Scheduling

Gliederung

Scheduling

Übersicht:

- Was ist Scheduling ?
- Kooperatives / präemptives Scheduling
- CPU- und I/O-lastige Prozesse
- Ziele (abhängig vom BS-Typ)
- Standard-Verfahren
- Praxis: Kommandos zum beeinflussen des Scheduling

Was ist Scheduling ?

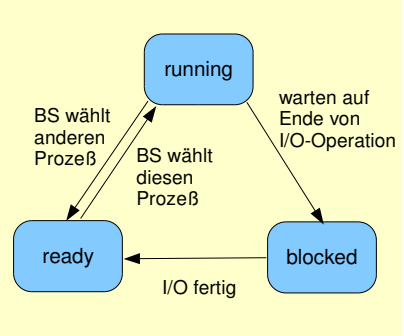
Was versteht man unter 'Scheduling' ?

- **Multitasking:** Mehrere Prozesse/Threads konkurrieren um ein Betriebsmittel
- das BS verwaltet die Betriebsmittel, z.B.
  - Rechenzeit auf dem Prozessor
    (folgende Beispiele beziehen sich auf's CPU-Scheduling)
  - I/O-Zugriffe auf Peripheriegeräte
- Scheduler entscheidet:
  - welchen Prozess wann ausführen ?
  - **Scheduling**: Zuteilung der CPU (Betriebsmittel) an Threads/Prozesse
- Ausführreihenfolge entschiedend für
  - Gesamt-Performance des BS
  - Performance individueller Prozesse

Prozess/Thread auswählen

Zustandsübergänge:

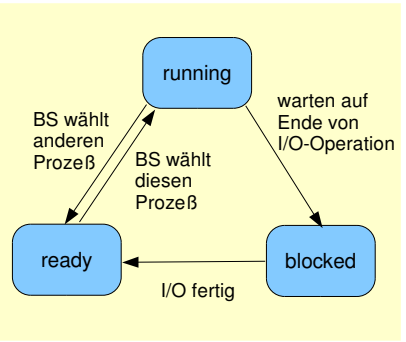
- der Scheduler wählt aus:



Prozess/Thread auswählen

Zustandsübergänge:

- der Scheduler wählt aus:
- wann wird Scheduler aktiv ?
  - neuer Prozess entsteht (fork)
  - aktiver Prozess endet (exit)
  - aktiver Prozess blockiert (z.B. wegen I/O)
  - blockierter Prozess wird bereit
  - Prozess rechnet schon zu lange
  - Interrupt tritt auf



Wie wird Scheduler aktiviert ?

- **Implementierung des Schedulers:**
  - meist als Interrupt-Handler
  - mit relativ niedriger Interrupt-Priorität
- **Aufruf des Schedulers:**
  - durch Auslösen dieses Interrupts, z.B.
    - durch Timer, der regelmäßig prüft, ob Quantum des laufenden Prozesses verbraucht ist
    - durch blockierenden System-Call (vgl. Folie davor)

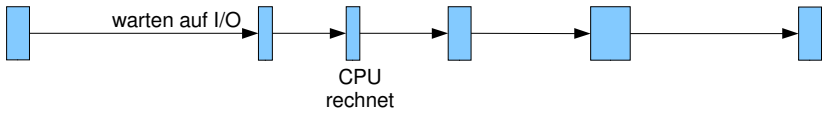
Scheduling-Prinzipien

Prozessunterbrechung möglich ?

- **Kooperatives Scheduling:**
  - Prozess rechnet solange wie er will
  - bis zum nächsten I/O-Aufruf oder exit()-Aufruf (alter Mac:...)
- **Präemptives (unterbrechendes) Scheduling:**
  - Timer aktiviert regelmäßig Scheduler, der neu entscheiden kann „wo es weiter geht“

Prozesse: I/O- oder CPU-lastig ?

- **I/O-lastig:**
  - Prozess hat zwischen längeren I/O-Phasen nur kurze Berechnungsphasen (CPU)



- **CPU-lastig:**
  - Prozess hat zwischen kurzen I/O-Phasen lange Berechnungsphasen

Häufige Prozesswechsel ?

Faktoren:

- **Wartezeit der Prozesse:**
  - Häufigere Wechsel

    --> stärkerer Eindruck von Gleichzeitigkeit

aber:

- **Zeit für Kontext-Wechsel:**
  - Scheduler benötigt Zeit, um Prozesszustand zu sichern

    --> verlorene Rechenzeit

Ziele / Kriterien

Aus Anwendersicht:

- **[A1] Ausführungsdauer:**
  - wie lange läuft der Prozess insgesamt ?
- **[A2] Reaktionszeit:**
  - wie schell Reaktion auf Benutzerinteraktionen ?
- **[A3] Deadlines:**
  - sind einzuhalten
- **[A4] Vorhersehbarkeit:**
  - gleichartige Prozesse sollten sich gleichartig verhalten
- **[A5] Proportionalität:**
  - „Einfaches“ geht schnell

Aus Systemsicht:

- **[S1] Durchsatz:**
  - Anzahl der Prozesse, die pro Zeit fertig werden ?
- **[S2] Prozessorauslastung:**
  - Zeit (in %), die der Prozessor aktiv ist
- **[S3] Fairness:**
  - Prozesse gleich behandeln, keiner darf verhungern
- **[S4] Prioritäten:**
  - zu beachten
- **[S5] Ressourcen**
  - gleichmäßig einsetzen

[A1] Ausführungsdauer

Wieviel Zeit vergeht vom Programmstart bis zu seinem Ende?

- n Prozesse p_1 bis p_n starten zum Zeitpunkt t_0 und sind zu den Zeiten t_1 bis t_n fertig
- Kriterium:
  durchschnittliche Ausführungsdauer:

$$1/n \sum_i^n t_i - t_0$$

- abhängig von konkreten Prozessen;

  Berechnung nur zum Vergleich verschiedener Scheduling-Verfahren sinnvoll

[A2] Reaktionszeit

Wie schnell reagiert das System auf Benutzereingaben ?

- warten nach Tastendruck / Mausklick
- Kriterium: Reaktionszeit:
  - Dauer zwischen Auslösen des Interrupts und Aktivierung des Prozesses, der die Eingabe auswertet
- Toleranz gering:
  - schon 100-200 ms störend bemerkbar !

[A3] Deadlines

Hält das System Deadlines ein ?

- Echtzeitsysteme: besondere Ansprüche
- Aufgaben sind in vorgegebener Zeit zu erledigen

  --> Prozessen ausreichend und rechtzeitig Rechenzeit zuzuteilen
- Kriterium:
  Wie oft werden Deadlines nicht eingehalten?
- Optimiere prozentualen Anteil der eingehaltenen Deadlines

[A4] Vorhersehbarkeit

Ähnliches Verhalten ähnlicher Prozesse ?

- Intuitiv: gleichartige Prozesse sollten sich gleichartig verhalten:
  - Ausführdauer und Reaktionszeit immer ähnlich
  - unabhängig vom sonstigen Zustand des Systems
- Schwierig, wenn das System beliebig viele Prozesse zulässt

  --> Beschränkungen ?

[A5] Proportionalität

Vorgänge, die „einfach“ sind, werden schell erledigt

- es geht um das (evtl. falsche) Bild, das sich Anwender von technischen Abläufen machen
- Anwender akzeptiert Wartezeit eher, wenn er den Vorgang als komplex einschätzt

[S1] Durchsatz

Es soll möglichst viel „Arbeit“ erledigt werden

- Anzahl der Prozesse (Jobs), die pro Zeit fertig werden, sollte hoch sein
- misst, wieviel „Arbeit“ erledigt wird
- Kriterium:
  Zahl erledigter Prozesse/Aufgaben pro Zeit
- abhängig von konkreten Prozessen;

  Berechnung nur zum Vergleich verschiedener Scheduling-Verfahren sinnvoll

[S2] Prozessorauslastung

CPUs immer gut beschäftigt halten

- Anteil der Taktzyklen, in denen die CPUs nicht 'idle' waren
- Interessantes Maß, wenn Rechenzeit sehr teuer ist,

  z.B. in kommerziellem Rechenzentrum
- hängt mit „Durchsatz“-Kriterium zusammen

[S3] Fairness

Alle Prozesse haben gleiche Chancen

- jeder Prozess sollte mal drankommen

  --> kein Verhungern (*starvation*)
- keine großen Abweichungen bei den Wartezeiten und Ausführungsdauern
- falls Prozess-Prioritäten:

  --> „manche sind gleicher“, also gleiche Behandlung bei entsprechenden Prioritäten

[S4] Prioritäten

Verschieden wichtige Prozesse auch verschieden behandeln

- Prioritätsklassen: Prozesse mit höherer Priorität bevorzugt behandeln
- Dabei verhindern, dass nur noch Prozesse mit hoher Priorität laufen
  (und alles andere steht)

[S5] Ressourcen-Balance

„BS verwaltet die Betriebsmittel...“

- Grundlage des BS: alle Ressourcen
  - gleichmäßig verteilen und
  - gut auslasten
- CPU-Scheduler hat auch Einfluss auf (un)gleichmäßige Auslastung der I/O-Geräte
- Prozesse bevorzugen, die wenig ausgelastete Ressourcen nutzen wollen

  Warum ?

Anforderungen an das BS

Drei Kategorien:

Stapelverarbeitung (Batch-Betrieb):

immer wichtig

- S1 Durchsatz
- A1 Ausführungsdauer
- S2 Prozessor-Auslastung

Interaktives System:

- S3 Fairness
- S4 Prioritäten
- S5 Ressource-Balance

- A2 Reaktionszeit
- A5 Proportionalität

Echtzeitsystem:

immer wichtig

- A3 Deadlines
- A4 Vorhersehbarkeit

Stapelverarbeitung (Batch-Betrieb)

Eigenschaften

- Nicht-interaktives System
  - keine normalen Anwenderprozesse, keine GUI
- Jobs:
  - werden über Job-Verwaltung abgesetzt
  - System informiert über Fertigstellung
- typische Aufgaben:
  - lange Berechnungsvorgänge
  - Vorgänge mit hohem Speicherbedarf
  - Cluster-Anwendungen

  --> Rechenzentrumsbetrieb

Moderne Batch-Systeme

- normale Rechner, meist Cluster

  z.B. RUS: IBM-Cluster
- Job-Management-Tool nimmt Jobs an
- Long-Term-Scheduler entscheidet,
  - wann Jobs gestartet werden
  - evtl. auf Basis von Informationen über zu erwartenden
    - Ressourcenbedarf
    - Laufzeit des Programms

    --> über explizite Angaben oder Statistiken

Interaktive Systeme

Eigenschaften

- Typisch:
  - Interaktive Prozesse und
  - Hintergrundprozesse
- Desktop- und Server-PCs
- Eventuell mehrere / zahlreiche Anwender, welche sich Rechenkapazität teilen
- Scheduler für interaktive Systeme prinzipiell auch für Batch-Systeme brauchbar (aber nicht umgekehrt)

  Warum ???

Scheduling-Verfahren für ...

... Batch-Systeme:

- **FCFS:**
  First Come, First Served
- **SJF:**
  Shortest Job First
- **SRT:**
  Shortest Remaining Time First
- **Prio-basiert:**
  Prioritäts-basiertes Scheduling

... Interaktive-Systeme:

- **Round-Robin**
  Zeitscheiben-Verfahren
- **Prio-basiert:**
  Prioritäts-basiertes Scheduling
- **(Lotterie-Scheduler)**

Scheduling-Verfahren

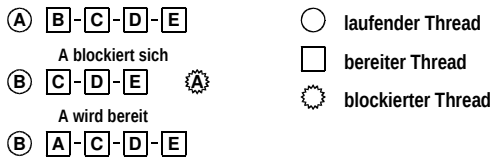
für

Batch-Betrieb

First Come, First Served (FCFS)

Arbeitsweise:

- nach Erzeugungszeitpunkt geordnete Warteschlange von bereiten Threads/Prozessen
- neue Prozesse reihen sich hinten in Warteschlange ein
- Strategie: Scheduler wählt jeweils nächsten Prozess in der Warteschlange
- Prozess arbeitet, bis er endet oder für I/O blockiert (typ. nicht präemptiv)
- nach I/O-Unterbrechung reiht sich Prozess vorne wieder ein



FCFS - Beispiel

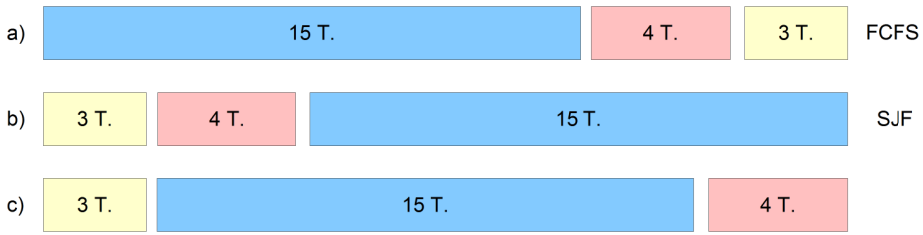
Drei Prozesse mit Rechendauern:

T1: 15 Takte
T2: 4 Takte
T3: 3 Takte

Durchschnittliche Ausführungsdauer (Verweil- bzw. Durchlaufzeit): (Rechendauer + Wartezeit):

a) (15 + 19 + 22) / 3 = 18,67
b) (3 + 7 + 22) / 3 = 10,67
c) (3 + 18 + 22) / 3 = 14,33

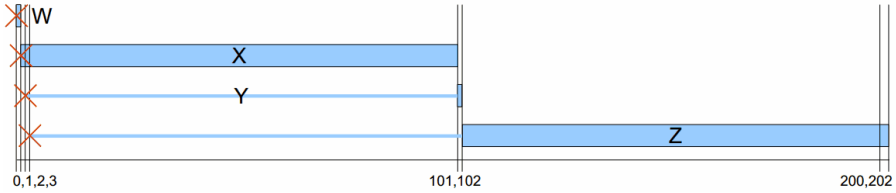
3 Varianten:



FCFS - Beispiel

- FCFS bevorzugt lang laufende Prozesse (in Bezug auf Ausführungsdauer)
- Beispiel: 4 Prozesse W, X, Y, Z

Prozess	Ankunftzeit	Service Time T_s (Rechenzeit)	Startzeit	Endzeit	Turnaround T_r (Endzeit-Ankunftzeit)	T_r/T_s
W	0	1	0	1	1	1,00
X	1	100	1	101	100	1,00
Y	2	1	101	102	100	**100,00**
Z	3	100	102	202	199	1,99



FCFS: CPU- vs. I/O-lastig

FCFS bevorzugt CPU-lastige Prozesse

- während CPU-lastiger Prozess läuft, müssen alle anderen warten
- I/O-lastiger Prozess läuft nur bis zu nächster Unterbrechung für I/O

  OHNE dafür einen Ausgleich zu bekommen
- ineffiziente Nutzung der I/O

Frage: kann ein Prozess verhungern ???

Shortest Job First (SJF)

Arbeitsweise:

- Strategie: Scheduler wählt Prozess, der am kürzesten laufen wird
- dabei: nächste Rechendauer (Burst) aller Prozesse bekannt oder geschätzt
- typ. nicht präemptiv

Eigenschaften:

- minimiert durchschnittliche Verweilzeit aller Prozesse (--> FCFS Bsp. b))

Frage: kann ein Prozess verhungern ???

Laufzeiten / Bursts

Woher wissen, wie lange Prozesse laufen werden ?

- Batch-System:
  - Programmierer muss Laufzeit bei Job-Beauftragung schätzen

    --> bei grober Fehleinschätzung: Job wird abgebrochen
  - System, auf dem immer gleiche / ähnliche Jobs laufen

    --> Statistiken führen
- Interaktive Systeme:
  - Durchschnitt der bisherigen Burst-Längen berechnen

Ohne diese Informationen ist dieses Verfahren nicht praktisch anwendbar

Burst-Dauer Prognose (1)

Einfachste Variante: Mittelwert

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i = \frac{1}{n}T_n + \frac{n-1}{n}S_n$$

mit

- T_i: Dauer des i-ten CPU-Bursts des Prozesses
- S_i: vorausgesagte Dauer des i-ten CPU-Burst
- S_1: vorausgesagte Dauer des 1. CPU-Burst (nicht berechnet)

Gleitender exponentieller Durchschnitt

$$S_{n+1} = \alpha T_n + (1-\alpha) S_n, \quad \alpha \in [0,1]$$

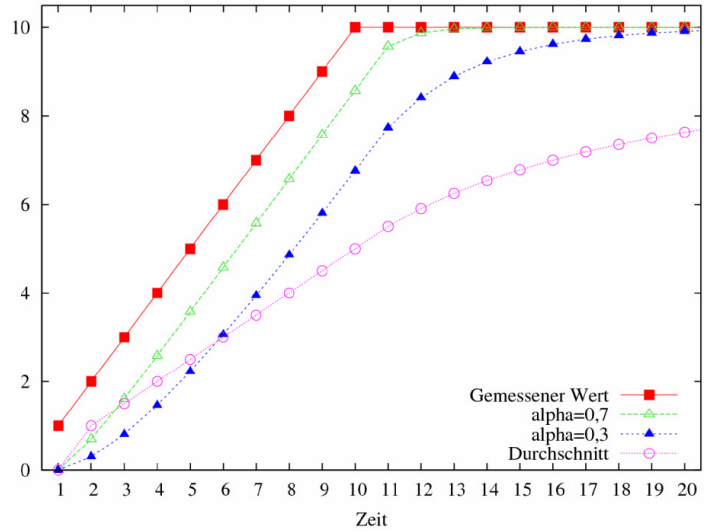
Beispiel:

$$S_2 = \alpha T_1 + (1-\alpha) S_1$$

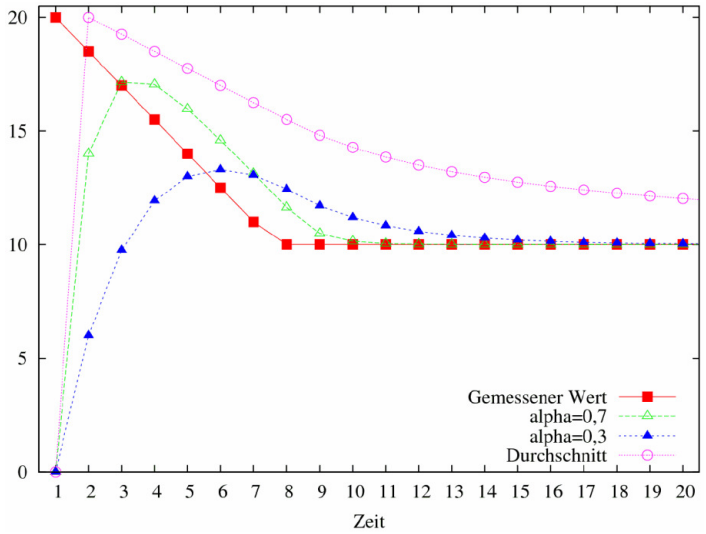
$$S_3 = \alpha T_2 + (1-\alpha) S_2 = \alpha T_2 + (1-\alpha)\alpha T_1 + (1-\alpha)^2 S_1$$

$$S_{n+1} = \sum_{i=0}^{n} (1-\alpha)^{n-i} \alpha T_i \quad \text{mit } T_0 := S_1$$

Burst-Dauer Prognose (2)



Burst-Dauer Prognose (3)



Shortest Remaining Time (SRT)

Arbeitsweise:

ähnlich SJF, aber

- regelmäßige Neuberechnung der voraussichtlichen Restzeit der Prozesse
- Strategie: Scheduler wählt Prozess/Thread mit kürzester Restlaufzeit
- für kürzeren (auch neuen) Job wird aktiver unterbrochen (präemptiv)
- wie bei SJF gute Laufzeitprognose nötig

Eigenschaften:

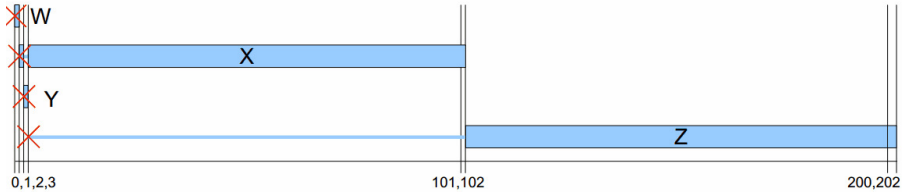
- minimiert durchschnittliche Wartezeit aller Prozesse

Frage: kann ein Prozess verhungern ???

