sichern

3-Zeiger für Kernel Stack sichern 4-spezzielle **Switch()**-Methode aufrufen

Register: **User Register:** werden von Hardware mittels Kernel Stack gesichert -> Zeitpunkt Sichern: Timer Interrupt (Timer stop)

Kernel Register: werden von Software in Struktur für Prozesse gesichert -> Zeitpunkt: Betriebsys. entscheidet Wechsel

Strategien der Ablaufplanung (Scheduling Policies) -> Zeitmanagement von Prozessen

Annahmen und Vereinfachung: müssen den Aufwand von Prozessen quantifizieren können

Annahmen über Prozesse (Jobs): 1. Jeder Job läuft für genau die gleiche Zeitspanne. 2. Alle Jobs kommen zur gleichen Zeit an.

3. Wenn einmal gestartet, dann läuft Job bis zum Ende

4. Jeder Job nutzt nur Prozessortzeit (kein I/O). 5. Laufzeit jedes Jobs ist bekannt.

Gütermaß für Scheduling: TurnaroundTime: $T_{turnaround} = T_{completion} - T_{arrival}$ (TurnaroundT ist Metrik für Performance)

Bemerk: - derzeitige Annahme: Alle Jobs kommen zur gleichen Zeit an. -> $T_{arrival} = 0$

- Andere Metriken bemessen dagegen Fairness (Bsp.: Jain's Fairness Index)

- Performance und Fairness sind entgegengesetzte Ziele

FIFO (First in, First Out): -einfache Prinzip: Abarbeitung in Reihenfolge der Ankunft -> einfach zu implementieren (als Warteschlange)

Bsp1: Jobs A, B, C; Tarrival = 0 (nahezu); 10s je Job -> alle 3 Jobs sind gleich lang (10s) (Bild nicht dabei)

-> Durchschnittliche TurnaroundTime: $((10 + 20 + 30) - 0) / 3 = 20$

Bsp2 -> siehe oben rechte Seite

Konvoi-Effekt: kurze/schnelle Jobs müssen auf einen langen/langsamen Prozess warten

Beispiel 2: Jobs A, B, C; $T_{arrival} = 0$ (nahezu): Jobs unterschiedlich lang (ohne Annahme 1) -> \Rightarrow Turnaround Time: $((100 + 110 + 120) - 0) / 3 = 110$.

-> Bessere Strategie: **SJF(Shortest Job First):** Abarbeitung in Reihenfolge der Länge -> implement als Vorrangwarteschlange

-> geändertes Bsp2: geänderte Reihenfolge (B,C,A): Berechnung: -> \Rightarrow Turnaround Time: $((10 + 20 + 120) - 0) / 3 = 50$.

-> nach Annahme 2 ist Strategie beweisbar optimal

- Bsp3: $T_{arrival}(A) = 0$, $T_{arrival}(B) = T_{arrival}(C) = 10$ (ohne Annahme 2); Längen wie Bsp2 -> Reihenfolge(A,B,C)

-> \Rightarrow Turnaround Time: $((100 - 0) + (110 - 10) + (120 - 10)) / 3 = 103,33$ -> Annahme 3 wird verworfen in dem Fall

Geändertes Bsp3 durch STCF: selbe geg Daten -> \Rightarrow Turnaround Time: $((120 - 0) + (20 - 10) + (30 - 10)) / 3 = 50$.

-> hier muss Job nicht erst beenden, also: A arrived bei 0, dann B,C, dann A weiter

-> Optimal, wenn Jobs nicht zur gleichen Zeit ankommen

- **STCF(Shortest-Time-to-Completion First):** -auch: Preemptive Shortest Job First (PSJF)

-> hat Werkzeuge(TimerInterrupt, ContextSwitch), um Prozesse zu pausieren/wechseln

-> bei Ankunft neuer Jobs: Entscheidung für Job, der am schnellsten beendet werden kann

Ansprechverhalten: - bei stapelverarbeitenden Systemen war STCF valide Strategie.

- Bei Maschinen mit paralleler Benützung nicht valide.

-> man braucht **Response Time:** $T_{response} = T_{firstrun} - T_{arrival}$

-> T_firstrun bezieht Zeitpunkt, wann Job erste mal zum Ausführen vorgesehen ist

-> Annahme: Job erzeugt sofort eine Verzögerung eine Rückmeldung (Best Case)

Round Robin(RR): - Jobs laufen für eine gewisse Zeitspanne, statt immer bis zum Ende (Time Slicing, Zeitscheibe)

-> erzwingen Wechsel des Prozesses am Ende der Zeitspanne

- Prinzip wiederholt, bis alle Prozesse beendet -> Rundlaufverfahren genannt

- wichtig: Zeitscheibe bildet ein Vielfaches der Zeit des Timer Interrupts ab

Bemerk: - Länge der Zeitscheiben ist entscheidend für die Response Time (je kürzer desto besser) -> if aber zu kurz, dann anfallende Kosten für ContextSwitch zu groß -> man muss optimale Länge festlegen

Ein-/Ausgabe: - bei I/O: erste Entscheid, da Prozess Rechenzeit nicht benötigt /-zweite Entscheidung, wenn I/O beendet wurde

- automatischer Interrupt: Prozess wechselt von Blocked nach Ready

Beispiel 5a: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5b: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5c: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5d: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5e: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5f: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5g: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5h: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5i: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5j: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5k: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5l: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5m: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5n: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5o: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5p: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5q: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5r: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5s: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5t: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5u: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5v: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5w: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5x: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5y: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5z: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5aa: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ab: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ac: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ad: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ae: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5af: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ag: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ah: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ai: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5aj: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ak: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5al: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5am: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5an: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ao: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ap: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5aq: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ar: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5as: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5at: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5au: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5av: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5aw: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ax: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ay: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5az: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ba: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bb: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bc: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bd: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5be: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bf: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bg: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bh: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bi: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bj: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bk: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bl: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bm: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bn: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bo: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bp: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bq: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5br: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bs: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bt: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bu: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bv: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bw: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bx: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5by: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5bz: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ca: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5cb: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5cc: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5cd: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ce: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5cf: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5cg: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ch: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt

Beispiel 5ci: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt