Prozessverwaltung (1)

Ute Bormann, TI2

2023-10-13

Prozessverwaltung in Unix

Verschiedene Teilaufgaben:

④ • Prozesserzeugung und -termination

⇒② • Prozesswechsel (Scheduling)

⑤ • Nebenläufigkeit

⑥ • Interprozesskommunikation (Spezialform: Signale)

①

⇒③ • Datenstrukturen zur Prozessverwaltung

Inhalt

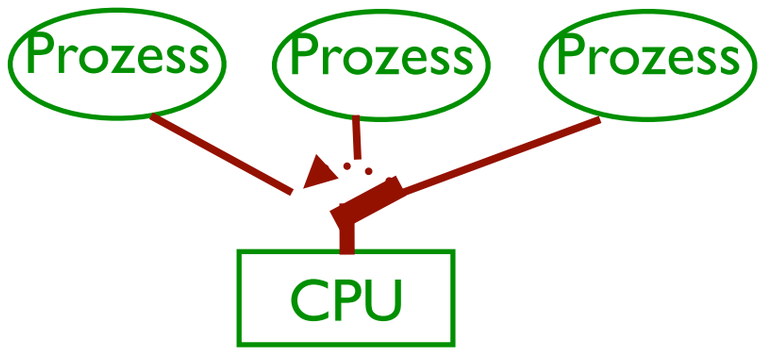
1. Prozessumschaltung
2. Scheduling
3. Kontrolliertes Warten
4. Unix-Datenstrukturen zur Prozessverwaltung

Teil 1:
Prozessumschaltung

Prozessverwaltung in Unix: Der Prozessbegriff

- Prozess: „Programm in Ausführung“
- Mehrere Prozesse teilen sich Betriebsmittel:
  - Ein-/Ausgabegeräte
  - die eine CPU (falls Einprozessorsystem... im folgenden vereinfachend angenommen)

⇒ zu jedem Zeitpunkt kann nur ein Prozess in Ausführung sein:



Gründe für Prozessumschaltung (CPU-Abgabe)

- Prozess hat Programm abgearbeitet
- Prozess wird abgebrochen: z.B. ctl-c (SIGINT), Division durch 0, ...

  ➔ CPU ist frei
- Prozess wird inaktiviert: ctl-z (SIGSTOP)
- Prozess kann momentan nicht weiterlaufen, weil er auf etwas wartet („legt sich schlafen“ ➔ sleep): z.B. Shell wartet auf nächste Kommando-Eingabe

  ➔ CPU soll nicht mitwarten

Gründe für Prozessumschaltung (CPU-Abgabe)

- Prozess hat Programm abgearbeitet
- Prozess wird abgebrochen: z.B. ctl-c (SIGINT), Division durch 0, ...

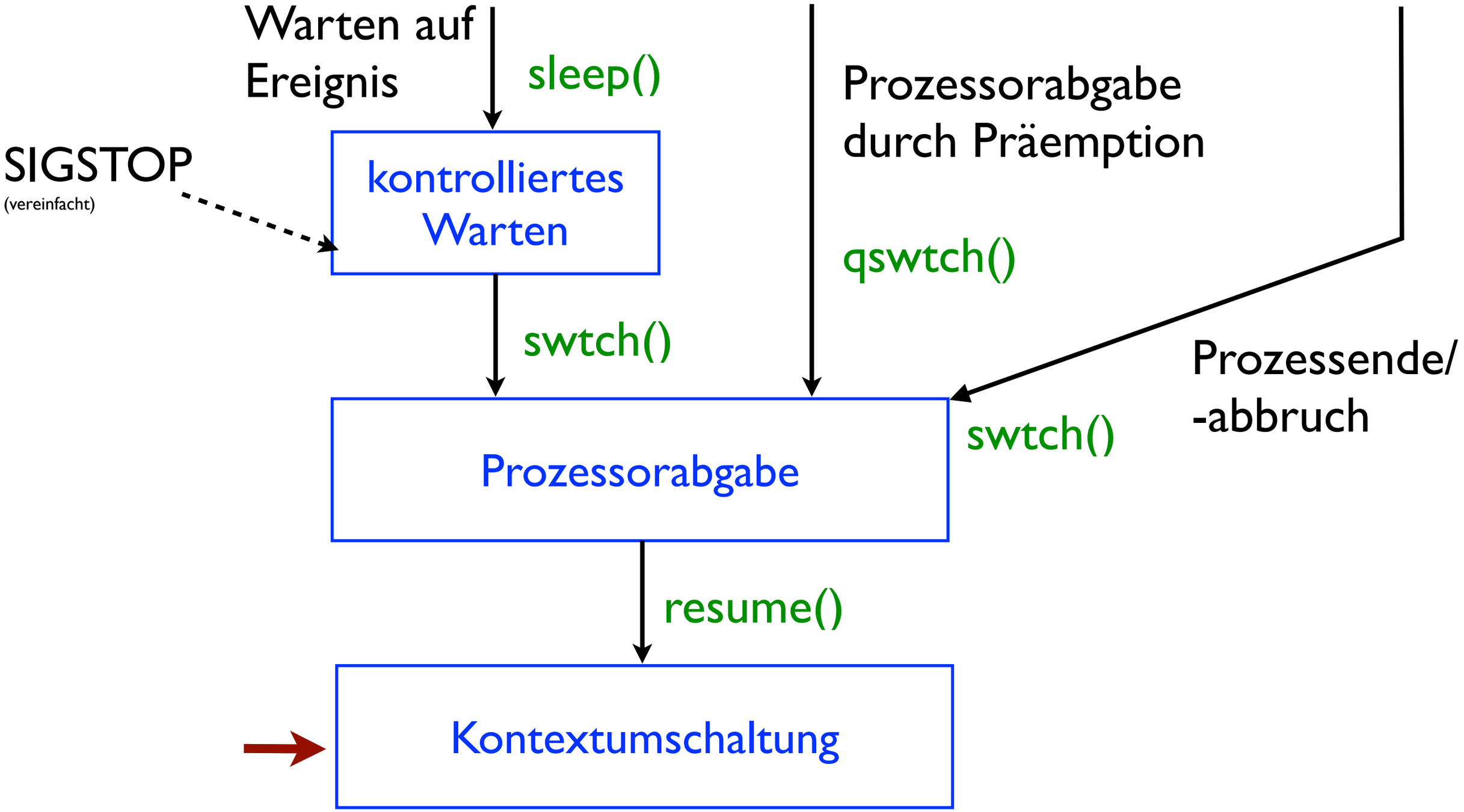
  ➔ CPU ist frei
- Prozess wird inaktiviert: ctl-z (SIGSTOP)
- Prozess kann momentan nicht weiterlaufen, weil er auf etwas wartet („legt sich schlafen“ ➔ sleep): z.B. Shell wartet auf nächste Kommando-Eingabe

  ➔ CPU soll nicht mitwarten
- Prozess hat noch mehr zu tun, hatte CPU aber schon eine ganze Weile
  - andere Prozesse (und deren Benutzer) warten schon lange
  - schlechtes Antwortzeitverhalten bei interaktiven Anwendungen

  ➔ „gerechtere“ Nutzung der CPU

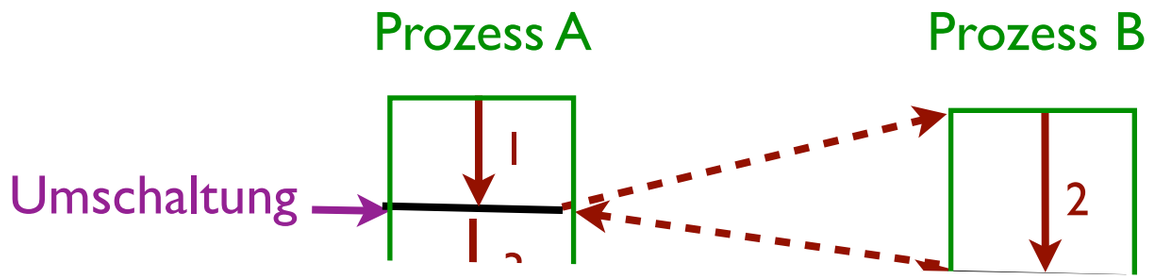
  ➔ „freiwillige“ Abgabe der CPU (Präemption)
    - nach Ablauf einer Zeitscheibe (nach x Clock Ticks)
      ➔ CPU-Abgabe (bei Clock-Interrupt-Behandlung angestoßen)

Prozesswechsel:
wird in Unix über drei Schichten realisiert



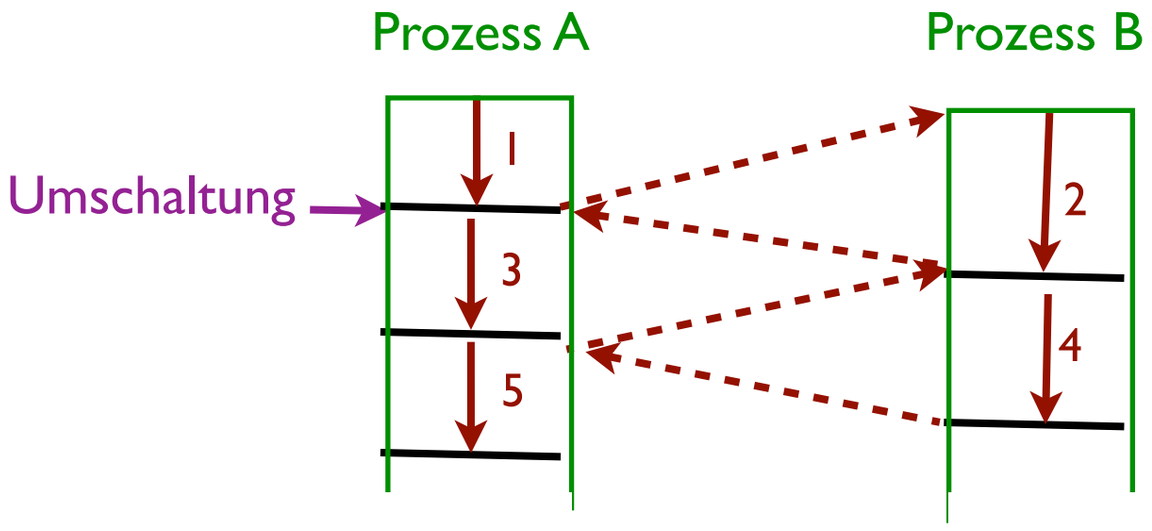
1) Kontextumschaltung (resume()) vereinfacht

- laufender Prozess wird inaktiviert
- anderer Prozess wird fortgeführt

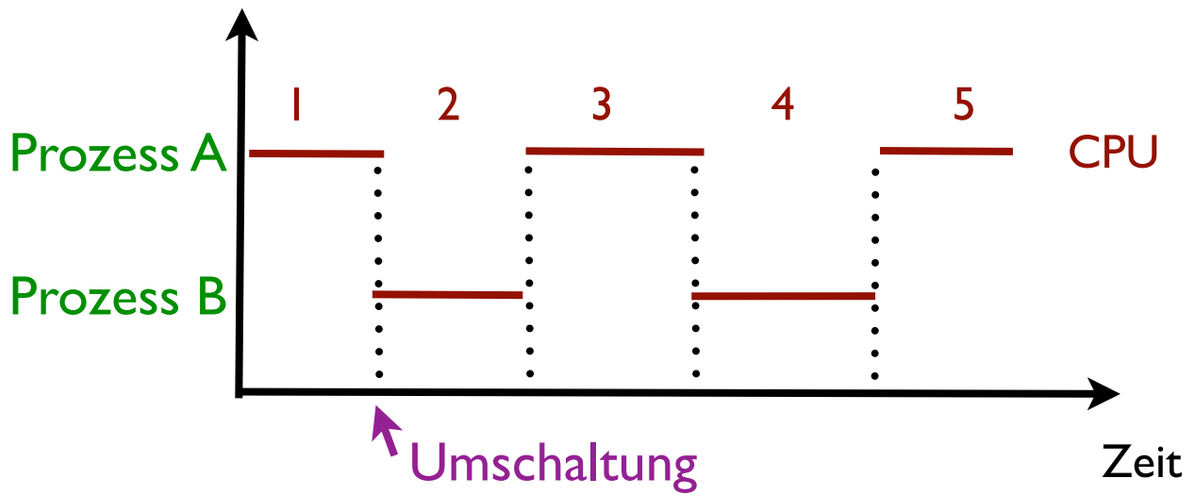


1) Kontextumschaltung (resume()) vereinfacht

- laufender Prozess wird inaktiviert
- anderer Prozess wird fortgeführt

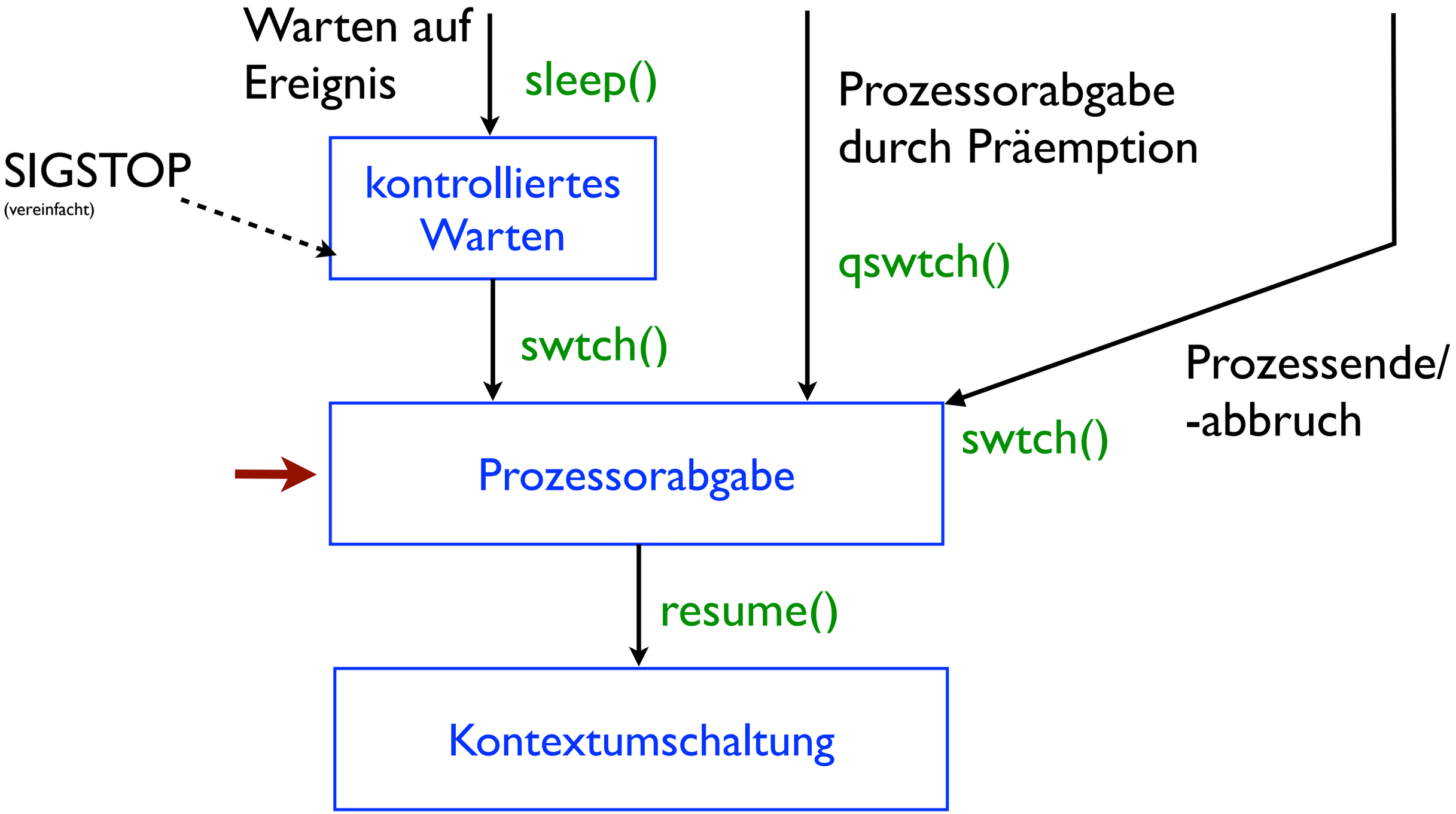


- Ablauf über die Zeit betrachtet:



- Aufgaben von resume():
  - Retten des Kontexts des zu unterbrechenden Prozesses
  - Wiederherstellen des aktuellen Kontexts des fortzuführenden Prozesses
  - Verweis auf fortzuführenden Prozess wird als Parameter übergeben
    - sehr hardwarenah → spezielle Maschineninstruktionen → sehr schnell

Prozesswechsel:
wird in Unix über drei Schichten realisiert



2) Prozessorabgabe (qswtch(), swtch())

- Bei qswtch(): Laufender Prozess wird als noch immer lauffähig markiert

  ⇒ Einhängen in die Run-Queue

- Auswahl des fortzuführenden Prozesses

  ⇒ muss sich ebenfalls in Run-Queue befinden

- Auswahl gemäß Scheduling-Strategie

  ⇒ in Unix von swtch() entkoppelt

- Aufrufen von resume() zur Kontextumschaltung

Fragen – Teil 1

- Nenne verschiedene Gründe für eine Prozessumschaltung.

Teil 2:
Scheduling

Scheduling

- Nach welchem Verfahren wird CPU zugeteilt?

  ⇒ Prozess muss lauffähig sein (Run-Queue)

- Achtung:
  Scheduling darf nicht selbst zuviel Zeit verbrauchen

  ⇒ kostet auch CPU-Zeit

- (Historisch) Verschiedene Ansätze:
  - Auftrags-orientiert
  - Zeitscheiben-orientiert

Auftrags-orientiertes Scheduling

a) FIFO (First in, first out)

- Neuer Prozess wird hinten in Auftragswarteschlange eingereiht

  ⇒kommt dran, sobald alle davor abgearbeitet wurden

Auftrags-orientiertes Scheduling

a) FIFO (First in, first out)

- Neuer Prozess wird hinten in Auftragswarteschlange eingereiht
  ⇒kommt dran, sobald alle davor abgearbeitet wurden

b) Shortest Job Next

- Prozesse werden nach ihrer voraussichtlichen Bearbeitungszeit sortiert
  ⇒kürzere Prozesse werden bevorzugt
  (mehr „zufriedene“ Kunden)

Auftrags-orientiertes Scheduling

a) FIFO (First in, first out)

- Neuer Prozess wird hinten in Auftragswarteschlange eingereiht
  ⇒kommt dran, sobald alle davor abgearbeitet wurden

b) Shortest Job Next

- Prozesse werden nach ihrer voraussichtlichen Bearbeitungszeit sortiert
  ⇒kürzere Prozesse werden bevorzugt
  (mehr „zufriedene“ Kunden)

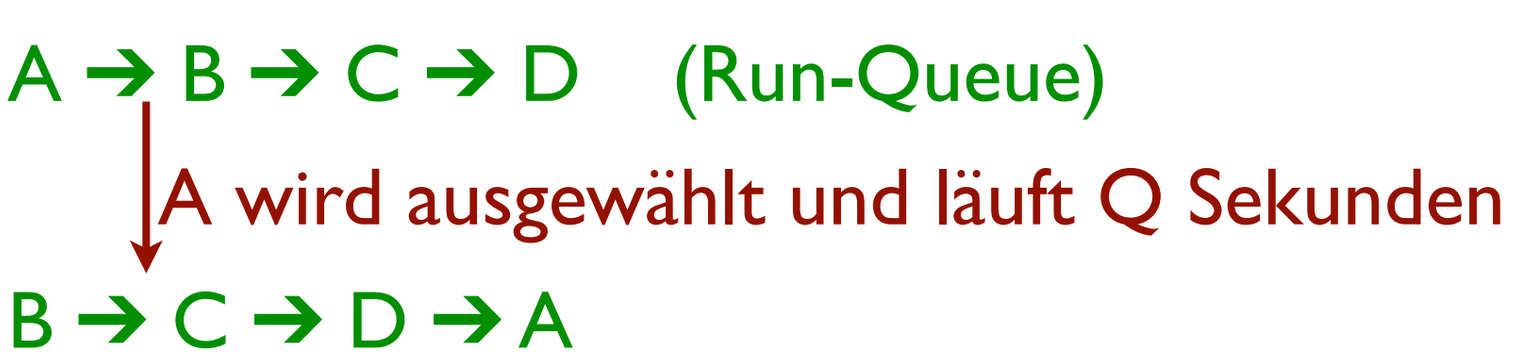
- Beide Verfahren nur für „Batch-ähnliche“ Systeme geeignet
- Interaktive Anwendungen benötigen schnelle Antwortzeiten

  ⇒ Zeitscheibenverfahren

Zeitscheiben-orientiertes Scheduling

a) Round Robin

- Run-Queue wird der Reihe nach abgearbeitet
- Laufender Prozess darf CPU nur für max. Q Sekunden behalten (bei sleep() früher abgeben)
- Danach wird er wieder hinten eingereiht



- Wie groß ist Q?
  - zu klein: Zuviel CPU-Zeit für Prozesswechsel
  - zu groß: Schlechte Antwortzeiten
  - Wert ggf. situationsbedingt bestimmen (z.B. abhängig von Anzahl der lauffähigen Prozesse)

- Round Robin behandelt alle Prozesse gleich ⇒ „gerecht“?
- Nicht immer wünschenswert:
  - Ein-/Ausgabeintensive Prozesse belegen CPU häufig nur kurz
    ⇒ dann wieder warten auf Gerätefertigmeldung
  - Belegen u.U. Gerät, während sie warten
    (exklusiver Gerätezugriff)
    ⇒ andere Prozesse kommen nicht ran
  - Aktivitäten zum Anfertigen von Sicherungskopien oder von Statistiken sind u.U. gar nicht so eilig

⇒ Prozesse haben unterschiedliche Prioritäten

- Round Robin behandelt alle Prozesse gleich ⇒ „gerecht“?
- Nicht immer wünschenswert:
  - Ein-/Ausgabeintensive Prozesse belegen CPU häufig nur kurz
    ⇒ dann wieder warten auf Gerätefertigmeldung
  - Belegen u.U. Gerät, während sie warten
    (exklusiver Gerätezugriff)
    ⇒ andere Prozesse kommen nicht ran
  - Aktivitäten zum Anfertigen von Sicherungskopien oder von Statistiken sind u.U. gar nicht so eilig

  ⇒ Prozesse haben unterschiedliche Prioritäten
- Verfeinerungen von Round Robin denkbar:
  - Unterschiedlich lange Zeitscheiben
  - Mehrfaches Einhängen in die Run-Queue
- Oder Abkehr vom reinen Round Robin
  ⇒ Auswahl des Prozesses mit der (momentan) „höchsten“ Priorität

b) Scheduling nach Prioritäten

- Woraus ermittelt sich Priorität?
  - Statische Aspekte
    - z.B. Dringlichkeit/„Wichtigkeit“
    - z.B. (beabsichtigte) Betriebsmittelnutzung

      ⇒ nicht immer klar vorhersehbar

    ⇒ Starre Prioritäten

    ⇒ Bei hoher Auslastung können Prozesse „verhungern“

b) Scheduling nach Prioritäten

- Woraus ermittelt sich Priorität?
  - Statische Aspekte
    - z.B. Dringlichkeit/„Wichtigkeit“
    - z.B. (beabsichtigte) Betriebsmittelnutzung

      ⇒ nicht immer klar vorhersehbar

    ⇒ Starre Prioritäten

      ⇒ Bei hoher Auslastung können Prozesse „verhungern“

  - Dynamische Aspekte
    - z.B. CPU-Nutzung in jüngerer Zeit

      ⇒ „Round-Robin“-Prinzip

- Sinnvolle Kombination anwenden

  ⇒ aber nicht zu aufwendig (keine „optimalen“ Lösungen)

In Unix: Kombination

a) Neuberechnung der Prioritäten nach Ablauf der Zeitscheibe

- (einstellbare) Basispriorität (nice())
- CPU-Nutzung in jüngerer Zeit:
  - Laufzeit wird auf CPU-Konto des Prozesses regelmäßig aufgeschlagen
  - CPU-Konto wird mit der Zeit wieder reduziert, damit langlebige Prozesse nicht benachteiligt werden

⇒ Je kleiner der Wert, desto höher die Priorität!

Beispiel

bei t=0:

Prozess	Basis-prior. (nice)	CPU-Konto
A	0	0
B	60	0

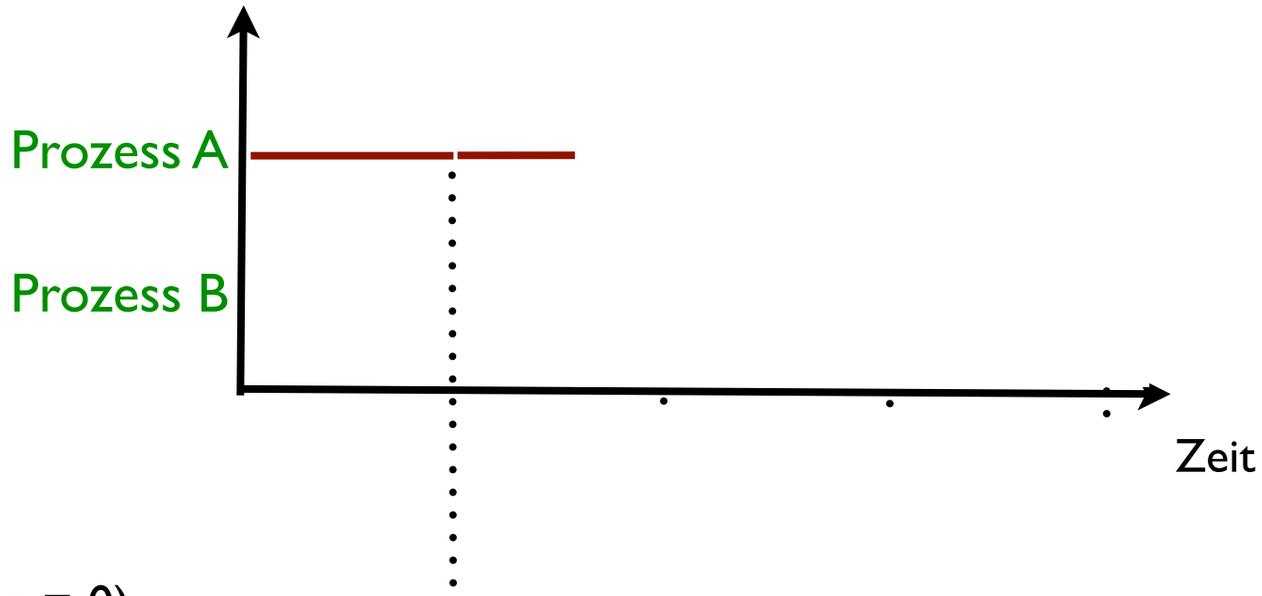
⇒ Prozeß A wird ausgewählt.

Beispiel

bei t=0:

Prozess	Basis-prior. (nice)	CPU-Konto
A	0	0
B	60	0

⇒ Prozeß A wird ausgewählt.



Prozess A (nice = 0)

Voriges CPU-Konto	0
⇓ + Aufschlag (%CPU)	100
⇓ Veraltern (/2)	
Aktuelles CPU-Konto	50
⇓ + Basispriorität	
Effektive Priorität	0 / 50

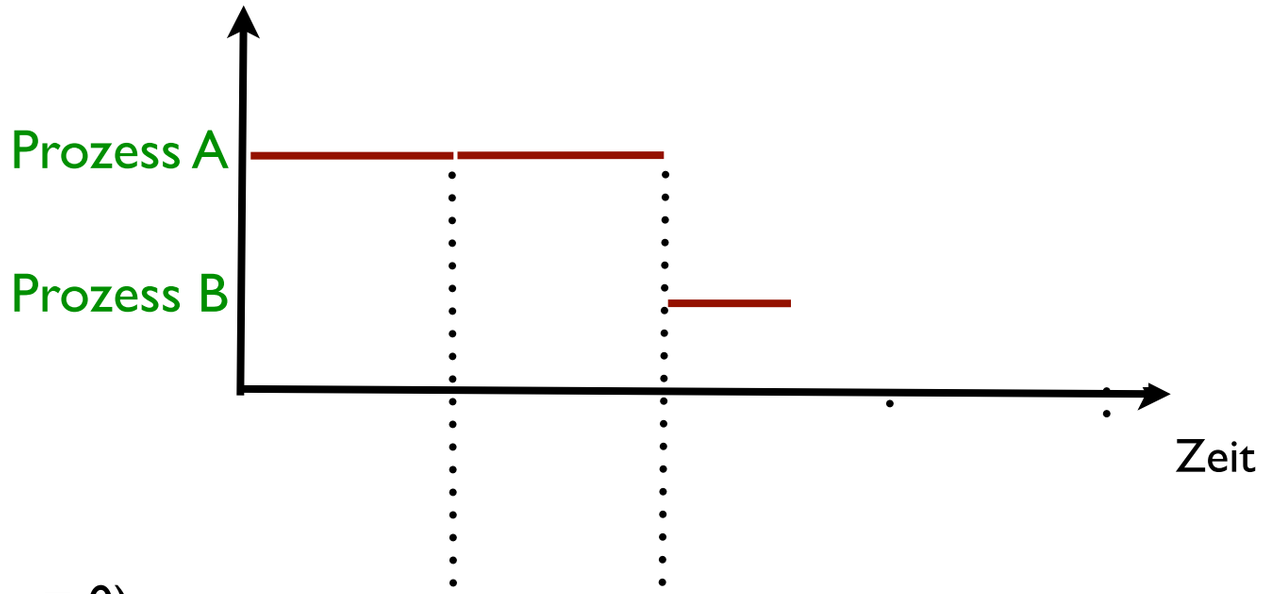
Prozess B (nice = 60)

Voriges CPU-Konto	0
⇓ + Aufschlag (%CPU)	0
⇓ Veraltern (/2)	
Aktuelles CPU-Konto	0
⇓ + Basispriorität	
Effektive Priorität	60 / 60

Beispiel

bei t=0:

Prozess	Basis-prior. (nice)	CPU-Konto
A	0	0
B	60	0



Prozess A (nice = 0)

Voriges CPU-Konto	0	50
⇓ + Aufschlag (%CPU)	100	150
⇓ Veraltern (/2)		
Aktuelles CPU-Konto	50	75
⇓ + Basispriorität		
Effektive Priorität	0	50	75

Prozess B (nice = 60)

Voriges CPU-Konto	0		0
⇓ + Aufschlag (%CPU)	0		0
⇓ Veraltern (/2)			
Aktuelles CPU-Konto	0		0
⇓ + Basispriorität			
Effektive Priorität	60	60	60

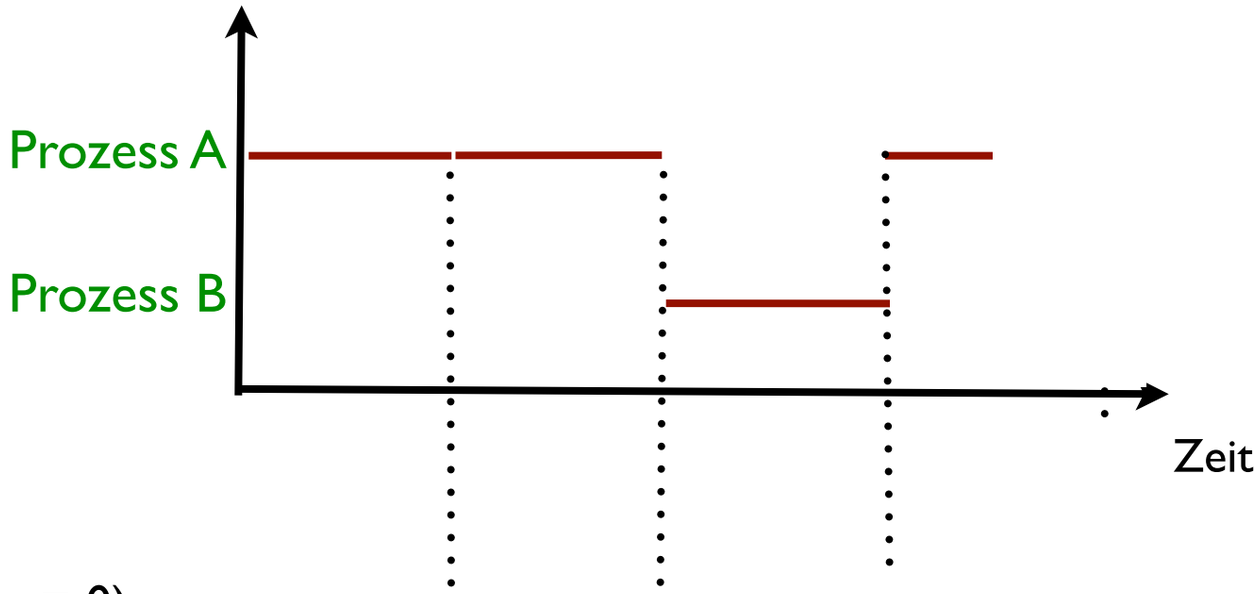
⇒ Prozeß B wird ausgewählt.