SRT - Beispiel

- <-> FCFS-Beispiel: SRT unterbricht jetzt X,
- denn Y kommt zwar später, ist aber kürzer

Prozess	Ankunftzeit	Service Time T_s (Rechenzeit)	Startzeit	Endzeit	Turnaround T_r (Endzeit-Ankunftzeit)	T_r/T_s
W	0	1	0	1	1	1,00
X (1)	1	100	1	2 (*)		
Y	2	1	2	3	1	1,00
X (2)			3	102	102-1=101	1,01
Z	3	100	102	202	199	1,99



Scheduling-Verfahren

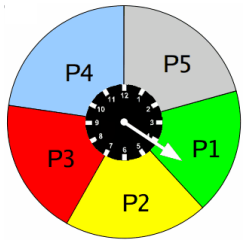
für

Interaktive-Systeme

Round Robin / Time Slicing (1)

Arbeitsweise

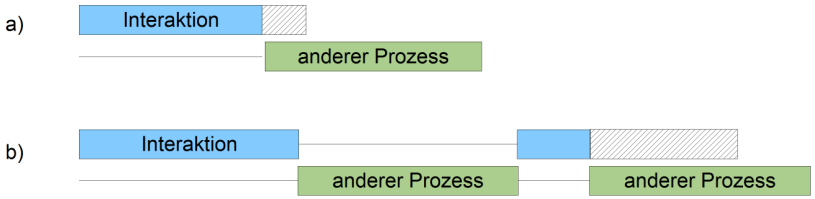
- alle bereiten Prozesse/Threads in einer Warteschlange
- jedem Prozess eine Zeitscheibe (Quantum / Time Slice) zuordnen
- ist Prozess nach Ablauf der Zeitscheibe noch aktiv, dann
  - Prozess verdrängen (preemtion):
    - in Zustand „bereit“ versetzen
    - ans Ende der Warteschlange
  - nächsten Prozess aktivieren
- blockierter Prozess, der „bereit“ wird, wird hinten in Warteschlange eingereiht



Round Robin: Quantum

Kriterien für die Wahl des Quantums:

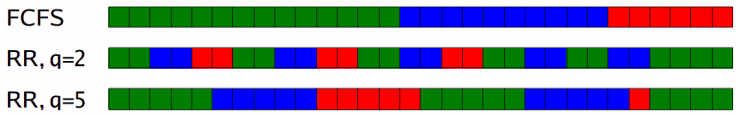
- Größe muss in Verhältnis zur Dauer eines Kontext-Wechsels stehen
  - zu groß: evtl. lange Verzögerungen (<-> Antwortzeiten)
  - zu klein: Overhead durch häufige Kontext-Wechsel
- --> oft Quantum etwas größer als typische Zeit zur Bearbeitung einer Interaktion z.B. zwischen 10-100ms



Round Robin: Beispiel

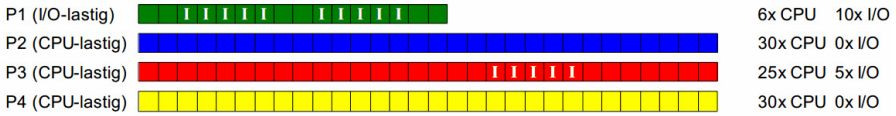
Szenarion: 3 Prozesse

- FCFS (einfache Warteschlange, keine Unterbrechung)
- Round Robin mit Quantum 2
- Round Robin mit Quantum 5

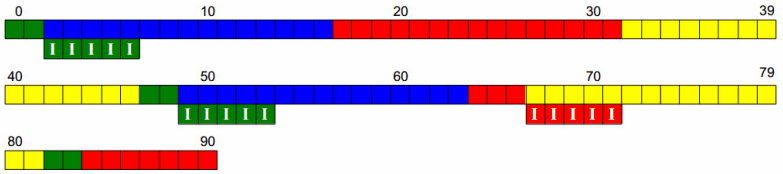


Round Robin: I/O vs. CPU-lastig

Idealer Verlauf (wenn jeder Prozess exklusiv läuft)



Ausführreihenfolge mit Round Robin, Zeitquantum 15:



Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

*) im Zustand *bereit*, nicht *blockiert*!

Virtual Round Robin (1)

Beobachtung:

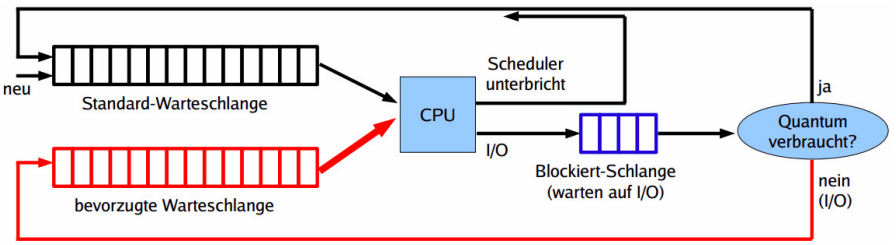
- Round Robin unfair gegenüber I/O-lastigen Prozessen
  - CPU-lastige nutzen ganzes Quantum
  - I/O-lastige nur einen Bruchteil

