Traps vs. Interrupts vs. Signale

Ute Bormann, TI2

2023-10-13

Inhalt

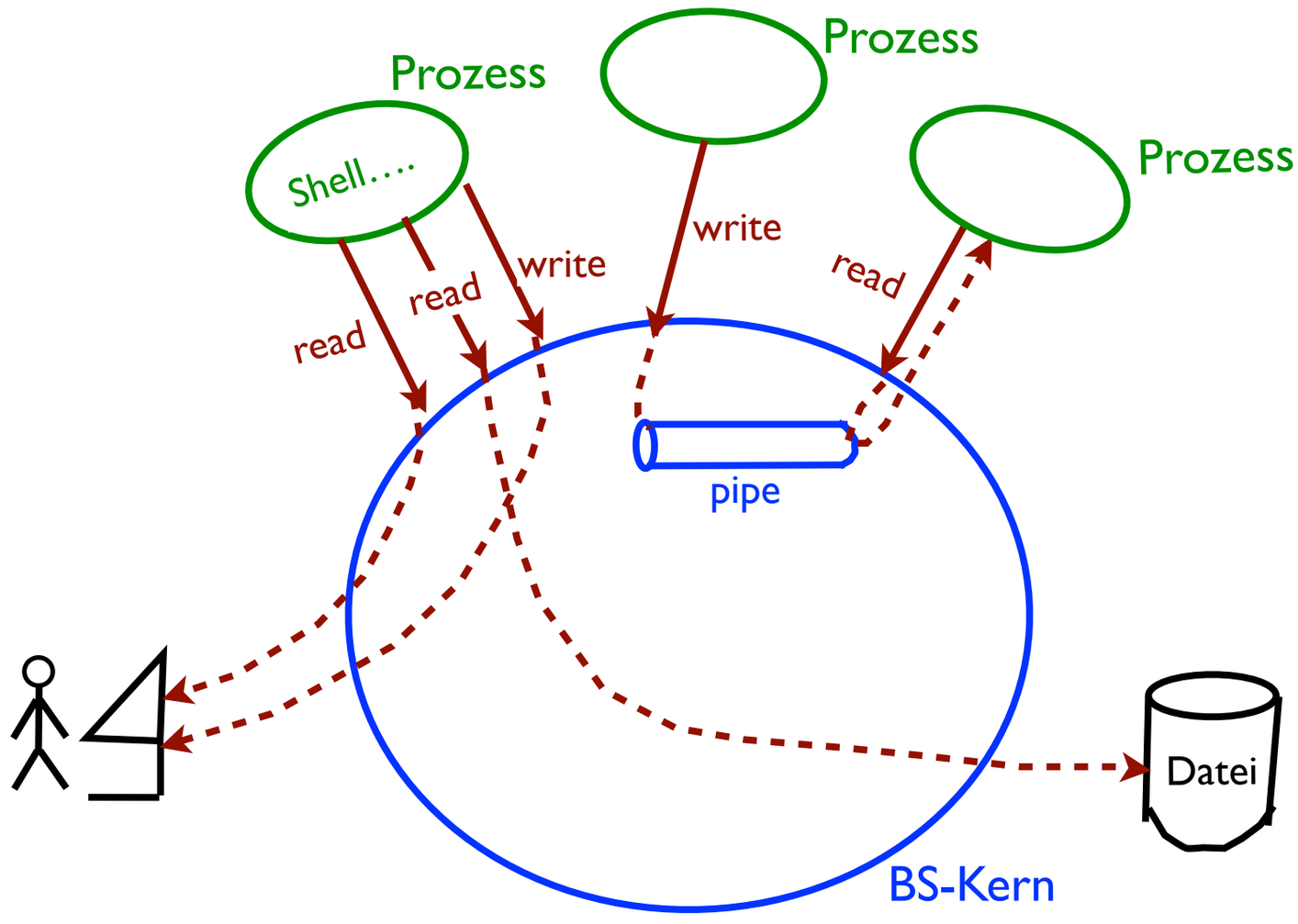
1. User Mode vs. Kernel Mode
2. Traps
3. Interrupts
4. Betriebssystemkomponenten
5. Signale

Teil 1:
User Mode vs. Kernel Mode

Bisher:

- Arbeiten mit dem System (Nutzersicht)
  - Dateisystem
  - Prozesse

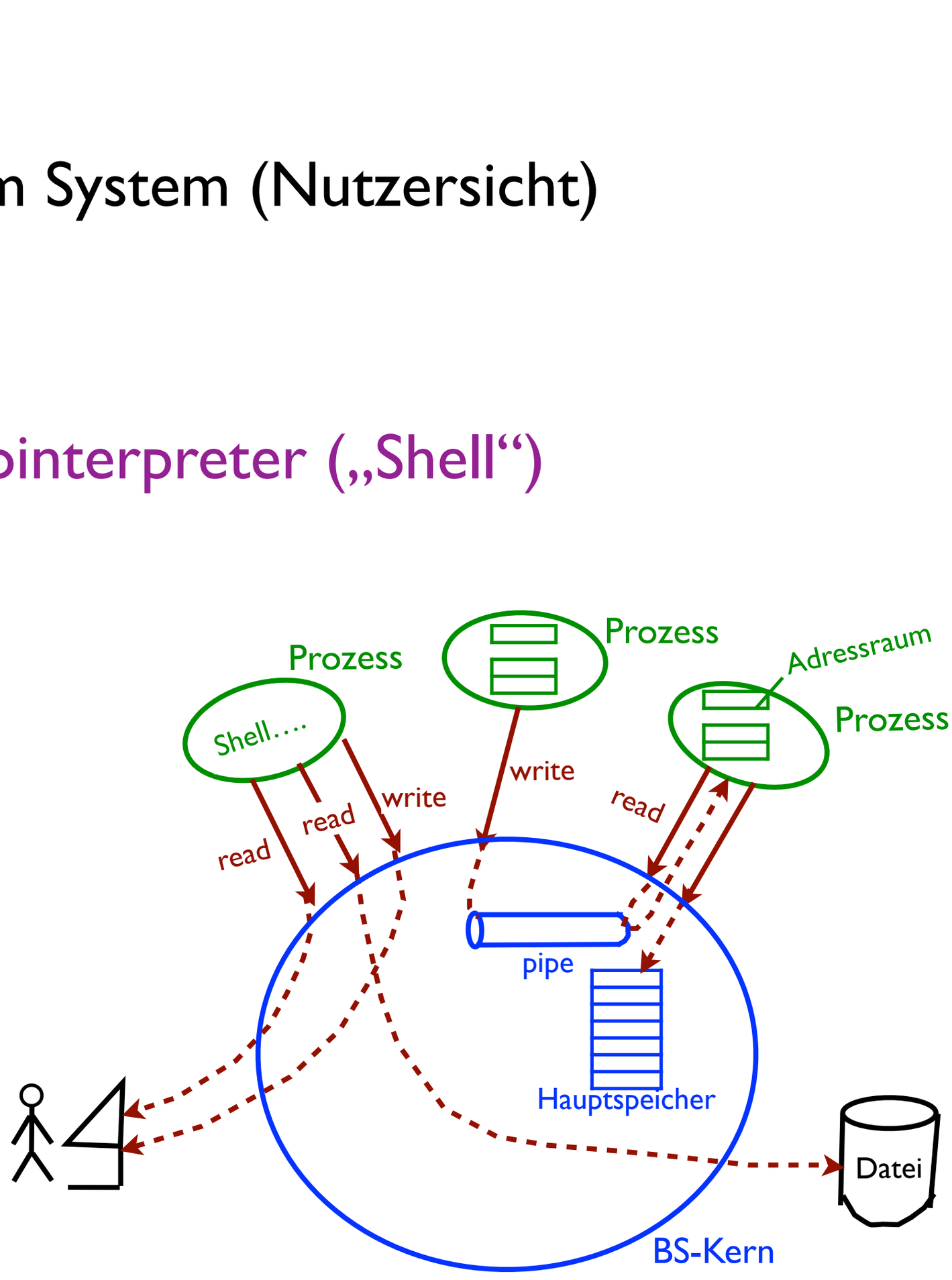
    ⇒ Kommandointerpreter („Shell“)



Bisher:

- Arbeiten mit dem System (Nutzersicht)
  - Dateisystem
  - Prozesse

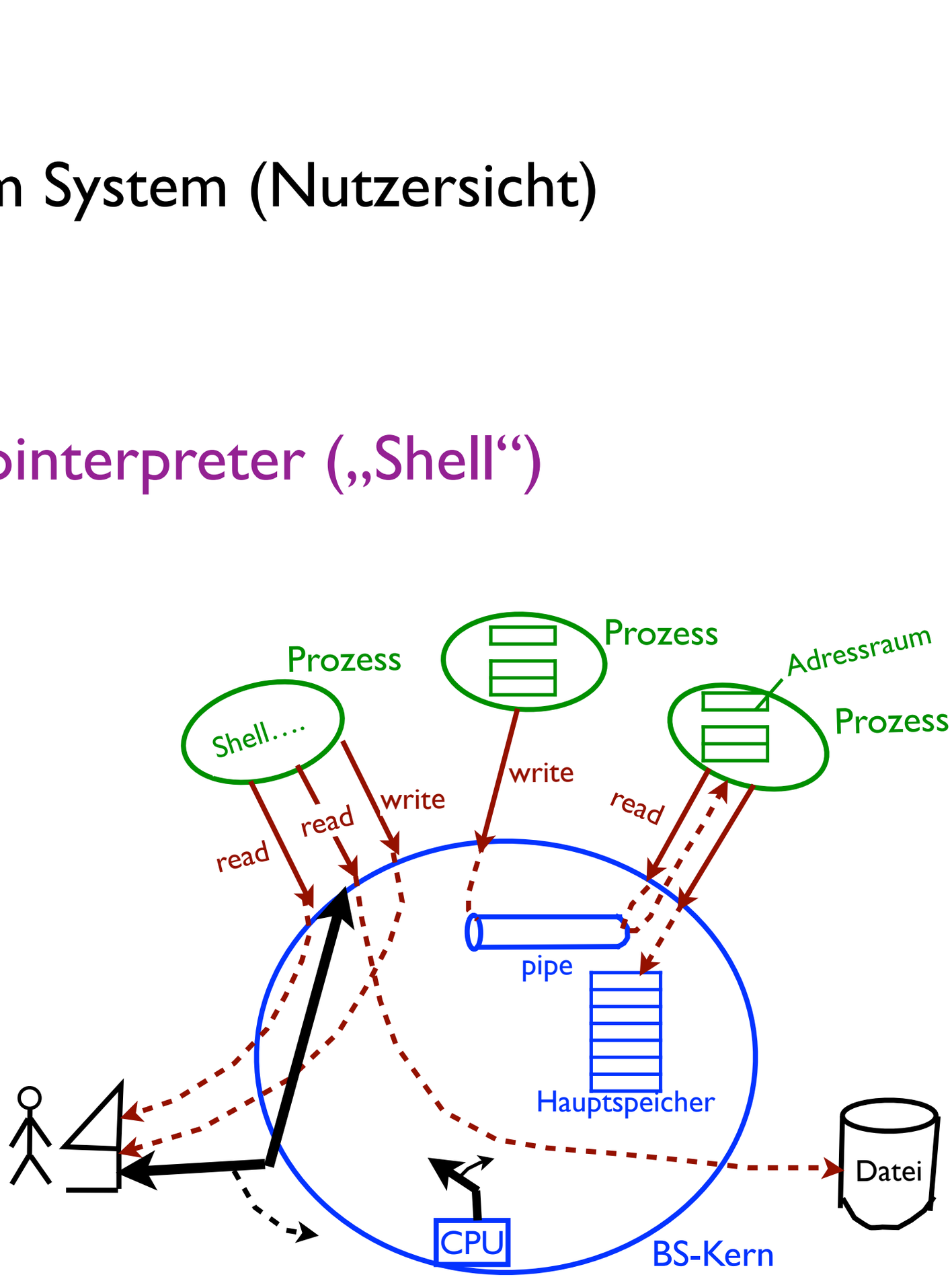
    ⇒ Kommandointerpreter („Shell“)



Bisher:

- Arbeiten mit dem System (Nutzersicht)
  - Dateisystem
  - Prozesse

    ⇒ Kommandointerpreter („Shell“)



Aktivierung des Betriebssystemkerns

- Prozess arbeitet (Anwendungs)Programm ab

  ⇒ eigener Adressraum (Text, Daten, Stack)

- Auftrag an Betriebssystem (z.B. write())

  ⇒ spezieller „Unterprogrammaufruf“ (Systemaufruf)

  ⇒ Abarbeiten von Betriebssystemcode

Exkurs: Verschiedene Begriffe

Deutsch
Prozedur
Unterprogramm
Funktion
Methode
Systemaufruf

Exkurs: Verschiedene Begriffe

Deutsch	Englisch	Kontext
Prozedur	Procedure	(generisch)
Unterprogramm	Subroutine	Fortran-Begriff; aber auch „Bibliotheksroutine“
Funktion	Function	C/C++-Begriff für globale Prozeduren
Methode	Method	Objektorientierte Programmierung: klassenspezifische Prozeduren
Systemaufruf	System Call	Aufruf einer Betriebssystem-Prozedur

Aktivierung des Betriebssystemkerns

- Prozess arbeitet (Anwendungs)Programm ab

  ⇒ eigener Adressraum (Text, Daten, Stack)

- Auftrag an Betriebssystem (z.B. write())

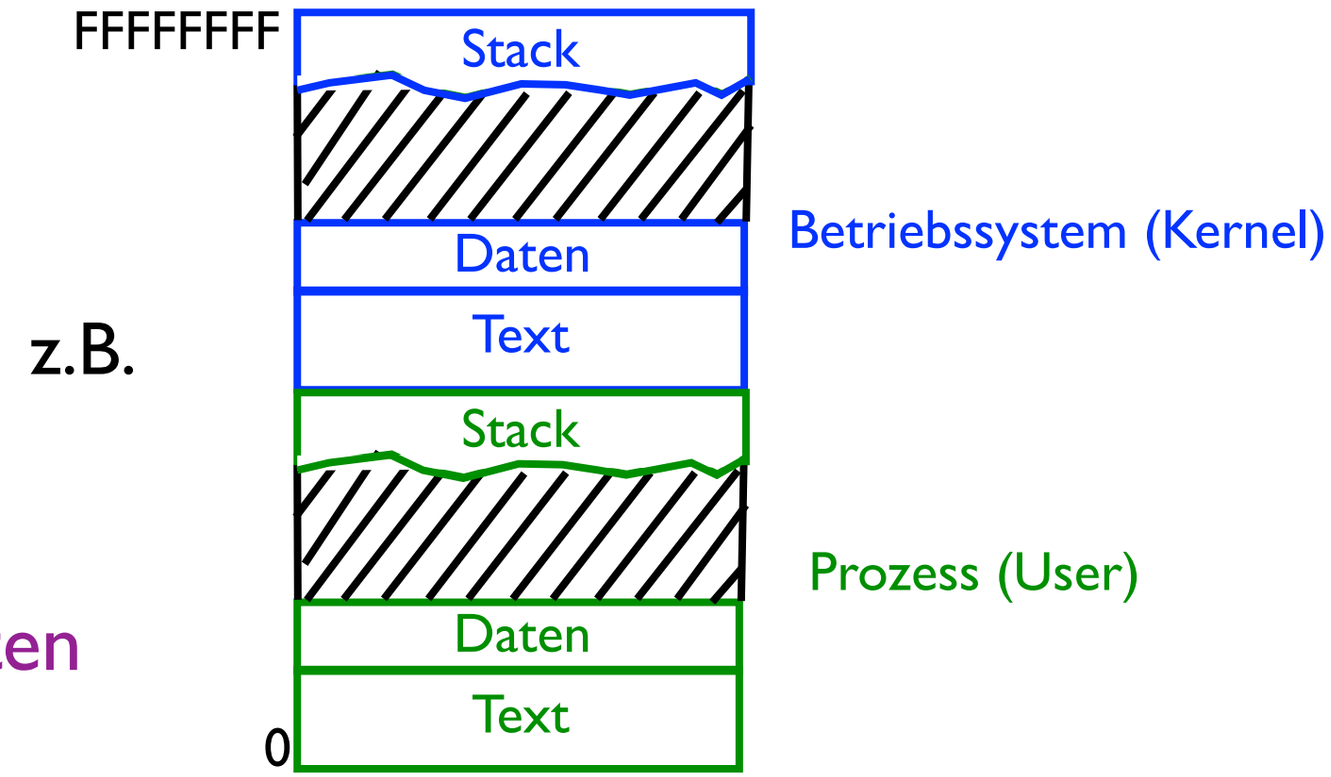
  ⇒ spezieller „Unterprogrammaufruf“ (Systemaufruf)

  ⇒ Abarbeiten von Betriebssystemcode

- Betriebssystemcode benötigt ebenfalls Adressraum (Text, Daten, Stack)

  ⇒ muss abgegrenzt sein

  ⇒ Reservieren eines bestimmten Adressraums für Kern



- Eigener Adressraum reicht zur Abschottung des Kerns nicht aus:

  Betriebssystem-Code ermöglicht direkten Zugriff auf Geräte, Dateien,...