Beispiel

bei t=0:

Prozess	Basis-prior. (nice)	CPU-Konto
A	0	0
B	60	0

⇒ Prozeß A wird ausgewählt.



Prozess A (nice = 0)

Voriges CPU-Konto	0	50	75
⇓ + Aufschlag (%CPU)	100	150	0
⇓ Veraltern (/2)			
Aktuelles CPU-Konto	50	75	37,5
⇓ + Basispriorität			
Effektive Priorität	0	50	75	37,5

Prozess B (nice = 60)

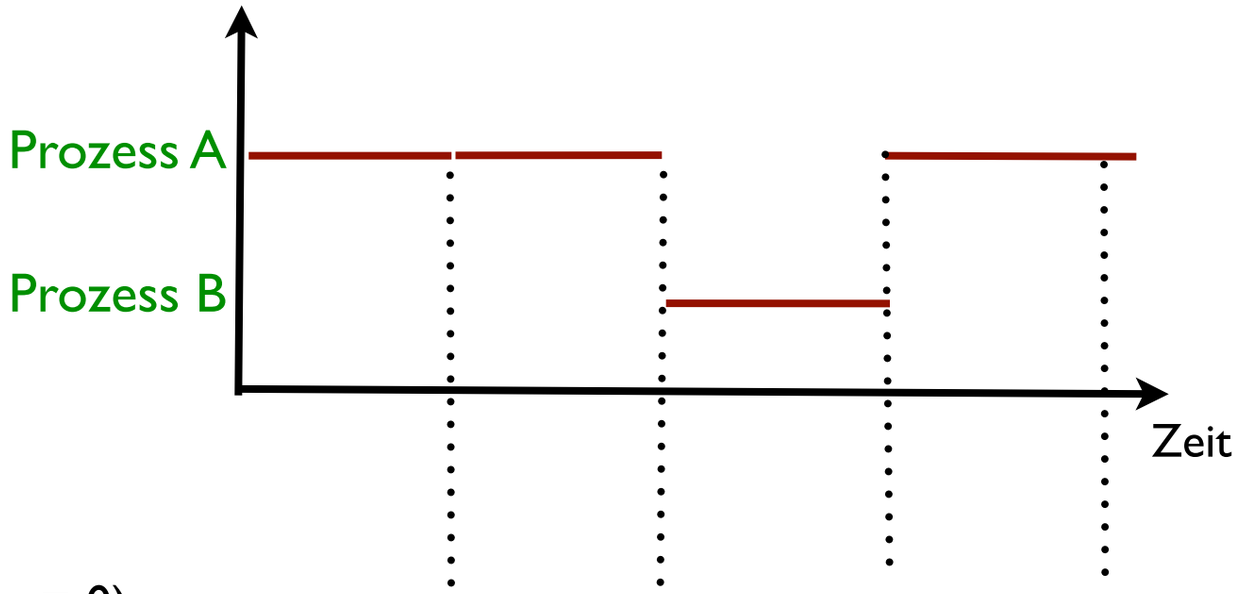
Voriges CPU-Konto	0	0	0
⇓ + Aufschlag (%CPU)	0	0	100
⇓ Veraltern (/2)			
Aktuelles CPU-Konto	0	0	50
⇓ + Basispriorität			
Effektive Priorität	60	60	60	110

Beispiel

bei t=0:

Prozess	Basis-prior. (nice)	CPU-Konto
A	0	0
B	60	0

⇒ Prozeß A wird ausgewählt.



Prozess A (nice = 0)				
Voriges CPU-Konto	0	50	75	37,5
⇓ + Aufschlag (%CPU)	100	150	0	137,5
⇓ Veraltern (/2)				
Aktuelles CPU-Konto	50	75	37,5	68,75
⇓ + Basispriorität				
Effektive Priorität	0	50	75	37,5	68,75

Prozess B (nice = 60)				
Voriges CPU-Konto	0	0	0	50
⇓ + Aufschlag (%CPU)	0	0	100	50
⇓ Veraltern (/2)				
Aktuelles CPU-Konto	0	0	50	25
⇓ + Basispriorität				
Effektive Priorität	60	60	60	110	85

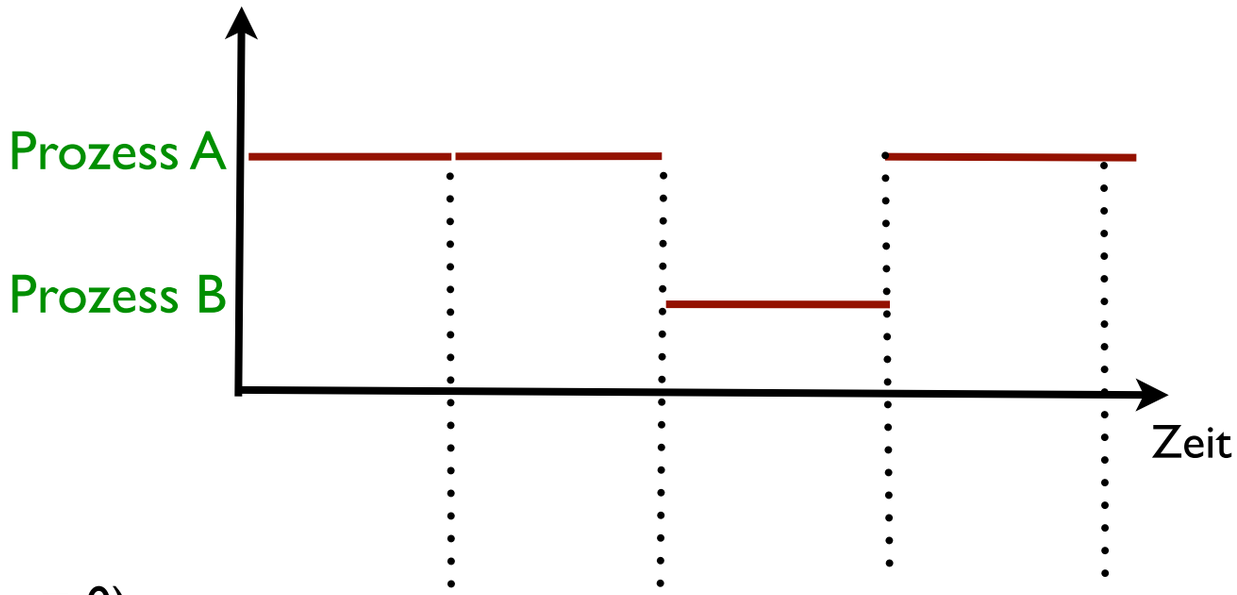
Beispiel

bei t=0:

Prozess	Basis-prior. (nice)	CPU-Konto
A	0	0
B	60	0

⇒ Prozeß A wird ausgewählt.

u.U. Tickless Scheduling: keine festen Zeitscheiben für Neuberechnung



Prozess A (nice = 0)

Voriges CPU-Konto	0	50	75	37,5
⇓ + Aufschlag (%CPU)	100	150	0	137,5
⇓ Veraltern (/2)				
Aktuelles CPU-Konto	50	75	37,5	68,75
⇓ + Basispriorität				
Effektive Priorität	0	50	75	37,5	68,75

Prozess B (nice = 60)

Voriges CPU-Konto	0	0	0	50	
⇓ + Aufschlag (%CPU)	0	0	100	50	
⇓ Veraltern (/2)					
Aktuelles CPU-Konto	0	0	50	25	
⇓ + Basispriorität					
Effektive Priorität	60	60	60	110	85

In Unix: Kombination

a) Neuberechnung der Prioritäten nach Ablauf der Zeitscheibe

- (einstellbare) Basispriorität (nice())
- CPU-Nutzung in jüngerer Zeit:
  - Laufzeit wird auf CPU-Konto des Prozesses regelmäßig aufgeschlagen
  - CPU-Konto wird mit der Zeit wieder reduziert, damit langlebige Prozesse nicht benachteiligt werden

→
- Falls vor Neuberechnung unterbrochener Prozess dann nicht mehr „höchste“ Priorität hat: Flag „runrun“ setzen (vor Rücksprung abfragen)

⇒ Je kleiner der Wert, desto höher die Priorität!

In Unix: Kombination

a) Neuberechnung der Prioritäten nach Ablauf der Zeitscheibe

- (einstellbare) Basispriorität (nice())
- CPU-Nutzung in jüngerer Zeit:
  - Laufzeit wird auf CPU-Konto des Prozesses regelmäßig aufgeschlagen
  - CPU-Konto wird mit der Zeit wieder reduziert, damit langlebige Prozesse nicht benachteiligt werden
  - Falls vor Neuberechnung unterbrochener Prozess dann nicht mehr „höchste“ Priorität hat: Flag „runrun“ setzen (vor Rücksprung abfragen)

b) Warten auf Ereignis

- Ereignis bestimmt u.U. Priorität bei Reaktivierung (verschiedene Arten)

  ⇒ Betriebsmittel sollen u.U. möglichst schnell wieder freigegeben werden

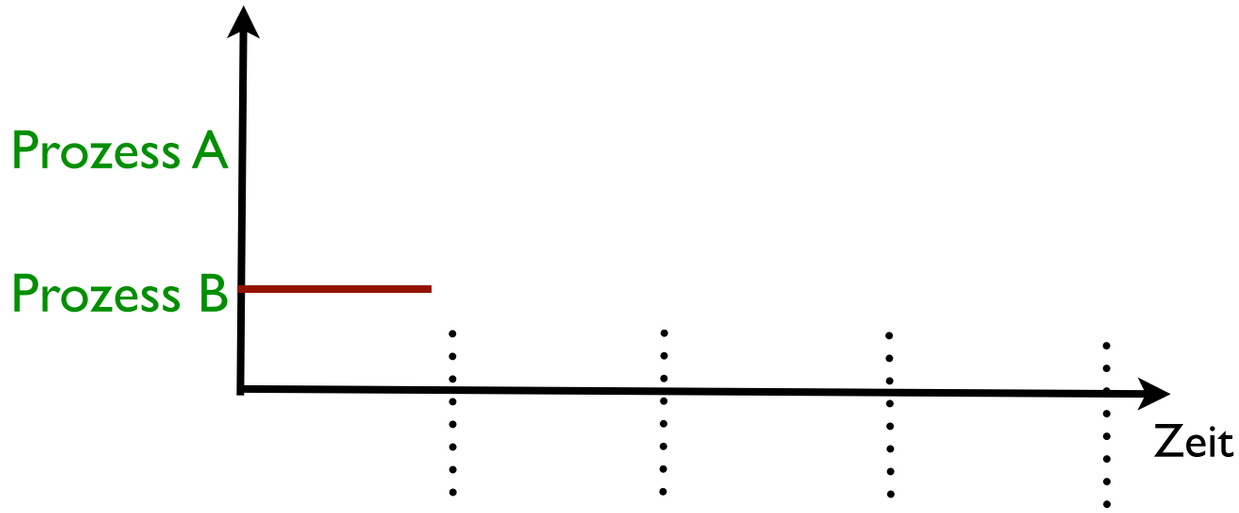
- Bei Reaktivieren dann bei „höherer“ Priorität gegenüber unterbrochenem Prozess ebenfalls Flag „runrun“ setzen

⇒ Je kleiner der Wert, desto höher die Priorität!