Lösungsvorschlag:

- nicht verbrauchten Quantum-Anteil als Guthaben des Prozesses merken
- sobald blockierter Prozess wieder bereit (I/O abgeschlossen), Restguthaben sofort aufbrauchen

Virtual Round Robin (2)

- Prozesse, die Quantum verbrauchen, wie bei normalem RR behandeln
  --> zurück in Warteschlange
- Prozesse, die wegen I/O blockieren und nur u < q verbraucht haben, in Zusatzwarteschlange
- Scheduler wählt bevorzugt aus Zusatzwarteschlange
  - Quantum dann q - u, Prozess bekommt den Rest dessen, was ihm zusteht



Prioritätsbasiertes Scheduling (1)

- Idee:
  - jedem Prozess einen Prioritätswert zuordnen
- Scheduler wählt
  - Prozesse mit höchster Priorität
  - bei mehreren Prozessen gleicher Priorität: Round-Robin
- i.d.R. präemptiv
- Priorität:
  - statisch: bei Prozesserzeugung fest vergeben (häufig bei Echtzeitsystemen)
  - dynamisch:
    - je nach Verhalten des Prozesses adaptiv angepasst, d.h. wird vom Scheduler regelmäßig neu berechnet
    - z.B.
      - --> Aging,
      - abh. v. Länge des letzten CPU-Bursts (~SJF)

Prioritätsbasiertes Scheduling (2)

Prozesse können sich gegenseitig blockieren

Prioritätsinversion:

- Prozess hoher Priorität benötigt ein Betriebsmittel
- Prozess niedriger Priorität besitzt dieses, wird aber vom Scheduler nicht aufgerufen, weil es mittlere-priore Prozesse gibt
- --> beide Prozesse kommen nie dran
- Auswege:
  - Prioritätsvererbung
  - Aging

Prioritätsvererbung:

- Prozess leiht seine hohe Prio dem mit niedriger Prio, der das benötigte Betriebsmittel hält.

Aging:

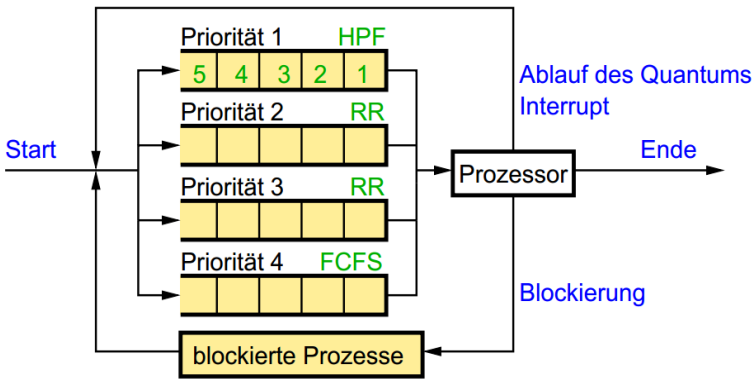
- Priorität eines Prozesses, der bereit ist und wartet, wird regelmäßig erhöht
- Prioritäten des aktiven und aller nicht-bereiten (blockierten) Prozesse bleiben gleich
- Ergebnis:
  lang wartender Prozess erreicht irgendwann ausreichend hohe Priorität, um aktiv zu werden

Multilevel Scheduling (1)

Multilevel Scheduling

- Einteilung der Prozesse in Prioritätsklassen
- mehrere Warteschlangen für bereite Prozesse je Prioritätsklasse
- jede Warteschlange kann eigene Auswahlstrategie haben
- zusätzlich:
  - Strategie zur Auswahl der aktuellen Warteschlange
    - z.B. Prioritäten oder Round-Robin
- statisch: Prozess fest einer Warteschlange zugeordnet
- dynamisch: Prozess kann je nach Verhalten zwischen Warteschlangen wechseln

Multilevel Scheduling (2)



Multilevel Scheduling (3)

Beispiel: Multilevel *Feedback* Scheduling

- dynamische Prioritäten und variable Quantenlänge
- mehrere Warteschlangen mit unterschiedlicher Priorität
  - innerhalb einer WS: Round Robin
  - bei niedrigerer Priorität: längeres Quantum
- falls Prozess Quantum aufbraucht:
  - erniedrigen der Priorität, erhöhen des Quantums