  ⇒ sollte aus Anwendungsprogramm heraus nicht einfach angesprungen und ausgeführt werden können

⇒ Zugriffsschutz, z.B.

  - kein unautorisiertes Lesen von Dateien
  - kein (versehentliches) Löschen von BS-Daten, etc.

⇒ Systemaufrufe sind keine normalen Prozeduraufrufe

Kernel Mode vs. User Mode

- Systemaufruf führt zum Betreten eines privilegierten Arbeitsmodus (hardwareunterstützt)

  ⇒ Kernel-Mode (im Gegensatz zum User-Mode)

- Nur im Kernel-Mode ist Kernadressraum zugreifbar

  ⇒ Zugriff nur über definierte Schnittstelle

- Im Kernel-Mode gibt es privilegierte Maschineninstruktionen

  ⇒ im User-Mode nicht verfügbar

Fragen – Teil 1

- Worin unterscheidet sich der *Kernel-Mode* vom *User-Mode* (in Unix)? Warum wird diese Unterscheidung getroffen?

Teil 2:
Traps

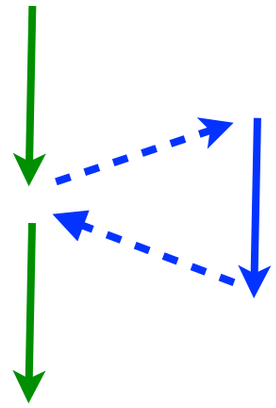
05 Traps vs. Interrupts vs. Signale, Ute Bormann

- Bei Systemaufruf wird die normale Abarbeitung des Anwendungsprogramms „ausgesetzt“

  ⇒ aus prozessinternen Gründen

  ⇒ Trap

  (manchmal Software-Interrupt genannt)
  (hier: gewollt ⇒ Trapinstruktion)

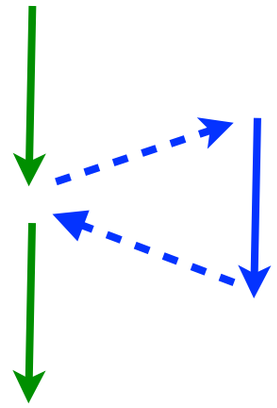


- Bei Systemaufruf wird die normale Abarbeitung des Anwendungsprogramms „ausgesetzt“

  ⇒ aus prozessinternen Gründen

  ⇒ Trap

  (manchmal Software-Interrupt genannt)
  (hier: gewollt ⇒ Trapinstruktion)



- Andere Ursachen für Traps sind in der Regel ungewollt, z.B.
  - Division durch 0
  - Zugriff auf eine illegale Adresse

  ⇒ führt i.d.R. zum Programmabbruch

  - Zugriff auf Informationen, die z.Zt. nicht im Speicher stehen

  ⇒ müssen nachgeladen werden (Page Fault → später)

- Traps haben unterschiedliche Ursachen und Auswirkungen, führen aber gleichermaßen zum Betreten des Kerns

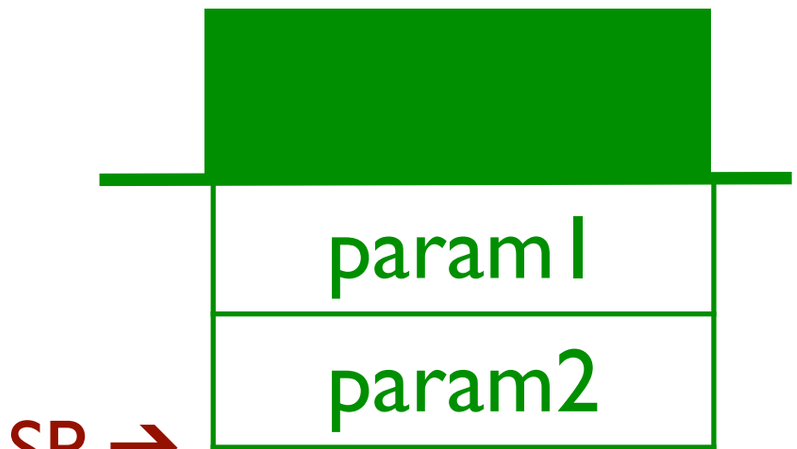
Behandlung von Systemaufrufen (und anderen Traps)

Ähnlich wie „normaler“ Prozeduraufruf:

Behandlung von Systemaufrufen (und anderen Traps)

Ähnlich wie „normaler“ Prozeduraufruf:

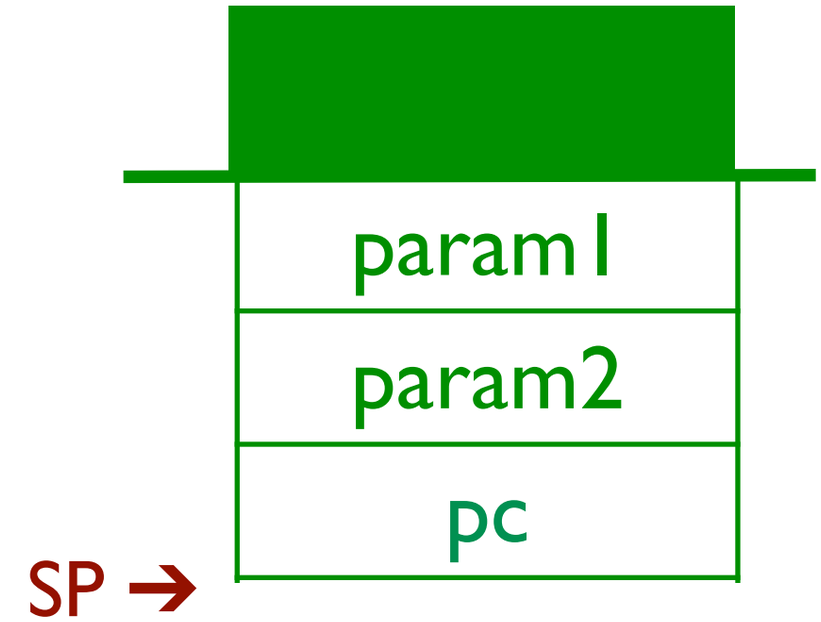
- Prozedurparameter auf den Stack schreiben (alternativ: Register)



Behandlung von Systemaufrufen (und anderen Traps)

Ähnlich wie „normaler“ Prozeduraufruf:

- Prozedurparameter auf den Stack schreiben (alternativ: Register)
- Aktuellen Kontext verlassen
  ⇒ pc auf den Stack retten (+ evtl. Register)
- Prozedur anspringen

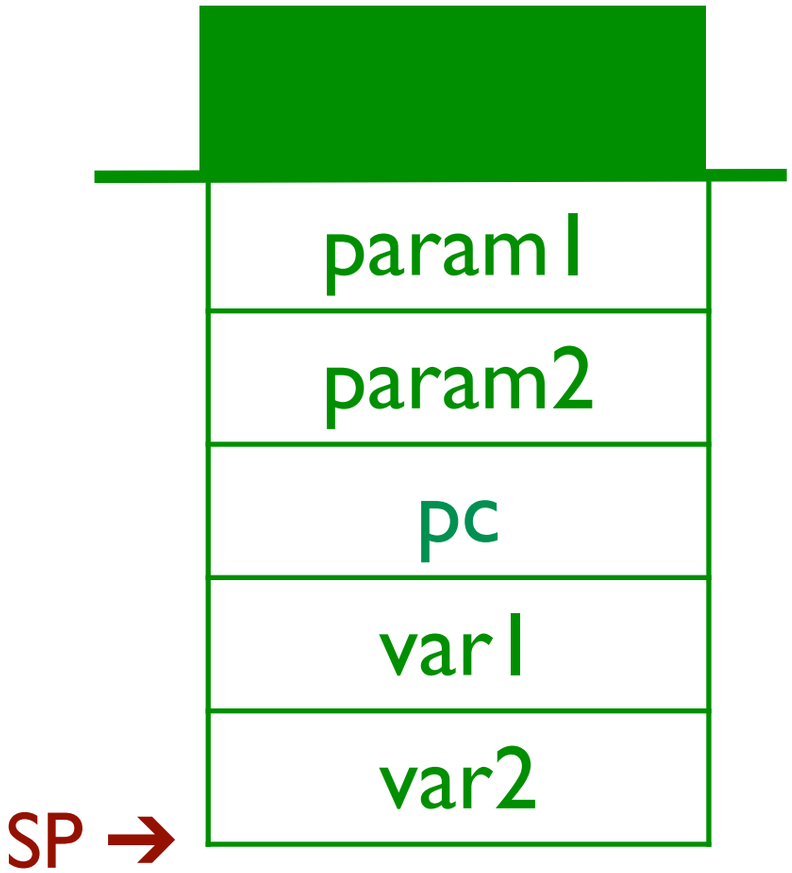


Behandlung von Systemaufrufen (und anderen Traps)

Ähnlich wie „normaler“ Prozeduraufruf:

- Prozedurparameter auf den Stack schreiben (alternativ: Register)
- Aktuellen Kontext verlassen
  ⇒ pc auf den Stack retten (+ evtl. Register)
- Prozedur anspringen
- Ggf. lokale Variablen auf den Stack schreiben (alternativ: Register)

  ...Arbeit ...