Kleine Aufgabe

Gegeben seien zwei gleichzeitig gestartete Prozesse, die dem vorgestellten Unix-ähnlichen Scheduling-Verfahren unterliegen. Wann läuft welcher Prozess?

- Basispriorität:
  Prozess A: nice = 40;
  Prozess B: nice = 20;
- Beide nutzen in zugewiesenen Zeitscheiben 100% der CPU



Prozess	Basis-prior. (nice)	CPU-Konto
A	40	0
B	20	0

Prozess A (nice = 40)

Voriges CPU-Konto

⇓ + Aufschlag (%CPU)

⇓ Veraltern (/2)

Aktuelles CPU-Konto

⇓ + Basispriorität

Effektive Priorität 40

Prozess B (nice = 20)

Voriges CPU-Konto

⇓ + Aufschlag (%CPU)

⇓ Veraltern (/2)

Aktuelles CPU-Konto

⇓ + Basispriorität

Effektive Priorität 20

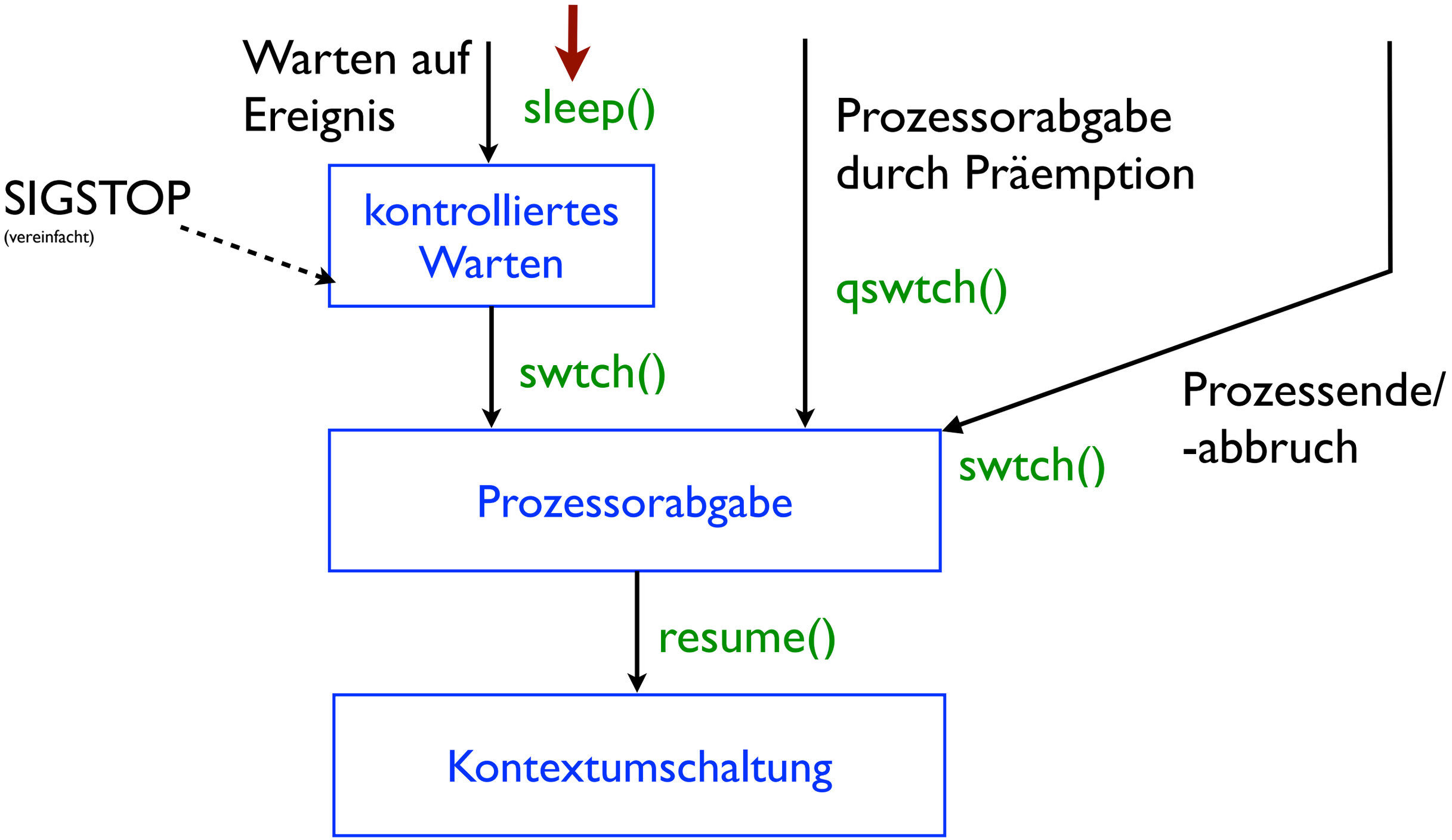
⇒ Prozess B wird zuerst ausgewählt.

Fragen – Teil 2

- Nenne einige Randbedingungen, auf die man beim Entwurf eines *Schedulers* achten sollte. Wie sollten rechenintensive bzw. Ein-/Ausgabe-intensive Prozesse dabei behandelt werden?
- Wie könnte man mit Hilfe eines *Round-Robin-Schedulers* Prozessprioritäten „simulieren“?

Teil 3:
sleep()/wakeup()

Prozesswechsel:
wird in Unix über drei Schichten realisiert



3) Kontrolliertes Warten (sleep())

- Prozess muss auf Ereignis warten ➔ legt sich „schlafen“
- Angabe des Ereignisses
  (damit „aufweckbar“, sobald Ereignis eingetreten)

  ⇒ „Waiting Channel“ (wchan ➔ 32-Bit-Zahl)

- Angabe der Priorität beim Aufwecken
  (abhängig von Art des Ereignisses)
- Einreihen des Prozesses in Sleep-Queue
- Prozessorabgabe mit swtch()

3) Kontrolliertes Warten (sleep())

- Prozess muss auf Ereignis warten → legt sich „schlafen“
- Angabe des Ereignisses
  (damit „aufweckbar“, sobald Ereignis eingetreten)

  ⇒ „Waiting Channel“ (wchan → 32-Bit-Zahl)
- Angabe der Priorität beim Aufwecken
  (abhängig von Art des Ereignisses)
- Einreihen des Prozesses in Sleep-Queue
- Prozessorabgabe mit swtch()

- Aufwecken (wakeup())
  - Wenn Ereignis eingetreten
  - Wird häufig in Interrupthandler festgestellt

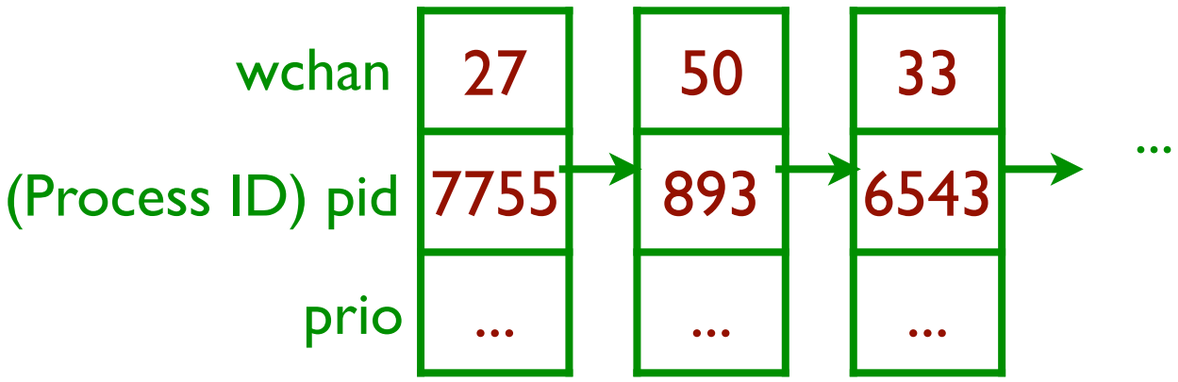
    ⇒ „Aufwecken“ des Prozesses, der auf zugehörigem wchan „schläft“
    (ggf. auch mehr als einer)
  - Einreihen des Prozesses in Run-Queue (und raus aus der Sleep-Queue)

    ⇒ wird ausgewählt, sobald höchste Priorität (ggf. Flag runrun setzen)

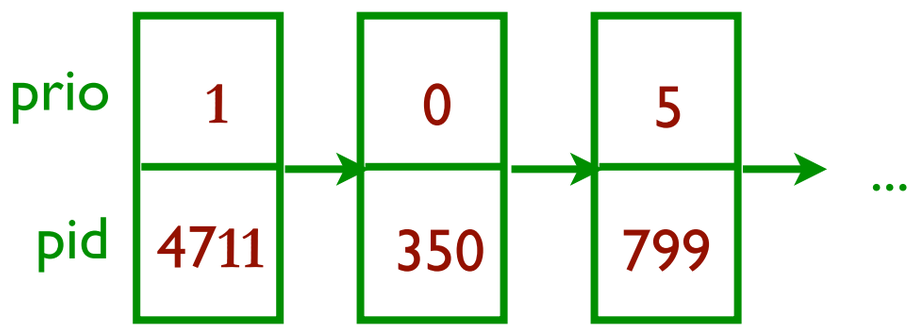
Organisation von Sleep-Queue und Run-Queue (1)

- **1. Versuch:**
  Könnten Listen sein, die bei wakeup() bzw. Prozessumschaltung (swtch()) sequentiell durchsucht werden.

