

---

# Grundlagen und Konzepte der Betriebssysteme

Bert Randerath

Institut für Informatik  
Universität zu Köln

# Prozesse

---

- *Prozessverwaltung*
- *Interprozesskommunikation*
- *Scheduling*
  - CRASHKURS: KOMPLEXITÄT
- *Synchronisation nebenläufiger Prozesse*
  - PETRINETZE

# Prozess

---

Beim Multitasking-/Multiuser-Betrieb ist die Verteilung der Betriebsmittel auf die Programme von großer Bedeutung. Dabei muss zu jedem Programm vermerkt werden, welche Betriebsmittel wie Speicherplatz, CPU-Zeit, ... von ihm benötigt werden. Diese Informationen über benötigte Betriebsmittel für Programme werden zusammen mit dem Programm selbst auch **Prozess (task)** genannt.

*Prozess = Programme im Stadium der Ausführung*

Beim Scheduling (Belegungsplan) von Prozessen in Multiprozessorsystemen tauchen selbst in Spezialfällen (Nonpreemptive Scheduling) aus algorithmischer Sicht sehr schwere Probleme auf. Um dies zu zeigen benötigen wir einige Hilfsmittel aus der Komplexitätstheorie.

# CRASHKURS KOMPLEXITÄTSTHEORIE

---

- Die Klassen  $\mathcal{P}$  und  $\mathcal{NP}$
- Polynomielle Reduktion und NP-Vollständigkeit
- NP-Vollständigkeitsbeweise

# Prozessverwaltung

---

Prozesse (=Elternprozess) können andere Prozesse (=Kindprozesse) erzeugen. Programme (**Job**) in einem Multiprogramm-System können mehrere Prozesse (Multiprocessing) erzeugen.

**Z.B. Unix** Übergibt Interpreter (shell) das Kommando:

```
cat Text 1 Text 2 | pr | lpr
```

welches 3 Prozesse erzeugt:

1. Aneinanderhängen von *Text 1* und *Text 2*
2. Konvertieren von Textfiles in ausdrückfähige Files
3. Ausdrucken

Symbol “|” bewirkt, dass Ausgabe eines Prozesses zur Eingabe des nächsten wird.

Bei Einprozessorsystemen ist nur 1 Prozess aktiv, die anderen sind blockiert.