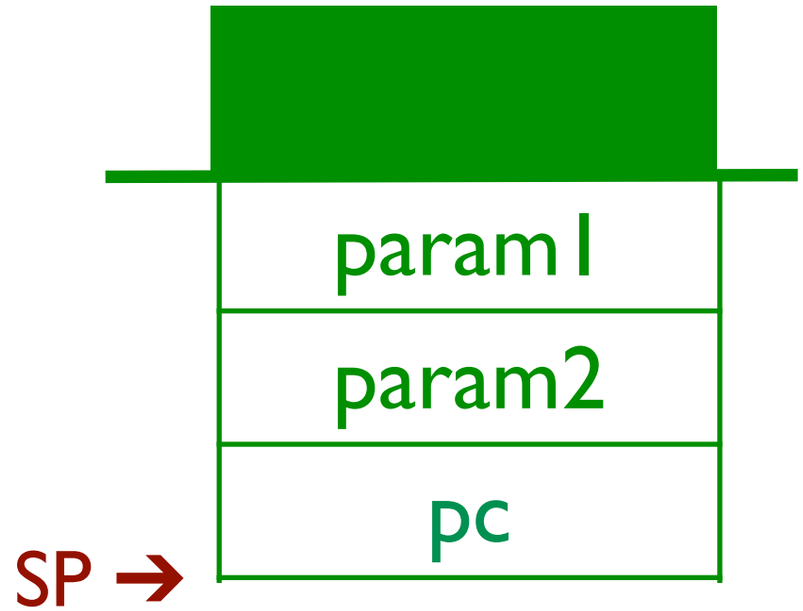


Behandlung von Systemaufrufen (und anderen Traps)

Ähnlich wie „normaler“ Prozeduraufruf:

- Prozedurparameter auf den Stack schreiben (alternativ: Register)
- Aktuellen Kontext verlassen
  ⇒ pc auf den Stack retten (+ evtl. Register)
- Prozedur anspringen
- Ggf. lokale Variablen auf den Stack schreiben (alternativ: Register)

  ...Arbeit ...
- Rückgabewert ermitteln (häufig in Register)
- Stack aufräumen (lokale Variablen weg)

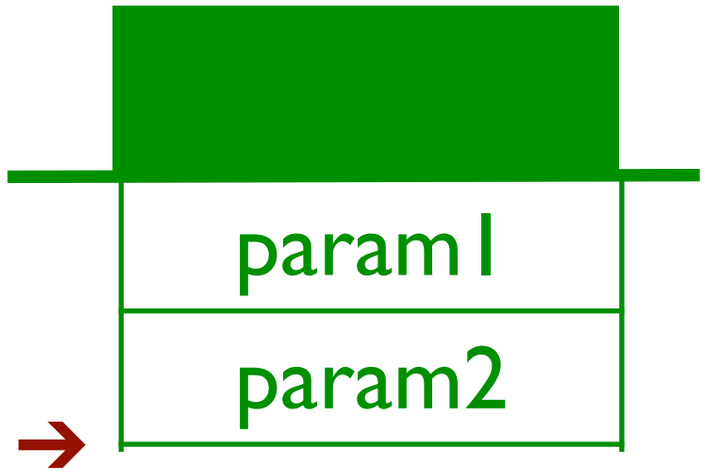


Behandlung von Systemaufrufen (und anderen Traps)

Ähnlich wie „normaler“ Prozeduraufruf:

- Prozedurparameter auf den Stack schreiben (alternativ: Register)
- Aktuellen Kontext verlassen
  ⇒ pc auf den Stack retten (+ evtl. Register)
- Prozedur anspringen
- Ggf. lokale Variablen auf den Stack schreiben (alternativ: Register)

  ...Arbeit ...
- Rückgabewert ermitteln (häufig in Register)
- Stack aufräumen (lokale Variablen weg)
- An Aufrufstelle zurückkehren
  (Adresse vom Stack in pc laden)



Behandlung von Systemaufrufen (und anderen Traps)

Ähnlich wie „normaler“ Prozeduraufruf:

- Prozedurparameter auf den Stack schreiben (alternativ: Register)
- Aktuellen Kontext verlassen
  ⇒ pc auf den Stack retten (+ evtl. Register)
- Prozedur anspringen
- Ggf. lokale Variablen auf den Stack schreiben (alternativ: Register)
  ...Arbeit ...
- Rückgabewert ermitteln (häufig in Register)
- Stack aufräumen (lokale Variablen weg)
- An Aufrufstelle zurückkehren
  (Adresse vom Stack in pc laden)
- Stack aufräumen (Parameter weg)

Systemaufruf: Beispiel: write (fd, buf, len) ⇒ 3 Parameter

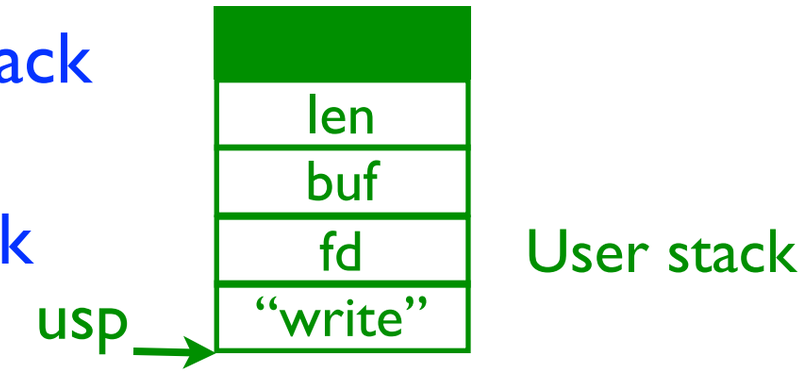
- Besonderheiten:
  - Auslösen eines gewollten Traps
  - Umschalten in den Kernel-Mode

Systemaufruf: Beispiel: write (fd, buf, len) ⇒ 3 Parameter

- Besonderheiten:
  - Auslösen eines gewollten Traps
  - Umschalten in den Kernel-Mode
- write() ist Bibliotheksroutine
  ⇒ erstmal normaler Prozeduraufruf

Nach Ansprung von write():

- Schreiben der Systemaufrufparameter auf den User-Stack (fd, buf, len)
- Schreiben der Systemaufrufnummer auf den User-Stack (write=4)

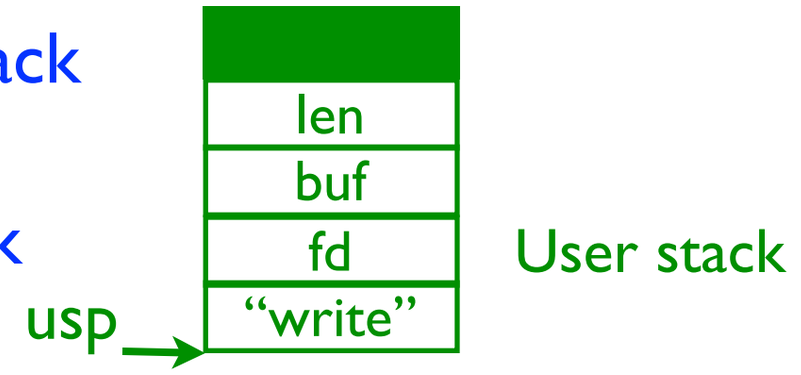


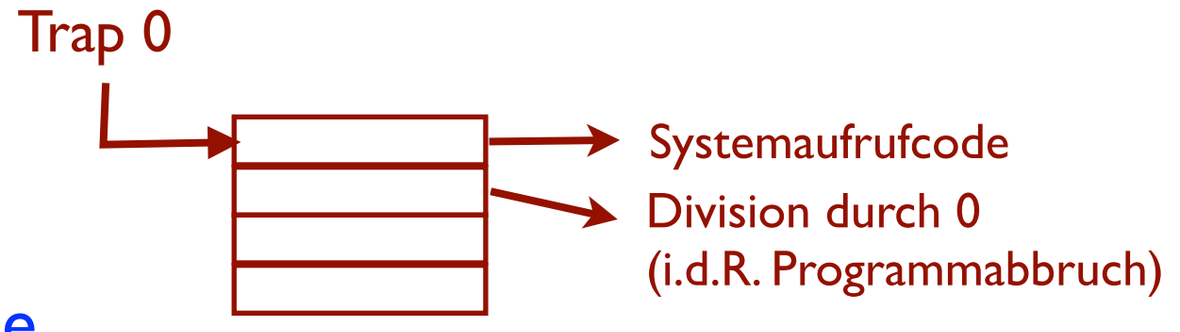
Systemaufruf: Beispiel: write (fd, buf, len) ⇒ 3 Parameter