Sleep-Queue



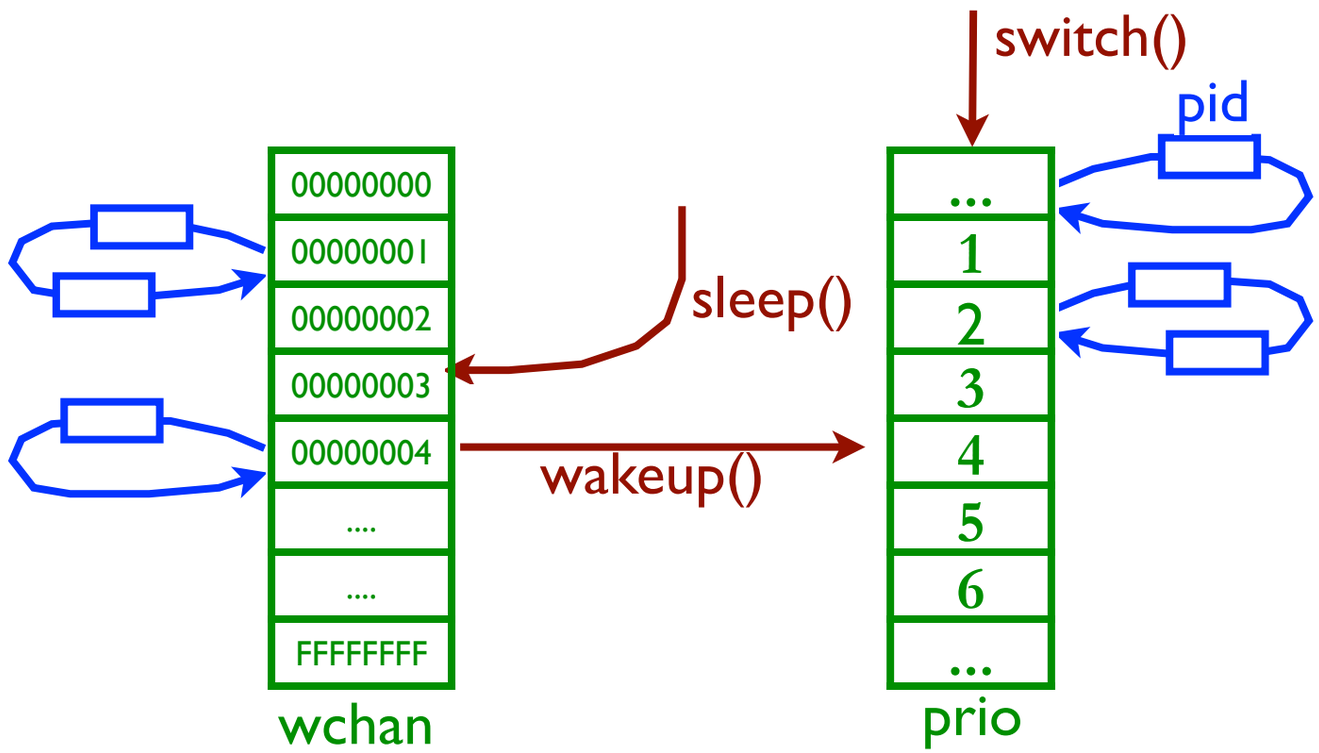
Run-Queue



⇒ dauert in der Regel zu lange

Organisation von Sleep-Queue und Run-Queue (2)

- **2. Versuch:** Stattdessen jeweils Array von Listen?
  - Sleep-Queue wird über wchan indiziert
  - Run-Queue wird über Priorität indiziert (vereinfacht)

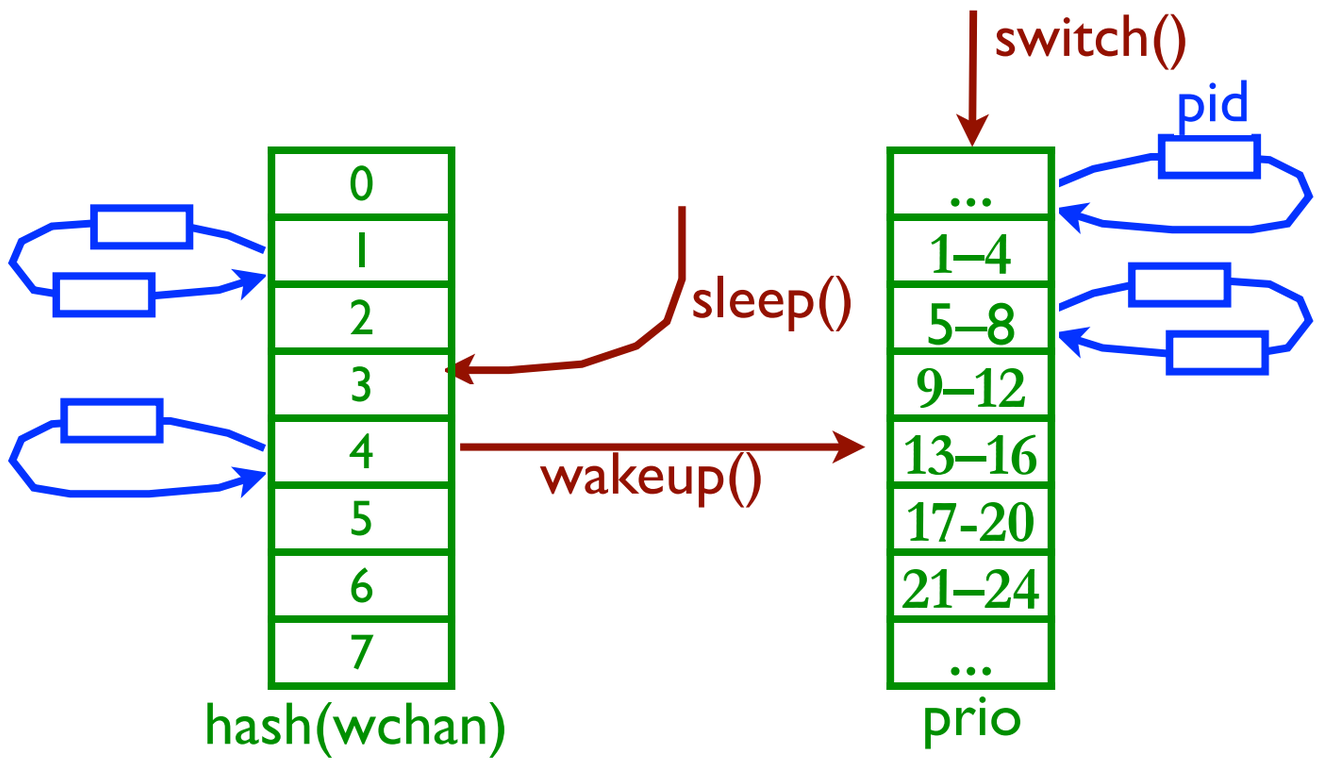


Sleep-Queue Run-Queue

⇒ Aber Arrays wären dann viel zu lang…

Organisation von Sleep-Queue und Run-Queue (3)

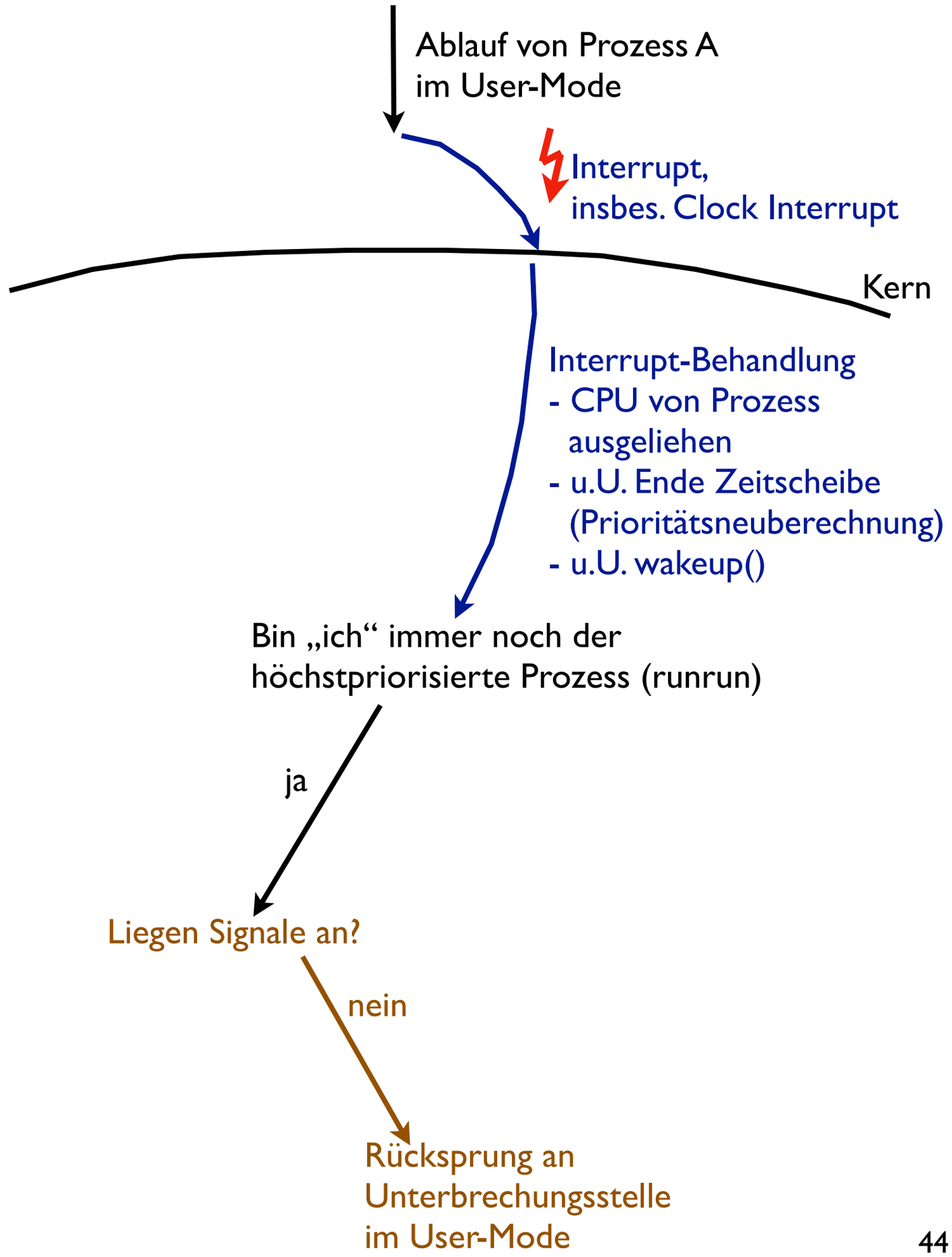
- **3. Versuch:** Zwar tatsächlich Array von Listen, aber:
  - Sleep-Queue wird über Hash-Funktion über wchan indiziert
  - Run-Queue wird über Gruppen von Prioritäten indiziert