# Prozesszustände

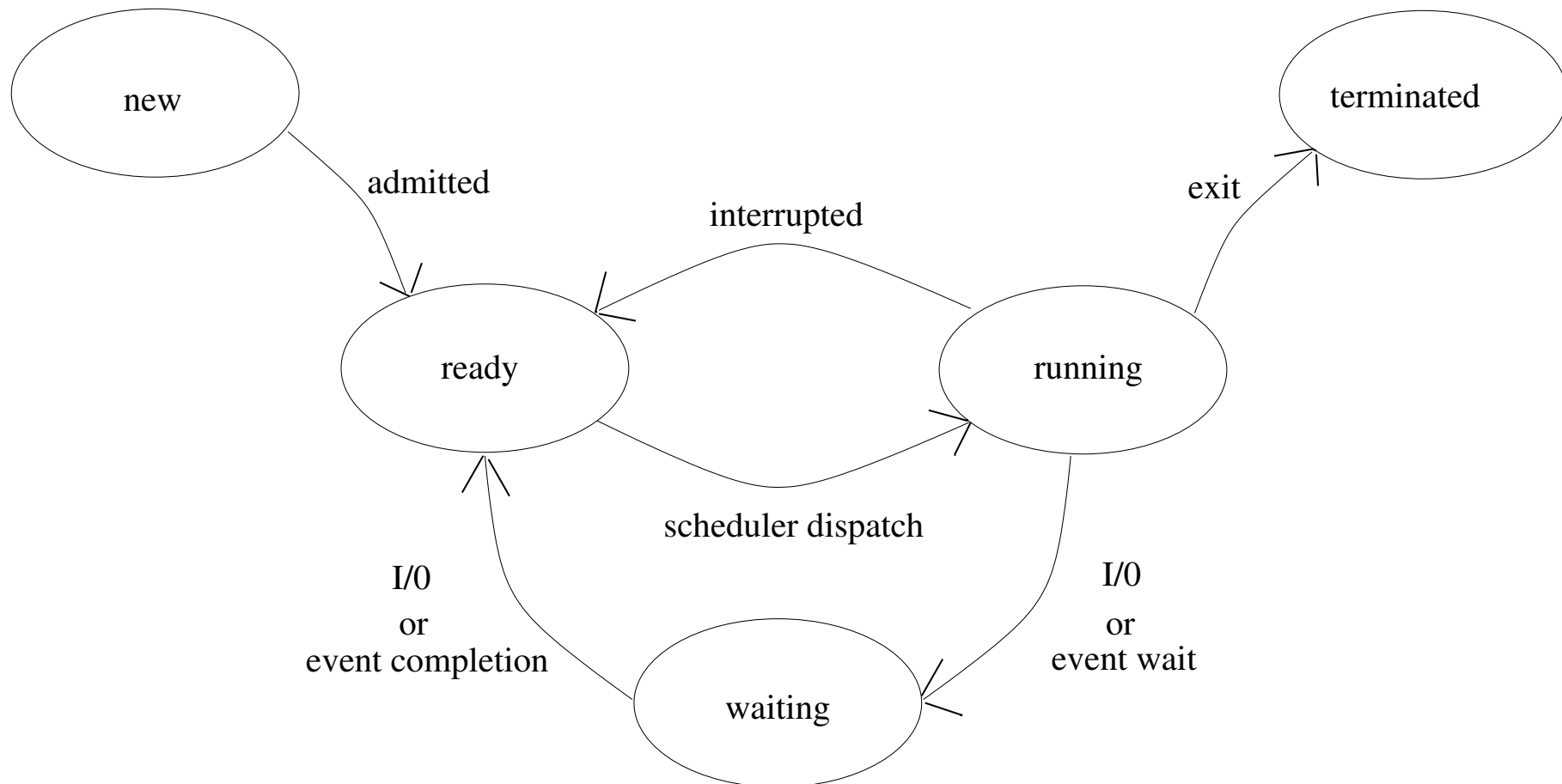
---

Bei gleichzeitigem Zugriff von zwei Prozessen auf das gleiche Betriebsmittel muss dieser Zugriff koordiniert werden. Dazu ordnet man den Prozessen Zustände zu, die die Koordination herstellen:

Prozesszustand	Bedeutung
running	Anweisung des Prozesses werden gerade ausgeführt
ready	Prozess zur Ausführung bereit, wartet noch auf freien Prozessor
waiting	Prozess wartet auf Eintreten eines Ereignisses (auf Freigabe eines von ihm belegten Betriebsmittels, das fertig wird)
blocked	Prozess wartet auf fremdbelegtes Betriebsmittel
new	Prozess wird erzeugt (kreiert)
killed	Prozess wird vorzeitig abgebrochen
terminated	Prozess beendet (Instruktionen bearbeitet)

# Zustandsdiagramm (Übergänge zwischen Zuständen)

Zu jedem Zeitpunkt kann auf jedem Prozessor nur ein Prozess laufen. Übrige Prozesse im *ready-and-stay-Status*.



# Prozessverwaltung

---

Bei Prozessen ist das Betriebssystem **primär zuständig** für

- Beendigung von Benutzer-/Systemprozessen
- Aufteilung von Speicherplatz und CPU-Zeit
- Bereitstellung von Mechanismen zur Synchronisation und Kommunikation unter Prozessen
- Behandlung von Deadlock-Situationen

## Prozesskontrollblöcke (PCB)

---

Repräsentation von Prozessen im Betriebssystem durch **Prozesskontrollblöcke** (PCB), die alle für das Betriebssystem wichtige Informationen über Prozesse enthalten: z.B. sind folg. Informationen über Prozesskontrollblöcke gespeichert:

PCB-Feld	Bedeutung
Status	enthält einen der Zustände aus o.g. Zustandsdiagramm
Programmzähler	enthält Adresse der nächsten auszuführenden Instruktion
Speichermanagment	enthält die letzten Werte der Speicherregister, Seitentabellen u.ä. Informationen für Speicherverwaltung
E/A-Status	Liste zugeordneter E/A-Geräte, offene Dateien u.s.w.
Accounting	Benötigte CPU-Zeit, Zeitbeschränkung, Prozessnummern, u.ä.
CPU-Scheduling	hier sind Prioritäten, Zeiger auf Warteschlangen u.a. Scheduling Parameter zu finden

# Interprozesskommunikation

---

Regelung der Kommunikation zwischen zwei Prozessen durch:

- Message passing
- Shared memory

## Message Passing

Über das Betriebssystem wird Verbindung zweier Prozesse *A* und *B* aufgebaut durch Systemaufrufe wie *gethostid*, *open\_connection*, *accept\_connection* oder *close\_connection*. Verbindungen zu zwei oder mehr Prozessen gehörig, uni- oder bidirektional möglich. Mit *send* und *receive* Nachrichtenaustausch über diese Verbindungen.

# Interprozesskommunikation

---

## Shared Memory

Durch Aufheben der exklusiven Nutzung von Speicherbereichen durch jeweils einen Prozess können über solche Speicherbereiche Nachrichten ausgetauscht werden: Prozess *A* speichert Nachrichten an Prozess *B* ohne Umweg über Betriebssystemkern, in Speicherbereich, auf den Prozess *B* Zugriff hat.

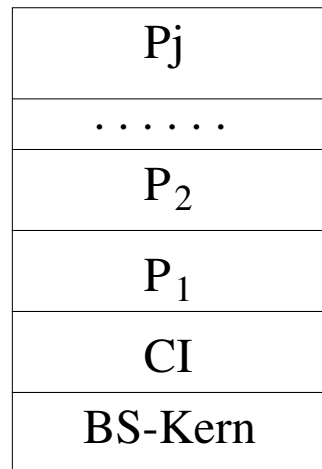
Vorteile von SHARED MEMORY gegenüber MESSAGE PASSING:  
Grosse Datenmengen können schneller übertragen werden.