- Besonderheiten:
  - Auslösen eines gewollten Traps
  - Umschalten in den Kernel-Mode
- write() ist Bibliotheksroutine
  ⇒ erstmal normaler Prozeduraufruf

Nach Ansprung von write():

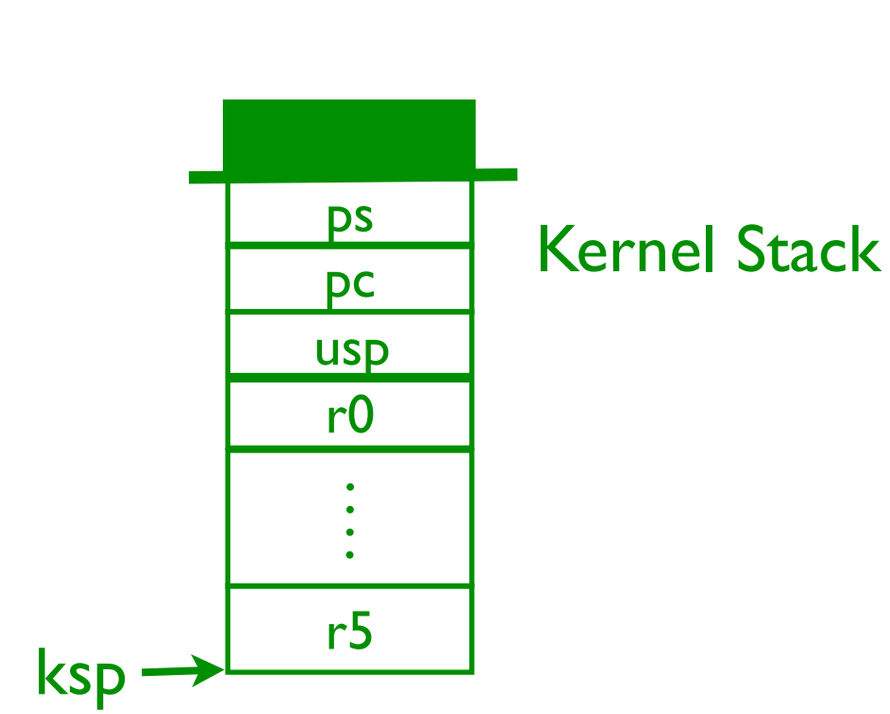
- Schreiben der Systemaufrufparameter auf den User-Stack (fd, buf, len)
- Schreiben der Systemaufrufnummer auf den User-Stack (write=4)
- Auslösen von Trap 0 (= Trap-Instruktion)
  ⇒ hardwarebasiert. Dabei:
  - Umschalten in den Kernel-Mode
  - Retten von pc und ps auf Kernel-Stack
  - Indizieren der kerninternen Trap-Tabelle
    ⇒ Eintrag 0 verweist auf Systemaufrufcode (Traphandler)





Globale Systemaufrufroutine (kernintern)

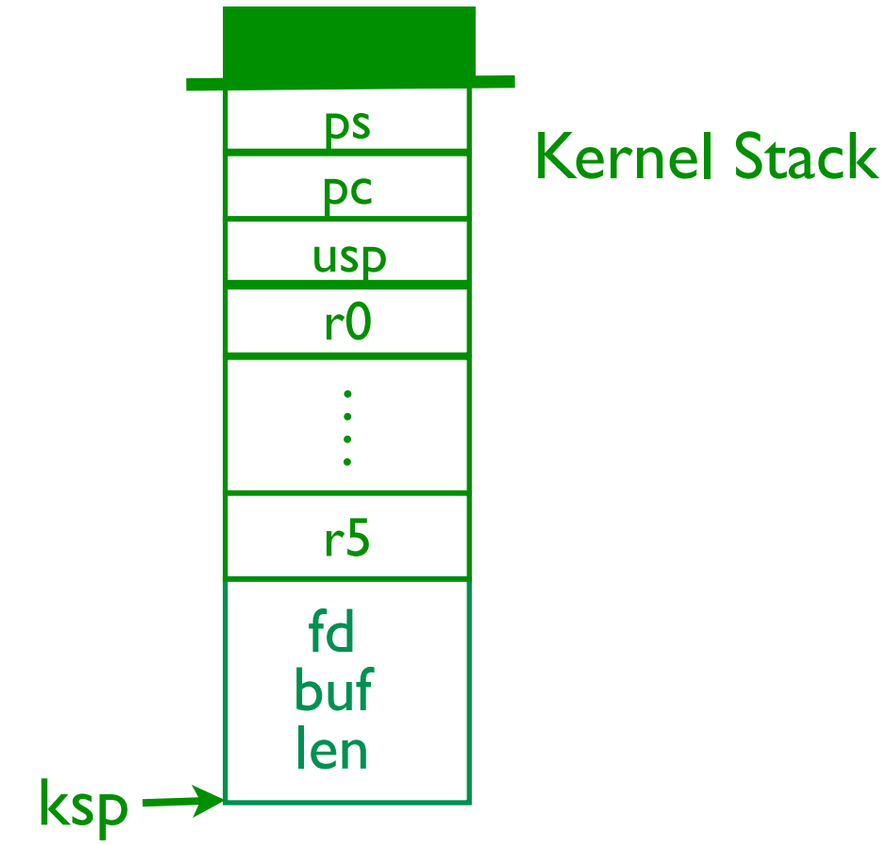
- Retten des User-Stack-Pointers (usp) auf den Kernel-Stack
- Retten der Register auf den Kernel-Stack (⇒ Kontext retten)



Globale Systemaufrufroutine (kernintern)

- Retten des User-Stack-Pointers (usp) 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 ...

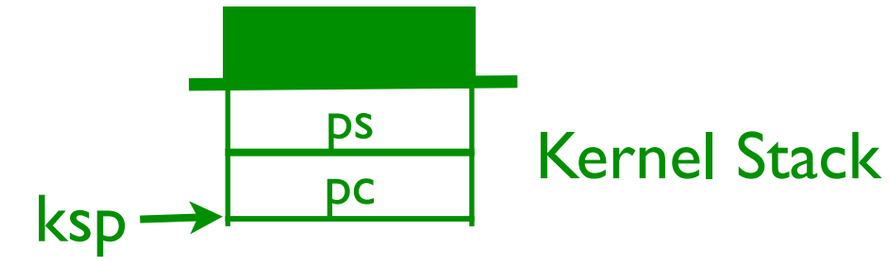


Globale Systemaufrufroutine (kernintern)

- Retten des User-Stack-Pointers (usp) 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



- Kernel-Stack aufräumen

Globale Systemaufrufroutine (kernintern)

- Retten des User-Stack-Pointers (usp) 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

- Kernel-Stack aufräumen
- Zurück in Bibliotheksroutine write() ⇒ User-Mode

  ⇒ dort User-Stack aufräumen + zurück ins Anwendungsprogramm

Globale Systemaufrufroutine (kernintern)

- Retten des User-Stack-Pointers (usp) 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 → später)
- (ggf. Signalauslieferung → später)
- Kernel-Stack aufräumen
- Zurück in Bibliotheksroutine write() ⇒ User-Mode

⇒ dort User-Stack aufräumen + zurück ins Anwendungsprogramm

Fragen – Teil 2

- Was passiert in etwa bei einem *Systemaufruf*?
  (Reihenfolge der Arbeitsschritte.)

- Was ist ein *Trap*? Nenne Beispiele.