⇒ Dann in Teillisten ggf. noch richtige Einträge suchen…

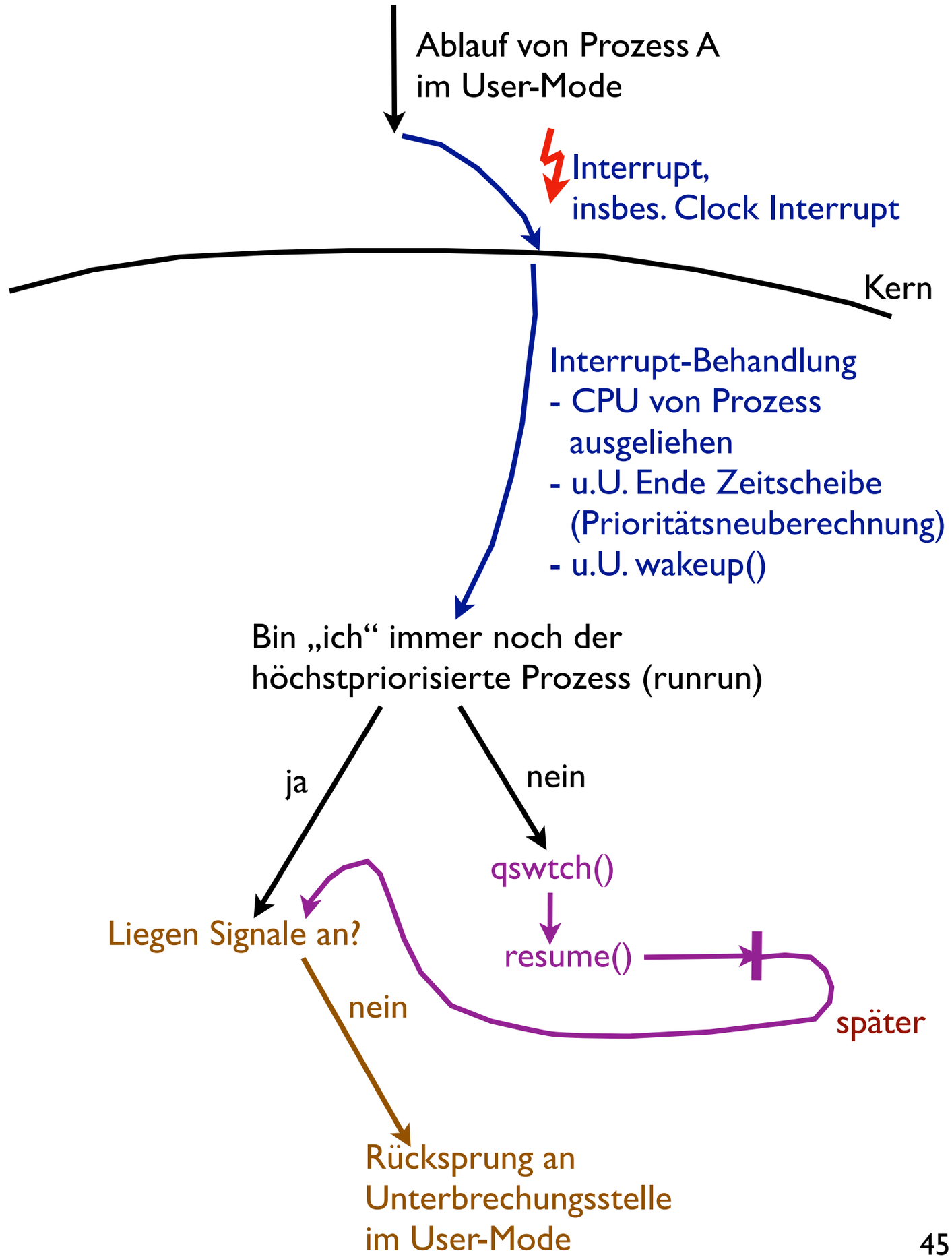
Einordnung einer Prozessumschaltung im Prozessablauf

(vereinfacht)



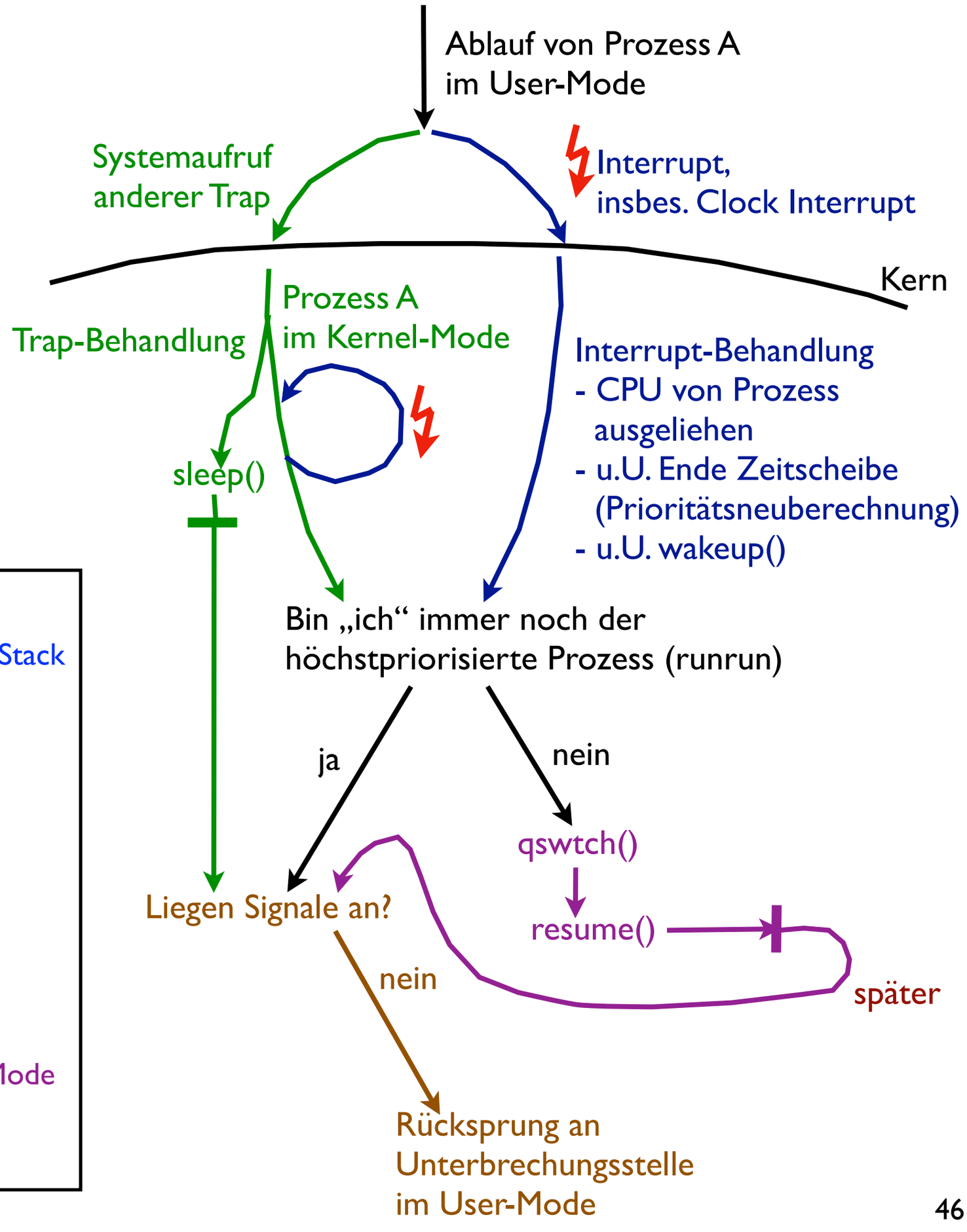
Einordnung einer Prozessumschaltung im Prozessablauf

(vereinfacht)



Einordnung einer Prozessumschaltung im Prozessablauf

(vereinfacht)



Globale Systemaufrufroutine (kernintern)

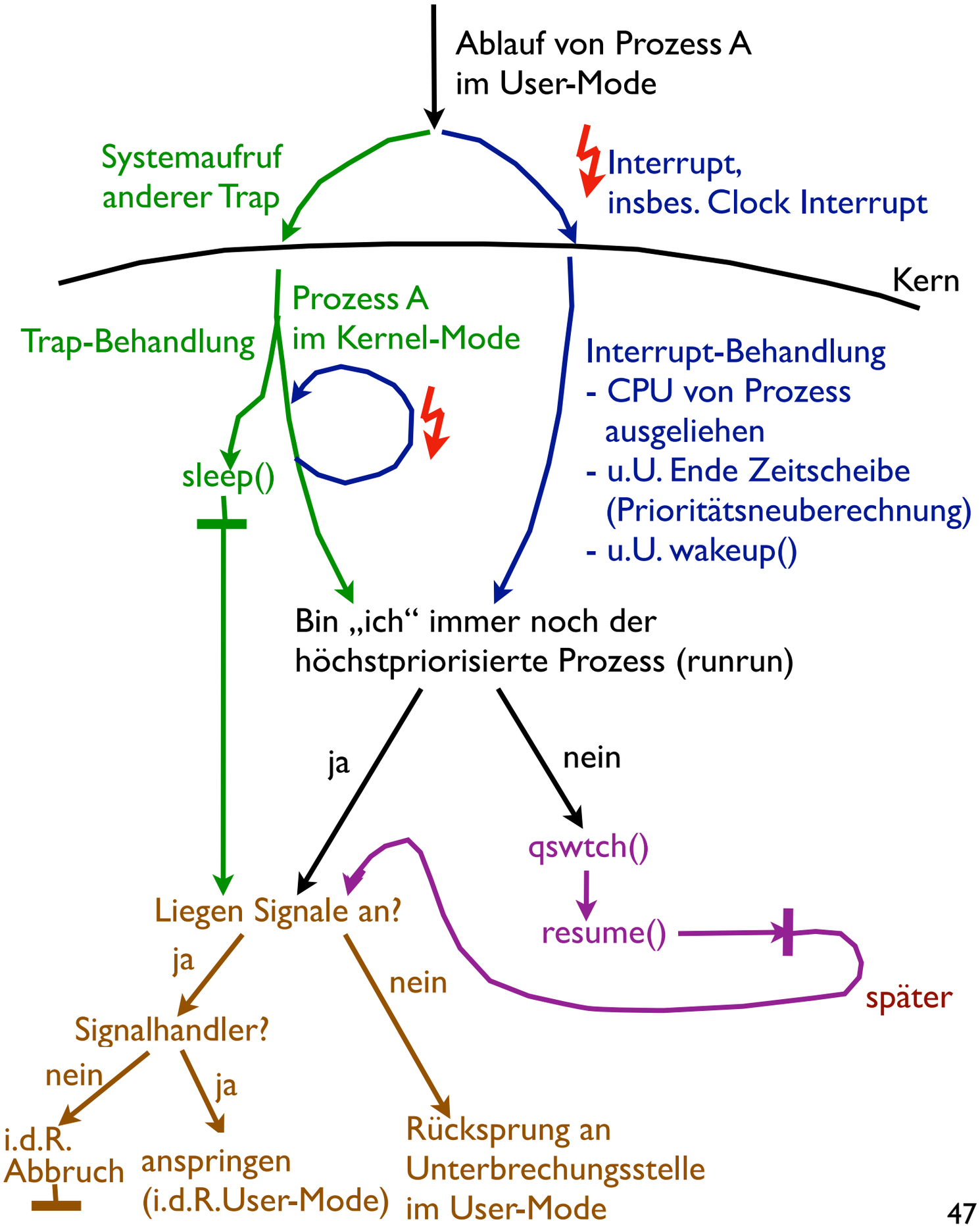
- Retten des User-Stack-Pointers auf den Kernel-Stack
- Retten der Register auf den Kernel-Stack
  (⇒ Kontext retten)
- Kopieren der Systemaufrufparameter vom User-Stack (fd, buf, len)
- Aufruf der kerninternen Routine für write()