Mögliche Probleme bei SHARED MEMORY: Synchronisations- und Konsistenzprobleme. *A* muss erst geschrieben haben, danach kann *B* lesen. (Organisation durch *mutual exclusion*)

# Scheduling

---

Beim **Singletasking** (Hauptspeicher enthält Betriebssystemkern und aufsetzend den Commando Interpreter CI) wird aufgerufener Prozess exkl. in Speicher geladen. Bei Multiprogramming und Multitasking neben Betriebssystemkern und CI mehrere Prozesse gleichzeitig im Speicher möglich.



Betriebssystem verwaltet Prozessoren gleichzeitig und regelt Zugriff auf CPU, dies geschieht durch **Scheduling**.

# Scheduling auf zwei Ebenen

---

- Langzeit-(Job)Scheduler = zuständig für Prozesse, die in Speicher geladen werden.
- Kurzzeit-(CPU)Scheduler = legt fest, welcher der geladenen Prozesse im *ready-Status* wann auf CPU zugreifen kann.

Gute Scheduler achten auf ausgewogenen Job-Mix zwischen E/A-intensiven und rechenintensiven Prozessen für gute Auslastung des Gesamtsystems. Bei Multitasking organisiert das Betriebssystem noch das *timesharing* meist durch *Round Robin Strategie*

# Prozess-Scheduling

---

Koordination des Zugriffs auf Betriebsmittel nötig, wenn Nachfrage das Angebot übersteigt. Scheduler koordinieren dann die Prozesse durch Einsortieren in Warteschlange, wodurch Zuweisung des/der Prozessor(s)/en an Prozesse geregelt wird.

Schedulingstrategien versuchen bestimmte **Ziele** zu realisieren:

- Möglichst hohe CPU-Auslastung (Wunsch : 100%, real 40%-90%)
- Hoher Durchsatz (throughput) (Möglichst hohe Zahl an Jobs pro ZE, Maß für hohe Systemauslastung)

# Ziele beim Prozess-Scheduling

---

- Fairness (Kein Job sollte anderen gegenüber ohne Sondervereinbarung bevorzugt werden!)
- Niedrige Ausführungszeit (turnaround time) (Zeitspanne von Jobbeginn bis Jobende; enthält Summe der Zeiten in Warteschlangen, der Ausführung (= **Bedienzeit**) und der E/A)
- Niedrige Wartezeit (waiting time) (Schedulerstrategie beschränkt sich auf *Minimierung* der Wartezeit in ready-Liste)
- Antwortzeit niedrig (response time) (Zeit zwischen interaktiver Eingabe und Ausgabe der Antworten auf Ausgabegerät sollte kurz sein!)

# Scheduling

---

Obige Ziele nicht konsistent: Z.B. Bevorzugung kurzer Prozesse beim Scheduling führt zu gutem Durchsatz und niedriger Antwortzeit, verletzt aber die Fairness.

~> Idealer Scheduling-Algorithmus existiert nicht.

~> Sinnvoll, in Scheduling-Algorithmus alle möglichen Schedulingstrategien zu integrieren, um je nach konkreter Situation verschiedene der o.g. Ziele stärker zu berücksichtigen.

# 1-Prozessorbetrieb

---

**Non-preemptive Scheduling (1-Prozessorbetrieb)** (running-Prozesse werden erst nach Abarbeitung beendet). Sinnvoll, wenn existierende Prozesse und ihre Anforderungen vollständig bekannt sind (z.B. bei DB-Zugriff)

## Scheduling-Strategien

- **First come first serve** (FCFS oder FIFO) Jobs werden bei Entstehen in Warteschlange eingefügt.
- **Shortest-Job-First Strategie** (SJF) Prozesse mit (geschätzt) kürzester Ausführungszeit werden zuerst bedient.
- **Highest response ratio next** (HRN) Bearbeitet Jobs mit maximalem Verhältnis  $\frac{\text{Antwortzeit}}{\text{Bedienzeit}}$  zuerst. (Zeiten liegen Schätzungen zugrunde).
- **Prioritätsscheduling** (PS) Nächster Job, der in ready-Liste aufgenommen wird, wird in Warteschlange gemäss seiner Prioritäten einsortiert.

## Non-Preemptive Scheduling-Strategie: First come first serve

---

Jobs werden bei Entstehen in Warteschlange eingefügt.

Z.B. gleichzeitig kommen 3 zu bearbeitende Jobs an mit Rechenzeit 1, 3, 9 ZE'en, wird erst Job mit Laufzeit 9, danach der mit 3 und dann der mit 1 abgearbeitet, so beträgt **mittlere turnaround-Zeit** dieser Reihenfolge

$$\frac{1}{3}(9 + 12 + 13) = \frac{34}{3} = 11\frac{1}{3} \text{ ZE'en}$$

Bei Reihenfolge 1, 3, 9 ist mittlere *turnaround-Zeit*

$$\frac{1}{3}(1 + 4 + 13) = 6$$

## Non-Preemptive Scheduling-Strategie: Shortest-Job-First

---

Prozesse mit (geschätzt) kürzester Ausführungszeit werden zuerst bedient. SJF BERECHNET REIHENFOLGE DER JOBABARBEITUNG MIT KLEINSTER MITTLERER TURNAROUND-ZEIT! SJF berechnet zu einer Folge von  $n$  Jobs mit Ausführungszeiten  $t_1, t_2, \dots, t_n$  eine Abarbeitungsreihenfolge mit minimaler *turnaround-Zeit*.