Teil 3:
Interrupts

05 Traps vs. Interrupts vs. Signale, Ute Bormann

Interrupts (Unterbrechungen)

- Traps führen zum (temporären) Verlassen des Programmablaufs aus prozessinternen Gründen
- Im System müssen i.d.R. auch noch andere, davon unabhängige Aktivitäten verwaltet werden. Beispiele:
  - Eingabe eines Zeichens auf einer Tastatur
  - Fertigmeldung eines Geräts (z.B. Schreiben eines Blocks auf eine Platte)
  - „Ticken“ der Systemuhr

⇒ haben i.d.R. nichts mit laufendem Prozess zu tun

⇒ (i.d.R.) prozessexterne Aktivitäten

Interrupts (Unterbrechungen)

- Traps führen zum (temporären) Verlassen des Programmablaufs aus prozessinternen Gründen
- Im System müssen i.d.R. auch noch andere, davon unabhängige Aktivitäten verwaltet werden. Beispiele:
  - Eingabe eines Zeichens auf einer Tastatur
  - Fertigmeldung eines Geräts (z.B. Schreiben eines Blocks auf eine Platte)
  - „Ticken“ der Systemuhr

  ⇒ haben i.d.R. nichts mit laufendem Prozess zu tun

  ⇒ (i.d.R.) prozessexterne Aktivitäten
- Benötigen temporär CPU zur Behandlung:
  - Retten des eingegebenen Zeichens
  - Absetzen eines neuen Auftrags an Platte
  - Weiterstellen der internen Uhrzeit

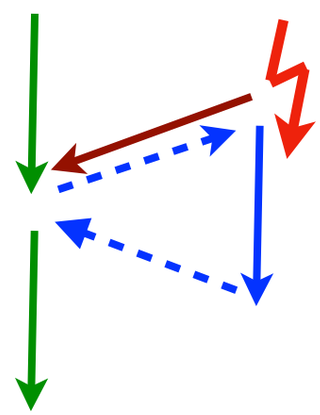
- Erfordern i.d.R. schnelle Behandlung:
  - Interne Uhr geht andernfalls mit der Zeit nach
  - Tastaturpuffer nur klein (Eingabe weiterer Zeichen führt zum Überlauf)
  - Platte braucht neuen Auftrag

⇒ Ausleihen der CPU vom laufenden Prozess

- Erfordern i.d.R. schnelle Behandlung:
  - Interne Uhr geht andernfalls mit der Zeit nach
  - Tastaturpuffer nur klein (Eingabe weiterer Zeichen führt zum Überlauf)
  - Platte braucht neuen Auftrag

⇒ Ausleihen der CPU vom laufenden Prozess

⇒ Prozessunterbrechung (Interrupt)

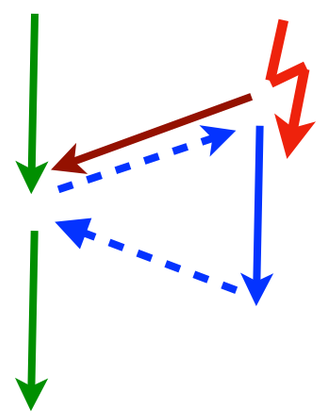


  - i.d.R. nach Beendigung des aktuellen Befehlszyklus
  - Retten des Prozesszustands
  - nach Behandlung i.d.R. Rücksprung

- Erfordern i.d.R. schnelle Behandlung:
  - Interne Uhr geht andernfalls mit der Zeit nach
  - Tastaturpuffer nur klein (Eingabe weiterer Zeichen führt zum Überlauf)
  - Platte braucht neuen Auftrag

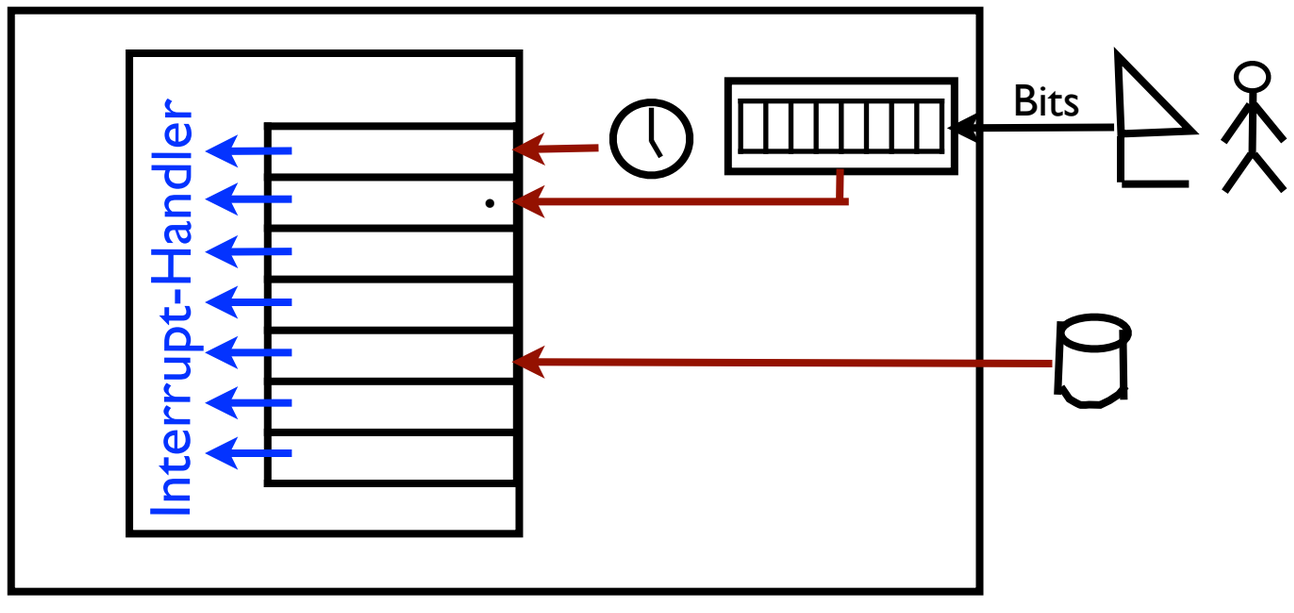
⇒ Ausleihen der CPU vom laufenden Prozess

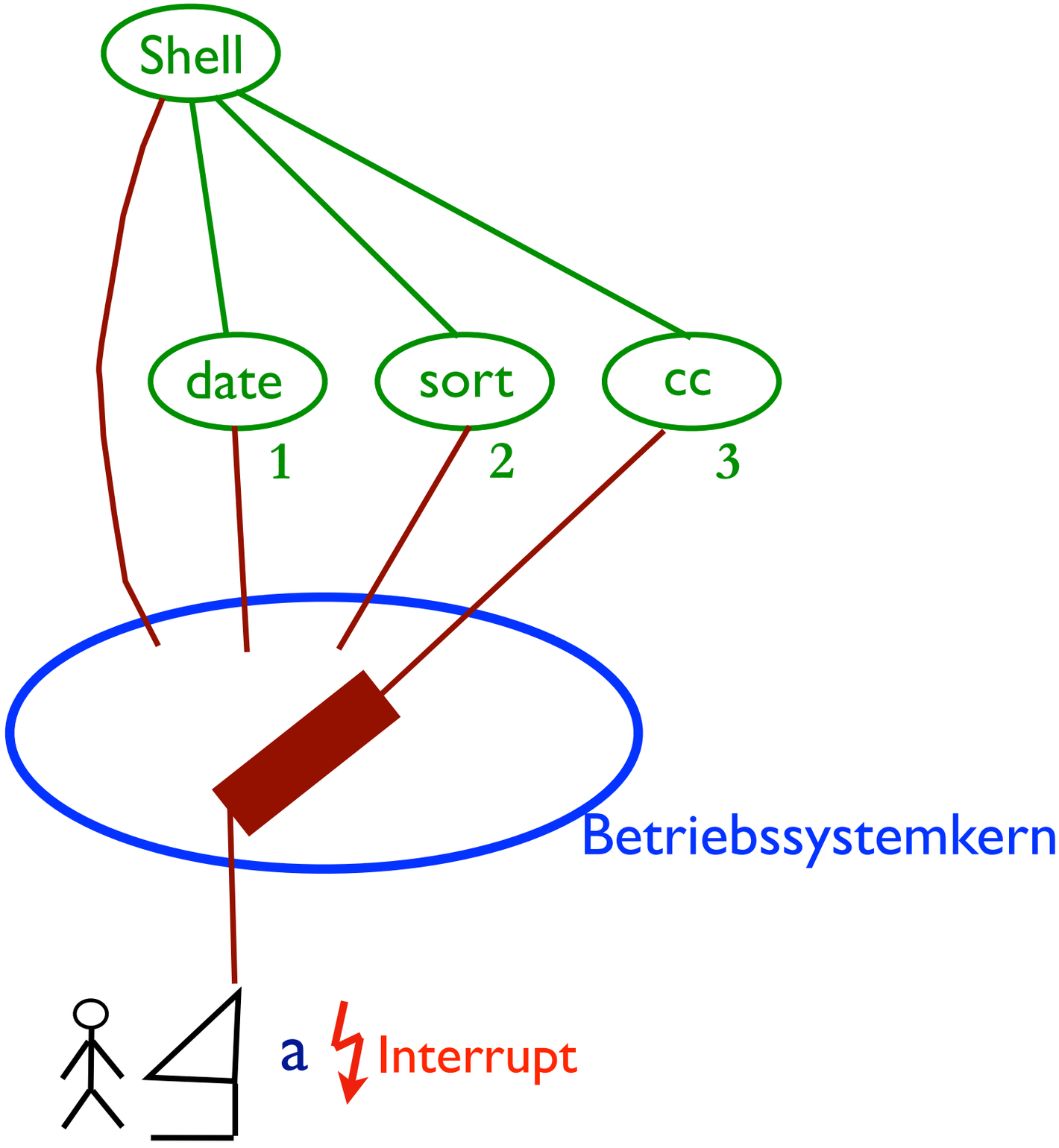
⇒ Prozessunterbrechung (Interrupt)