  ...Arbeit ...
- Rückgabewert in Register
- (ggf. Prozessumschaltung) ←
- (ggf. Signalauslieferung)
- Kernel-Stack aufräumen
- Zurück in Bibliotheksroutine write() ⇒ User-Mode

  ⇒ dort User-Stack aufräumen

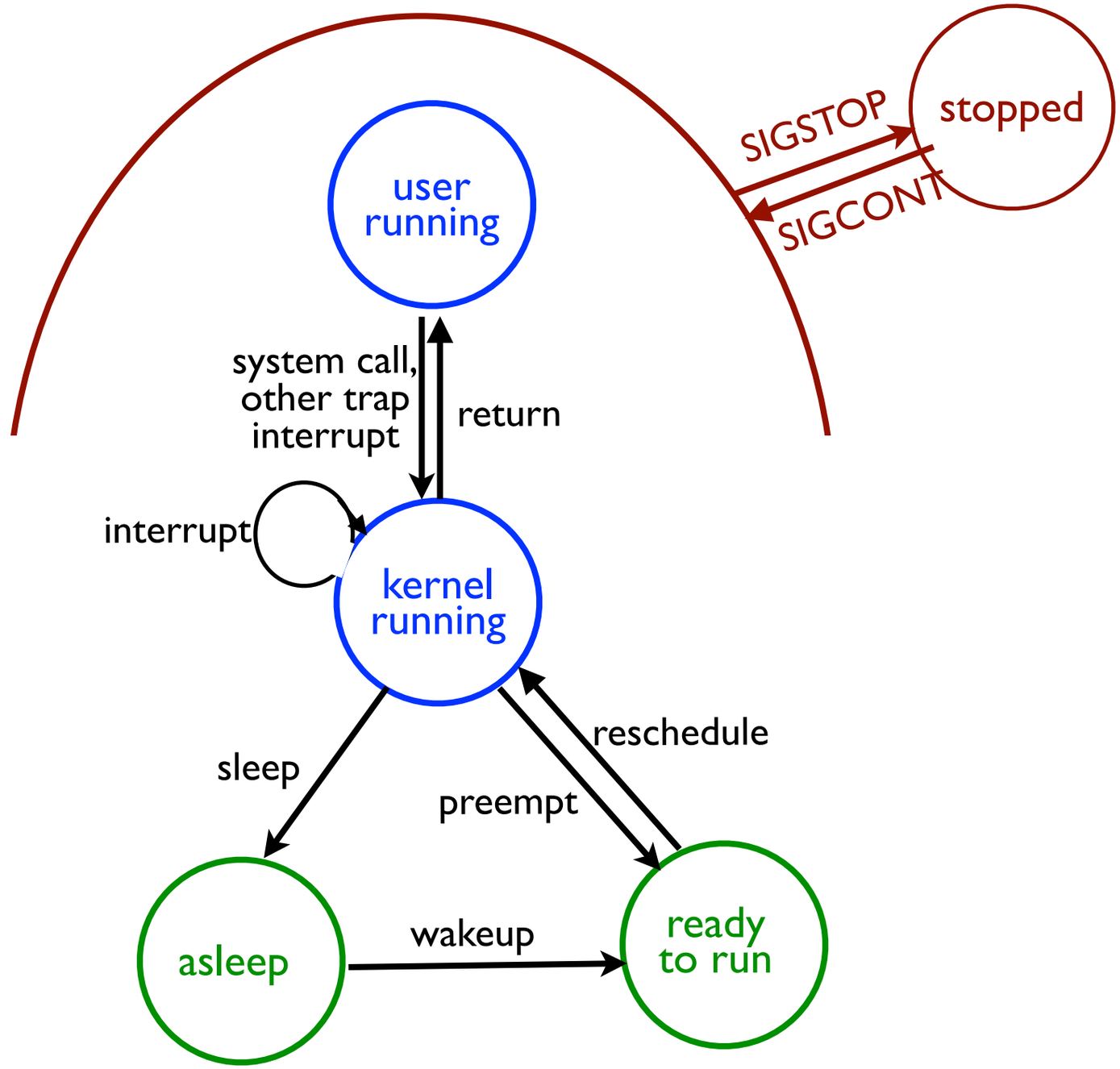
Einordnung einer Prozessumschaltung im Prozessablauf

(vereinfacht)



Prozesszustände

(vereinfacht)



Fragen – Teil 3

- Warum bestehen die *Sleep*- und die *Run-Queue* in Unix nicht aus jeweils einer einzigen Warteschlange? Wie sind sie stattdessen organisiert?
- Beschreibe kurz einige Zustände, in denen sich ein (Unix-)Prozess befinden kann.

Teil 4:
Datenstrukturen zur Prozessverwaltung

06 Prozessverwaltung (1), Ute Bormann

Prozessverwaltung in Unix

Verschiedene Teilaufgaben:

④ • Prozesserzeugung und -termination

② • Prozesswechsel (Scheduling)

⑤ • Nebenläufigkeit

⑥ • Interprozesskommunikation (Spezialform: Signale)

①

⇒③ • Datenstrukturen zur Prozessverwaltung

Informationen über Prozesse

Bezeichner:	pid, parent pid, user id, (process group)
Prozesszustand:	Prozesszustand: z.B. SRUN, SSLEEP, SZOMB, ...
Ggf. Warten auf Ereignis:	wchan
Verkettungen:	Verweise auf Vater-, Geschwisterprozesse
	Sleep-Queue-/Run-Queue-Verkettung
	3 Listen: normale Prozesse, Zombies, freie Einträge
Flags	z.B. SSYS: Systemprozess; SLOAD: Prozess im Hauptspeicher
Scheduling	nice: Benutzerdefinierte Priorität
	CPU-Nutzung
	➔ Aktuelle Priorität (Ermittelter oder bei sleep() angegebener Wert)
Signale	Auszuliefernde Signale
	Zu ignorierende Signale
	Zu behandelnde Signale
Speicherverwaltungsinfos (später)	Für's paging (z.B. Verweis auf Pagetable)
	Für's swapping (z.B. wohin)
Accounting-/Quota-Infos	Verweis auf diese Infos

Informationen über Prozesse

Bezeichner:	Zugriffsrechte: Benutzer, Gruppe; Defaultwerte
Prozesszustand:	Geöffnete Dateien:
Ggf. Warten auf Ereignis:	Aktuelles Verzeichnis:
Verkettungen:	Platz zum Retten des Prozesszustands (z.B. bei sleep())
Flags	
Scheduling	
Signale	
Speicherverwaltungsinfos (später)	
Accounting-/Quota-Infos	

Proc-Struktur: ➔ immer verfügbar für alle Prozesse	User-Struktur ➔ nur für laufenden Prozess verfügbar
Bezeichner:	Zugriffsrechte: Benutzer, Gruppe; Defaultwerte
Prozesszustand:	Geöffnete Dateien:
Ggf. Warten auf Ereignis:	Aktuelles Verzeichnis:
Verkettungen:	Platz zum Retten des Prozesszustands (z.B. bei sleep())
Flags	
Scheduling	
Signale	
Speicherverwaltungsinfos (später)	
Accounting-/Quota-Infos	

Proc-Struktur:

➔ immer verfügbar für alle Prozesse

User-Struktur

➔ nur für laufenden Prozess verfügbar

Bezeichner:	Zugriffsrechte: Benutzer, Gruppe; Defaultwerte
Prozesszustand:	Geöffnete Dateien:
Ggf. Warten auf Ereignis:	Aktuelles Verzeichnis:
Verkettungen:	Platz zum Retten des Prozesszustands (z.B. bei sleep())
Flags	
Scheduling	
Signale →	Wie sollen Signale behandelt werden?
Speicherverwaltungsinfos (später) →	Ablagebereich auf Hintergrundspeicher
Accounting-/Quota-Infos	

Für Zombies

Stack
Daten
Text
Prozess A
Prozess
Systemaufruf
Prozess
user-area von A
user-area
User-
Struktur
Kernel--
Stack
nur im
Kernel-Mode
zugreifbar
user-area
Proc-Struktur von A
frei
frei
freie Einträge
Kern
Proc-Tabelle

Stack
Daten
Text
Prozess A
Systemaufruf
Prozess
Prozess
user-area von A
user-area
user-area
User-Struktur
Kernel--Stack
nur im Kernel-Mode zugreifbar
hash(wchan)
0
1
2
3
4
5
6
7
Sleep-Queue
(Prozess wartet auf 3)
Proc-Struktur von A
(Prozess wartet auf 3)
(Prozess wartet auf 6)
(Prozess wartet auf 6)
(Prozess wartet auf 6)
frei
frei
freie Einträge
Proc-Tabelle
Kern

Fragen – Teil 4

- Warum werden die Zustandsinformationen eines Unix-Prozesses teilweise in der *Proc-Struktur* und teilweise in der *User-Struktur* abgelegt? Nenne jeweils zwei charakteristische Beispiele für Angaben darin.

Zusammenfassung

- Prozessumschaltung
- Scheduling-Verfahren
- Sleep()/Wakeup()
- Proc-Struktur vs. User-Struktur

Prozessverwaltung 1 – Fragen

1. Nenne verschiedene Gründe für eine Prozessumschaltung.
2. Nenne einige Randbedingungen, auf die man beim Entwurf eines *Schedulers* achten sollte. Wie sollten rechenintensive bzw. Ein-/Ausgabe-intensive Prozesse dabei behandelt werden?
3. Wie könnte man mit Hilfe eines *Round-Robin-Schedulers* Prozessprioritäten „simulieren“?
4. Warum bestehen die *Sleep-* und die *Run-Queue* in Unix nicht aus jeweils einer einzigen Warteschlange? Wie sind sie stattdessen organisiert?
5. Beschreibe kurz einige Zustände, in denen sich ein (Unix-)Prozess befinden kann.
6. Warum werden die Zustandsinformationen eines Unix-Prozesses teilweise in der *Proc-Struktur* und teilweise in der *User-Struktur* abgelegt? Nenne jeweils zwei charakteristische Beispiele für Angaben darin.