**Beweis:** Die mittlere Ausführungszeit bei Abarbeitung der Jobs in Reihenfolge  $t_{i_1}, t_{i_2}, \dots, t_{i_n}$ :

$$\frac{1}{n} \sum_{k=1}^n \sum_{j=1}^k t_{i_j} = \frac{1}{n} \sum_{k=1}^n (n+1-k) t_{i_k} \quad (*)$$

$\stackrel{!}{=} \min.$

## Non-Preemptive Scheduling-Strategie: Shortest-Job-First

---

Offensichtlich gilt, dass (\*) minimal ist, wenn

$$n \cdot t_{i_1} + (n - 1)t_{i_2} + \dots + 2t_{i_{n-1}} + t_{i_n} = \min. \quad (**)$$

(\*\*) ist offensichtlich minimal, wenn

$$t_{i_1} \leq t_{i_2} \leq \dots \leq t_{i_{n-1}} \leq t_{i_n}$$

gewählt werden, d.h. in der von SJF gewählten Reihenfolge.  $\square$

Nachteil der SJF Strategie: Fairnessgebot nicht beachtet. Bei kontinuierlich entstehenden neuen Jobs kann es zum **Verhungern** (starvation) eines Jobs mit hoher benötigter Rechenzeit kommen.

# Non-Preemptive Scheduling-Strategien

---

- **Highest response ratio next (HRN)**

Bearbeitet Jobs mit maximalem Verhältnis  $\frac{\text{Antwortzeit}}{\text{Bedienzeit}}$  zuerst. (Zeiten liegen Schätzungen zugrunde). HRN bevorzugt Jobs mit kurzer Bedienzeit. Mit zunehmender Wartezeit steigt aber auch Präferenz von Jobs mit langen Bedienzeiten.

- **Prioritätsscheduling (PS)**

Nächster Job, der in ready-Liste aufgenommen wird, wird in Warteschlange gemäss seiner Prioritäten einsortiert. Bei statischen Prioritäten können Prozesse wie bei SJF verhungern. Kann bei dynamischer Prioritätenzuweisung verhindert werden.

# Preemptive Scheduling

---

Nachteil von Non-Preemptive Scheduling: Jobs mit hoher Ausführungszeit verzögern wartende Jobs in erheblichem Maße.

Multitasking mit FIFO-Warteschlange (FCFS-Strategie) gesteuert durch jeweils gleich grosse Zeitscheiben ist als **Round-Robin-Algorithmus** bekannt.

## Scheduling-Strategien

- **Dynamic Priority Round Robin (DPRR)** Prioritäten der Prozesse in ready-Schlange wachsen nach jeder sie nicht berücksichtigenden Zeitscheibe.
- **Shortest Remaining Time First** Variante von SJF: bevorzugt Jobs mit kleinster restlicher Bedienzeit.

## (Non-preemptives) Scheduling in Multiprozessorsystemen

---

Vorbedingung: **Alle Prozesse unabhängig** (Keine Abhängigkeit zwischen Prozessen)  $m$  gleichartige Prozessoren  $P_1, \dots, P_m$ , auf denen  $n$  Jobs  $J_1, \dots, J_n$  mit Ausführungszeiten  $t_1, \dots, t_n$  abgearbeitet werden sollen.

Ein **Belegungsplan (Schedule)**  $S$  legt für jeden Job  $J_i$  den Prozessor und das Zeitintervall zur Abarbeitung fest.

Sei  $f_i$  Zeitpunkt, zu dem Abarbeitung von  $J_i$  unter  $S$  beendet ist. Dann beträgt die **mittlere Beendigungszeit** (mean finish time (MFT)) der Jobs unter  $S$ :

$$MFT(S) = \frac{1}{n} \sum_{i=1}^n f_i$$

## (Non-preemptives) Scheduling in Multiprozessorsystemen

---

Sei  $w_i > 0$  zu  $J_i$  gehöriges Gewicht,  $1 \leq i \leq n$ . Die mean weighted finish time (MWFT) unter  $S$  ist:

$$MWFT(S) = \frac{1}{n} \sum_{i=1}^n w_i f_i$$

Sei  $T_j$  Zeitpunkt, zu dem Prozessor  $P_j$  alle durch  $S$  zugewiesenen Jobs abgearbeitet hat. Dann beträgt die **Beendigungszeit** (makespan, finish time (FT)) von  $S$ :

$$FT(S) = \max_{1 \leq j \leq m} \{T_j\}$$

**Ziel:** Bestimme Schedule  $S$ , welche die jeweils gesuchte Zielfkt. FT(S) bzw. MWFT(S) minimiert!

# Non-Preemptives Scheduling in Multiprozessorsystemen

---

Betrachte nun einen Spezialfall (Entscheidungsproblemvariante) des Minimierungsproblems *Finde ein Schedule  $S$ , welches die Zielfunktion  $FT(S)$  minimiert*

MINIMUM FINISH TIME NONPREEMTIVE SCHEDULE BEI 2 PROZESSOREN

**2MFTNS:** Gegeben seien  $n$  Jobs  $J_1, \dots, J_n$  mit Ausführungszeiten  $t_1, \dots, t_n$ .

Können diese  $n$  Jobs auf 2 Prozessoren in Zeit  $\frac{1}{2} \sum_{i=1}^n t_i$  ausgeführt werden ?