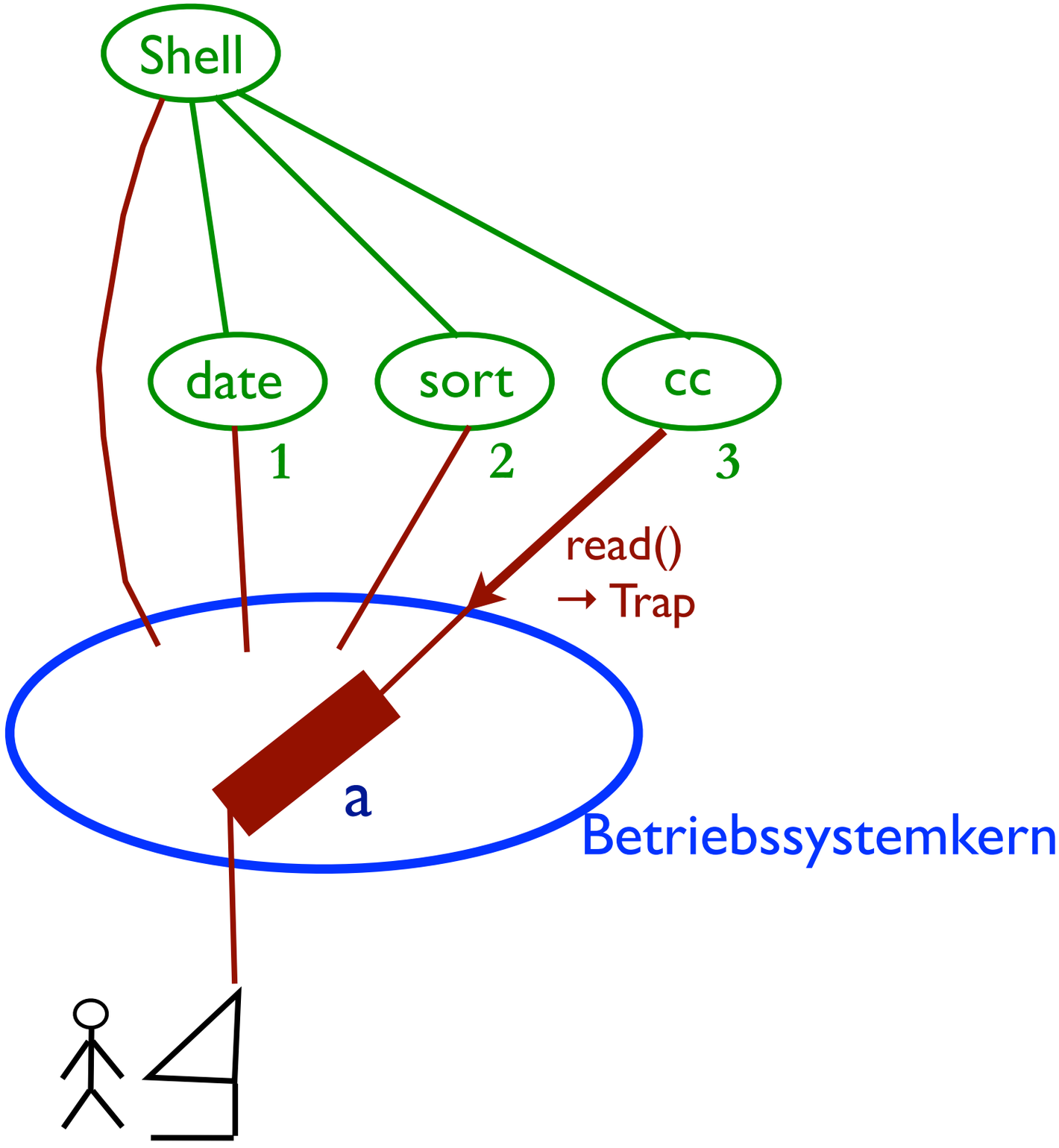
  - i.d.R. nach Beendigung des aktuellen Befehlszyklus
  - Retten des Prozesszustands
  - nach Behandlung i.d.R. Rücksprung

- Ansprung der entsprechenden Behandlungsroutine (Interrupt-Handler, Interrupt Service Routine (ISR))