    --> CPU-lastiger Prozess erhält längeres Quantum und wird seltener unterbrochen
- sonst: Priorität nicht verändern bzw. wieder erhöhen

  --> I/O-lastiger (bzw. interaktiver) Prozess erhält bevorzugt die CPU, aber nur für kurze Zeit

Scheduling auf SMP-Systemen (1)

Mögliche Randbedingungen

- Hard Affinity: Thread kann/darf nur auf bestimmten CPUs laufen
- Soft Affinity: Thread soll bevorzugt auf CPU laufen, auf der er zuletzt lief (wegen Caches!)

Nachfolgend: nur grobe Skizzierung:

Scheduling auf SMP-Systemen (2)

Auswahl einer CPU für bereit gewordenen Thread

- **freie CPU ?** (bei Hard-Affinity: auf der der Thread laufen darf)
  - ja: --> Zuweisung
  - Thread geringerer Priorität auf
    - der letzten CPU dieses Threads ?
      - ja:
        --> Verdrängung und Zuweisung
    - auf irgendeiner CPU?
      - ja:
        --> Verdrängung und Zuweisung
  - sonst: Thread muss weiter warten

Auswahl eines bereiten Threads für frei gewordene CPU

- **bereiter Thread** mit höchster Priorität ist Primärkandidat
- Zuweisung an Primärkandidat, falls
  - dies seine letzte CPU war (Soft-Affinity)
  - er länger als X Quanten wartete
  - er Echtzeitanforderungen hat (Prio > XX)
- ansonsten:
  - Prüfung des nächsten bereiten Threads und ggf. Zuweisung
  - Zuweisung ggf. an Primärkandidat, falls bisher keine Zuweisung

Praxisbeispiele

Praxis: Beeinflussung des Schedulings

nice

Startet einen Prozess mit herabgesetzter Scheduling-Priorität (höherer nice-Wert)

```
bash% nice -10 <prog>
```

Es gibt auch einen entsprechenden System-Call

Der Superuser darf die Priorität auch erhöhen

renice

Ändern der Priorität eines laufenden Prozesses

Prioritäten werden zyklisch neu berechnet:

NeuePrio = Basis-Prio + CPU-Nutzung/2 + nice-value

Praxis: CPU-Affinity

Steuerung der CPU-Affinität auf der Shell

- taskset -c 1,2 -p <PID>
- **taskset** 0x00000003 -p <PID> #CPU0+1, change of existing prog
- taskset -c 0,1 <prog> #CPU0+1, launch of new prog

Steuerung der CPU-Affinität im Programm

- #define _GNU_SOURCE
- #include <sched.h>
- int **sched_setaffinity**(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
- int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);

Praxis: pthread Scheduling-Policy

Steuerung des pthread-Schedulings

#include <pthread.h>

pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);

pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);

zu linken mit -lpthread

Zusammenfassung

Zusammenfassung (1)

- Scheduling:
  - Entscheidung, welcher Prozess wann, wie lange, und ggf. auf welcher CPU rechnen darf
  - Unterschiedliche Anforderungen, je nach Sichtweise und Betriebsmodus
  - Nicht-präemptives und präemptives Scheduling
    - präemptiv: BS kann einem Thread die CPU zwangsweise entziehen
- Scheduling-Algorithmen:
  - FCFS: FIFO-Warteschlange rechenbereiter Threads, nicht-präemptiv
  - SJF: Shortest Job First
    - optimiert Durchlaufzeit von Jobs

Zusammenfassung (2)

- Scheduling-Algorithmen ...
  - Round Robin (RR): präemptive Version von FCFS
    - Prozesse dürfen nur bestimmte Zeit rechnen
  - Prioritätsbasiertes Scheduling:
    - nur der Prozess mit höchster Priorität bekommt CPU (bzw. die n höchstprioren Prozesse bei n CPUs)
  - *Multilevel* Scheduling:
    - mehrere Warteschlangen mit unterschiedlicher Auswahlstrategie
    - statisches Multilevel Scheduling:
      - feste Zuordnung Thread → Warteschlange
    - multilevel Feedback Scheduling
      - dynamische Zuordnung Thread → Warteschlange