**Satz:**  $2MFTNS \in NP_c$

**Beweis:**  $2MFTNS \in NP$

$PARTITION \leq_p 2MFTNS$  ■

Bemerkung: Das Optimierungsproblem ist somit  $NP$ -hart.

# Non-Preemptives Scheduling in Multiprozessorsystemen

---

## 2 Prozessoren Minimum Mean Weighted Finish Time (2MMWFT)

$n$  Jobs mit Ausführungszeiten  $t_i$  und Gewichten  $w_i$ , wobei

$$w_1 = t_1 = a_1, \dots, w_n = t_n = a_n.$$

**Behauptung:** Die  $n$  Jobs auf 2 Prozessoren mit mittlerer gewichteter Ausführungszeit von  $\leq \frac{1}{n} \left( \frac{1}{2} \sum_{i=1}^n a_i^2 + \frac{1}{4} \left( \sum_{i=1}^n a_i \right)^2 \right)$  ausführbar (2MWFT) genau dann, wenn  $\{a_1, \dots, a_n\} \in PARTITION$  ist.

# Non-Preemptives Scheduling in Multiprozessorsystemen

## Beweis der Behauptung:

Seien  $(w'_1 = t'_1, \dots, w'_k = t'_k)$  Gewichte und Ausführungszeiten der Jobs auf Prozessor 1 und  $(w''_1 = t''_1, \dots, w''_l = t''_l)$  analog für Jobs von Prozessor 2. Ausführung der Jobs in dieser Reihenfolge jeweils. Für diesen Schedule  $S$  gilt:

$$\begin{aligned}
 n \times MWFT(S) &= \overbrace{w'_1 t'_1 + w'_2(t'_1 + t'_2) + \dots + w'_k(t'_1 + \dots + t'_k) + w''_1 t''_1 + \dots + w''_l(t''_1 + \dots + t''_l)}^{\sigma} = \\
 &= \frac{1}{2} \sum_{i=1}^n w_i^2 + \frac{1}{2} \left( \sum_{i=1}^k w'_i \right)^2 + \frac{1}{2} \left( \sum_{i=1}^n w_i - \sum_{i=1}^k w'_i \right)^2 \quad (**) \\
 &\quad \underbrace{\hspace{15em}}_{\frac{1}{2} \left( \sum_{i=1}^l w''_i \right)^2}
 \end{aligned}$$

# Non-Preemptives Scheduling in Multiprozessorsystemen

---

## Beweis der Behauptung:

Falls  $\{a_1, \dots, a_n\} \in PARTITION$  ist, so sind in (\*\*) die 2. und 3. Summe gleich.

$$\rightsquigarrow n \times MWFT(S) = \frac{1}{2} \sum_{i=1}^n w_i^2 + \frac{1}{4} \left( \sum_{i=1}^n w_i \right)^2 \quad (***)$$

Dies ist minimal möglicher Wert von (\*\*).

$PARTITION \leq_p 2MWFT$  mit Schranke (\*\*\*)

# Non-Preemptives Scheduling in Multiprozessorsystemen

---

**Satz:**  $2MWFT \in NP_C$

**Beweis:**  $2MWFT \in NP$

$PARTITION \leq_p 2MWFT$  ■

Bemerkung: Das entsprechende Optimierungsproblem, d.h. die Bestimmung eines minimalen MWFT-Schedule ist somit  $NP$ -hart.

## Ein pseudopolynomieller Algorithmus für das *SUBSETSUM*-Problem

---

Input:  $A = \{a_1, \dots, a_n\} \subseteq \mathbb{N}$ ,  $M \in \mathbb{N}$

Frage:  $\exists A' \subseteq A : \sum_{a \in A'} a = M$  ?

Lösungsansatz mit Dynamischer Programmierung:

Fülle  $n \times (M + 1)$  Matrix  $T(i, j)$ ,  $1 \leq i \leq n$ ,  $0 \leq j \leq M$  mit *Booleschen Werten*.

$$T(i, j) = \begin{cases} 1, & \text{falls } A_i = \{a_1, \dots, a_i\} \subseteq A \text{ und } A'_i \subseteq A_i \text{ ex. mit } \sum_{a \in A'_i} a = j \\ 0, & \text{sonst} \end{cases}$$

$\rightsquigarrow T(n, M) = 1 \Leftrightarrow$  Für  $A, M$  läßt sich obige Frage mit JA beantworten.

## Ein pseudopolynomieller Algorithmus für das *SUBSETSUM*-Problem

---

Berechne die  $T(i, j)$  wie folgt:

$$\triangle \begin{cases} T(i, j) = 1 \Leftrightarrow j = 0 \vee j = a_1 \\ T(i + 1, j) = 1 \Leftrightarrow T(i, j) = 1 \vee (1 \leq i \leq n - 1 \wedge j - a_{i+1} \geq 0 \wedge T(i, j - a_{i+1}) = 1) \end{cases}$$

Berechnung der Matrix  $T$  in Zeit  $O(n \cdot M)$ .

$\rightsquigarrow T(n, M)$  enthält Lösung der *SUBSETSUM* – Instanz.

Widerspruch zu  $SUBSETSUM \in NPC$ ?

Nein, denn Laufzeit =  $(n \cdot M)$  nicht (immer) polyn. in Inputlänge von  $A, M$ .

## Ein pseudopolynomieller Algorithmus für das *SUBSETSUM*-Problem

---

$$\text{Länge } (A, M) = \sum_{i=1}^n (\lfloor \log_2(a_i) \rfloor + 1) + \lfloor \log(M) \rfloor + 1 = l$$

$\rightsquigarrow f(l) = n \cdot M$  kann schneller als jedes Polynom wachsen, da für  $n = \log(M)$  beispielsweise  $l \leq \log^2(M)$  gilt:

$$\begin{aligned} f(l) = f(\log^2(M)) &= \log(M) \cdot M \\ &= \sqrt{\log^2(M)} \cdot 2^{\sqrt{\log^2(M)}} \\ &\geq \sqrt{l} \cdot 2^{\sqrt{l}} \quad \text{wächst schneller als jedes Polynom in } l \end{aligned}$$

- Dynamischer Programmieransatz  $\triangle$  benötigt jedoch polynomielle Zeit, für *kleine* Zahlen  $M$  (d.h.  $M$  polynomial in  $n$  beschränkt)
- Verfahren speicherplatzaufwendig

## Online-Verfahren für $k$ MFTNS mit Fehlerbegrenzung 2

---

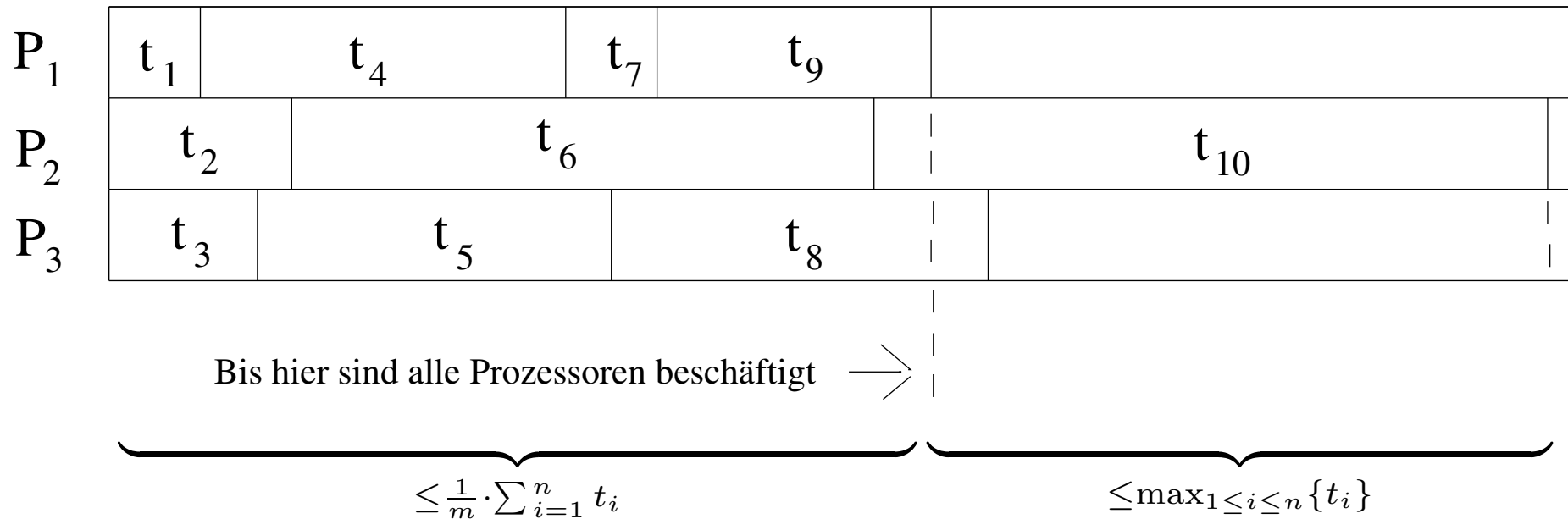
$n$  Jobs mit Ausführungszeiten  $t_1, \dots, t_n$  sollen  $m$  Prozessoren zur vollständigen Abarbeitung zugeordnet werden. Die Jobs entstehen so schnell, dass keiner der Prozessoren vorher *idle* (arbeitslos) wird.

**Strategie:** Ordne den nächsten ankommenden Job  $J_i$  dem Prozessor  $P_k$  zu, dessen bisherige Belegung ihn am wenigsten lange auslastet.

Sei  $FT_{opt}$  *makespan* eines optimalen Schedules der  $n$  Jobs und  $FT_{approx}$  *makespan* der Wert eines Schedules, welches durch obige Strategie erzielt wird.

## Online-Verfahren für $k$ MFTNS mit Fehlerbegrenzung 2

$t_1, t_2, \dots, t_{10}$  seien 10 Jobs für  $m = 3$  Prozessoren.



Offensichtlich gilt:  $FT_{opt} \geq \max\left\{\frac{1}{m} \sum_{i=1}^n t_i, \max_{1 \leq i \leq n} \{t_i\}\right\}$

$\Rightarrow FT_{approx} \leq \frac{1}{m} \sum_{i=1}^n t_i + \max_{1 \leq i \leq n} \{t_i\} \leq 2 \cdot FT_{opt}$

## Multiprozessorscheduling bei Präzedenzen zwischen den Prozessen

---

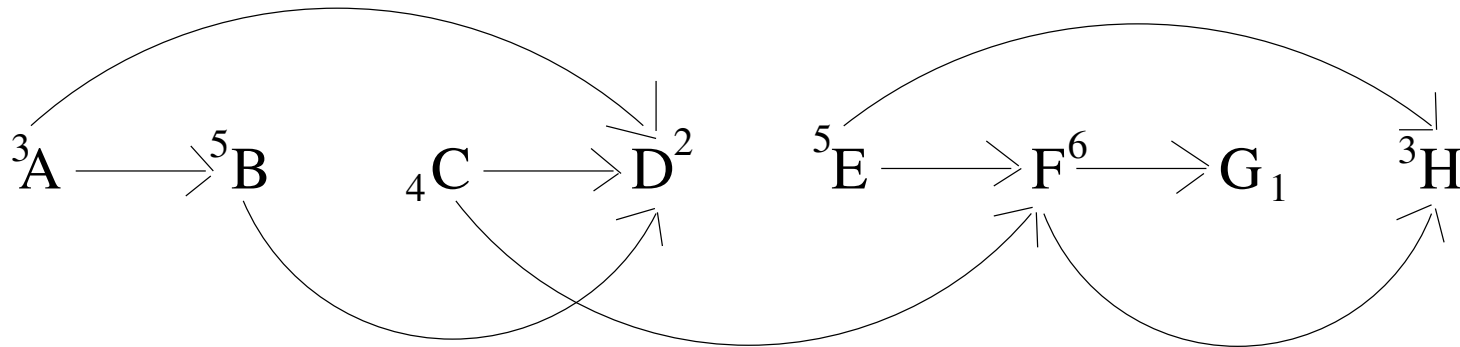
Häufig gibt es zwischen Prozessen Abhängigkeiten, wobei ein Prozess  $B$  erst starten kann, wenn ein anderer Prozess  $A$  fertig ist, ausgedrückt durch eine Kante  $A \rightarrow B$ . Derartige Vorrangrelationen (*Präzedenzen*) werden verursacht durch eine Betriebsmittelanforderung von  $B$ , wobei  $A$  zuvor das Betriebsmittel (BM) benötigt. Oft ist Anforderungszeit von  $A$  an das Betriebsmittel, um das  $A$  und  $B$  hier konkurrieren, bekannt, ausgedrückt durch Belegung der Kante mit diesem Wert.

$$\text{z.B. : } A \xrightarrow{t} B \quad , t \in \mathbb{N}$$

$B$  muß  $t$  Zeiteinheiten lang auf das angeforderte BM warten, wenn es  $A$  zugewiesen wird. Solche Abhängigkeiten sind durch kantenbewertete *Präzedenzdigraphen* (PD) darstellbar. Da Prozess  $A$  über die Zeitdauer seiner Rechnung alle von ihm benötigten Betriebsmittel festhält, haben in der Regel alle von  $A$  ausgehenden Kanten den selben Wert.

## Beispiel: Präzedenzdigraph

---



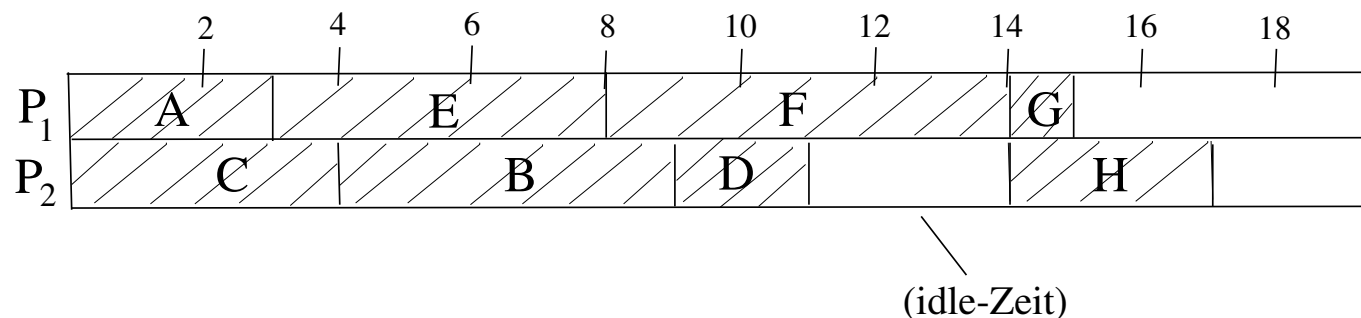
**Bemerkung:** Ein PD muss ein Directed Acyclic Graph (DAG) sein, weil Prozesse in einem Kreis sich gegenseitig beim Weiterrechnen blockieren (**Deadlock-Situation**)

Makespan eines bel. Schedule für Prozess-PD (PPD) kann nicht kürzer sein als ein längster Weg im PPD (inkl. Wert an letztem Knoten). Dieser heißt auch **kritischer** Weg. Im o.g. Beispiel ist längster Weg  $E^5 \rightarrow F^6 \rightarrow H^3$  mit Kosten 14.

# Scheduling Heuristik (Earliest Scheduling)

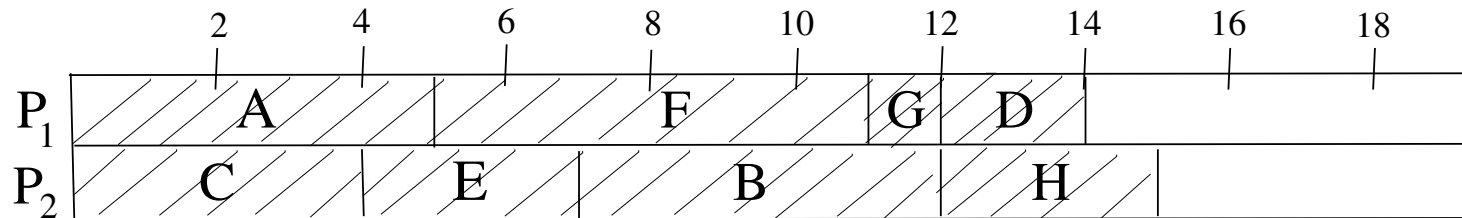
- Sortiere PPD topologisch
- Zerlege Knoten-(Prozess-)Menge in Klassen wie folgt: Fasse alle Knoten ohne einlaufende Kante zu einer Klasse zusammen und entferne sie aus PPD. Fasse im entstandenen PPD wieder alle die Knoten zu einer Klasse zusammen und entferne auch diese Knoten aus PPD. U.s.w. bis alle Knoten Klassen zugeordnet worden sind.
- Ordne dann Klasse für Klasse die Knoten den Prozessoren zu  
Klasse 1:  $A, C, E$  Klasse 2 :  $B, F$  Klasse 3 :  $D, G, H$

↪ 2 Prozessor Schedule  $S$  mit makespan 17:



## Scheduling Heuristik (Earliest Scheduling)

Anderer 2-Prozessor Schedule  $\hat{S}$  mit makespan 15:



$\hat{S}$  ist optimal, da

$$\begin{aligned} FT_{opt} &\geq \max_1 \left\{ \max_i \{t_i\}, \left\lceil \frac{1}{2} \sum_{i=1}^n t_i \right\rceil \right\} \\ &= \max \left\{ 6, \left\lceil \frac{1}{2} \cdot 29 \right\rceil \right\} = 15 \end{aligned}$$

Heuristik führt oft zu brauchbaren aber nicht notwendig optimalen Ergebnissen, wie das Beispiel zeigt.

## Multiprozessorscheduling bei Präzedenzen zwischen den Prozessen

---

Nonpreemptive Multiprozessor Scheduling mit Präzedenzen zwischen den Prozessen hat folgende Eigenschaften:

- $FT_{opt} \geq \max \left\{ \frac{1}{m} \sum_i t_i, \max_i \{t_i\}, \lambda_{krit} \right\}$ , wobei  $\lambda_{krit}$  Kosten eines kritischen (längsten) Weges im *PPD* sind.
- Sei  $k$  die Zahl der Klassen, in die Schritt 2 der *Earliest-Heuristik* den *PPD* zerlegt. Dann liefert das Verfahren immer ein Scheduling  $S_{approx}$  mit  $FT(S_{approx}) \leq 2k \cdot FT_{opt}$ .
- (Chudak/Shmoys 1997): Es gibt schnell berechenbares Approx-Scheduling.  $S_{app}$  mit  $FT(S_{app}) \leq O(\log(m)) \cdot FT_{opt}$

# Preemptives Multiprozessor Scheduling mit Präzedenzen

---

Die betrachteten Problemstellungen beim Nonpreemptiven Scheduling für Multiprozessorsysteme mit Präzedenzen bleiben auch beim *Preemptive*-Fall schwierig. Hier wird oft ein List-Scheduling vorgenommen:

Alle Prozesse befinden sich in zentraler Liste und haben Prioritäten. Ein freigewordener Prozessor entnimmt der Liste einen Prozess mit höchster Priorität, dessen Vorgänger im *PPD* bereits abgearbeitet sind. Hier werden typisch *ready-Liste* und *waiting-Warteschlange* verwendet.

Bei Prozessorbelegung wird oft *Round-Robin-Zeitscheibenverfahren* benutzt, modifiziert durch Prioritäten, die sich mit der Länge der Wartezeiten erhöhen. Prioritäten aus Bereich 0..255, wobei Prozesse mit gleicher Priorität in einer *FIFO-Warteschlange* einsortiert sind. Durch Änderung der Prioritäten wandern Prozesse von einer Warteschlange in eine andere. Man spricht auch vom *Multilevel Feedback Scheduling* (UNIX).