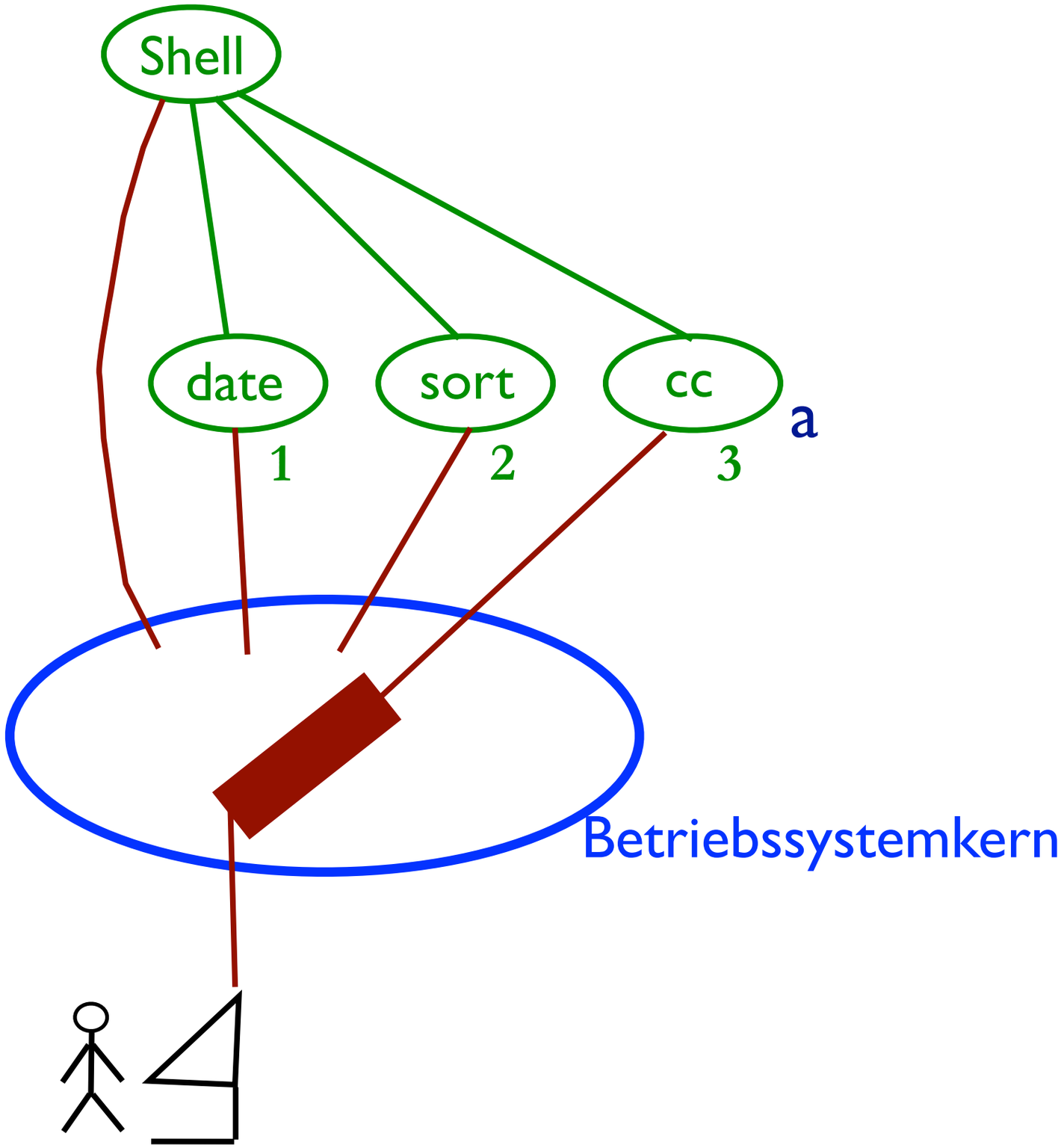




Tastatureingabe
erzeugt Interrupt

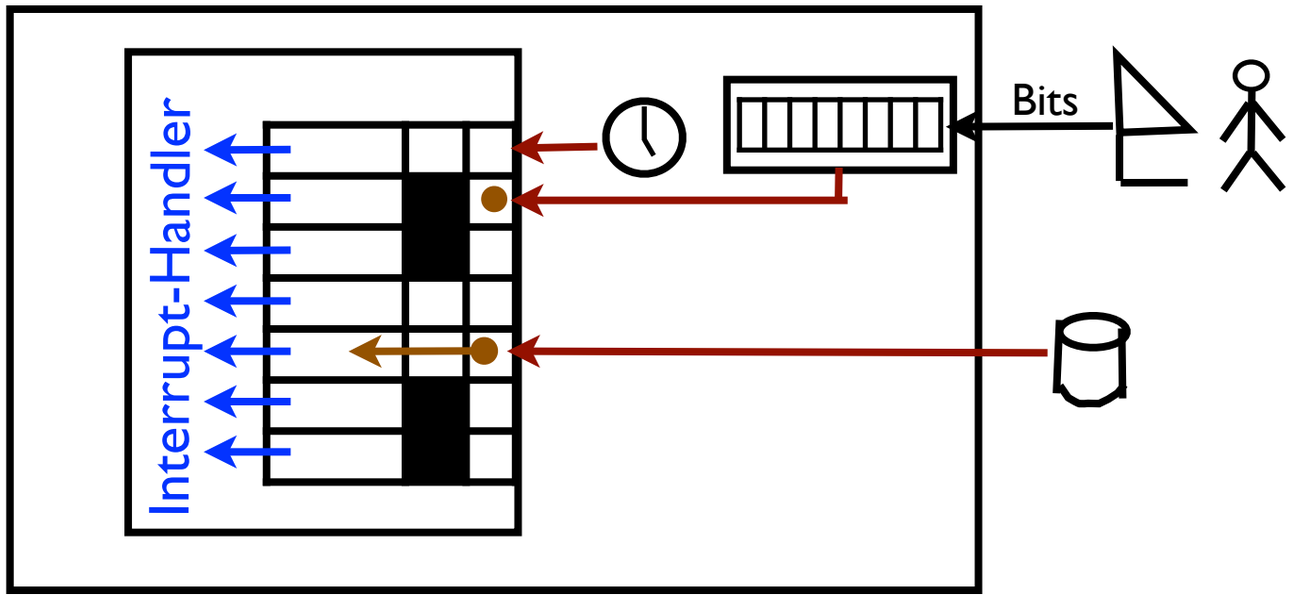


Tastatureingabe
erzeugt Interrupt

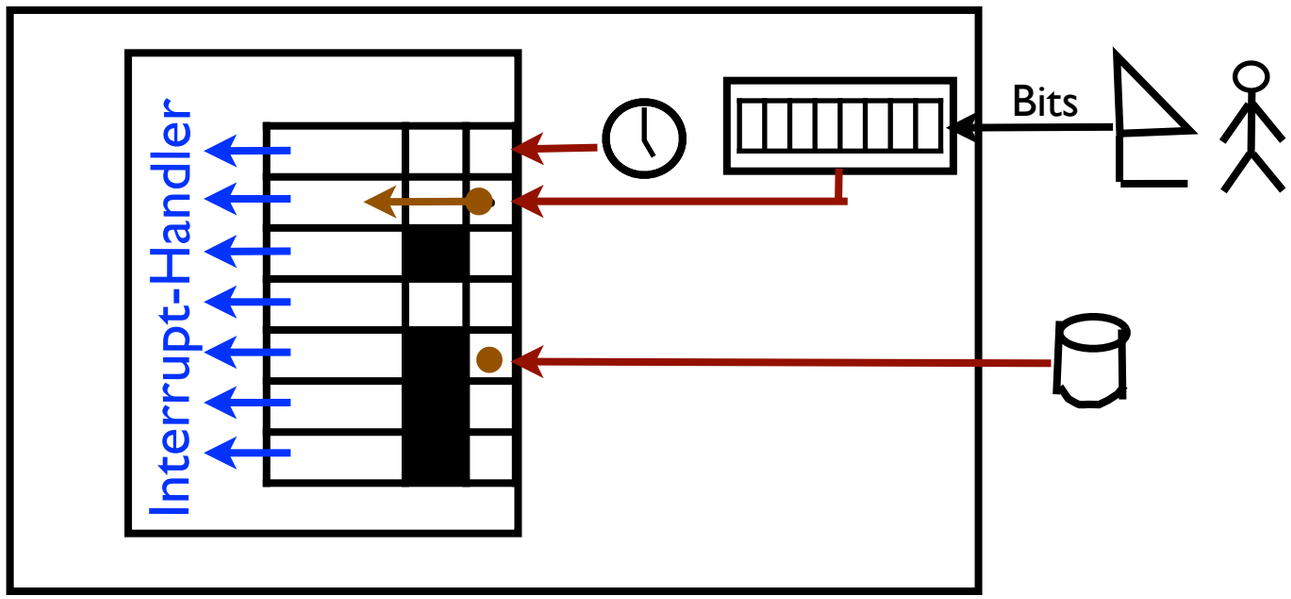


Tastatureingabe
erzeugt Interrupt

- Interrupt kommt u.U. ungelegen (z.B. Zugriff auf gemeinsame Daten)
  ⇒ (Temporäre) Ausmaskierung möglich

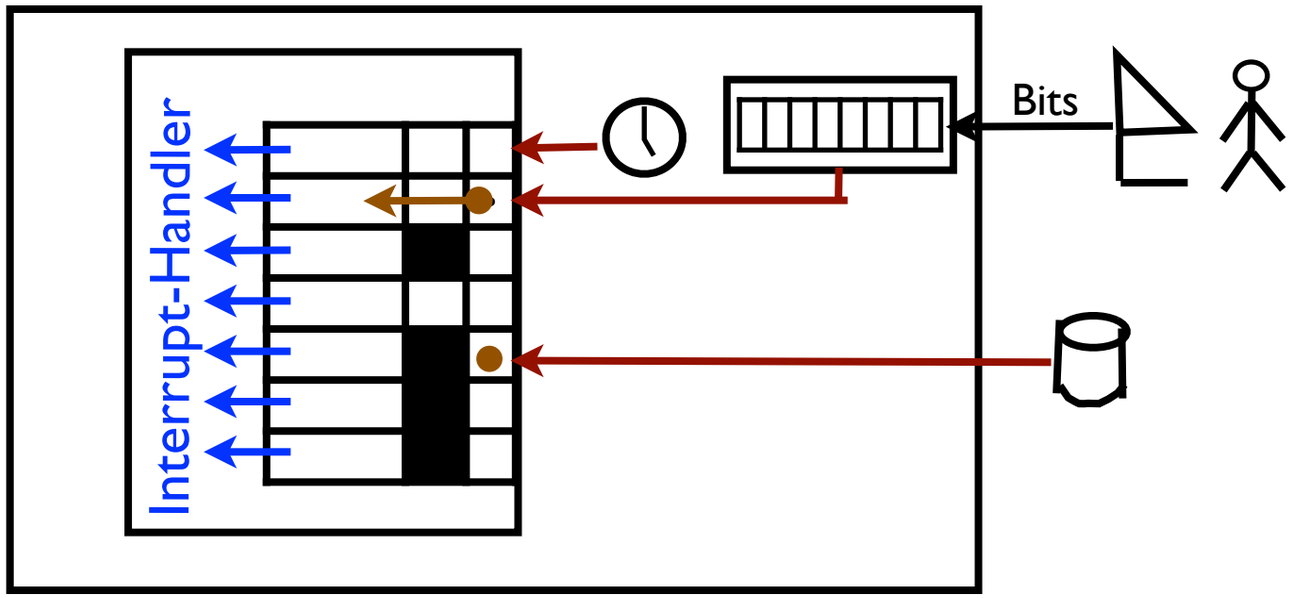


- Interrupt kommt u.U. ungelegen (z.B. Zugriff auf gemeinsame Daten)
  ⇒ (Temporäre) Ausmaskierung möglich



- Unterscheidung:
  - Edge-triggered
    ⇒ ereignisorientiert (Interrupt verpufft u.U. ohne Behandlung)
  - Level-triggered
    ⇒ zustandsorientiert (Interrupt liegt an bis zur Behandlung)

- Interrupt kommt u.U. ungelegen (z.B. Zugriff auf gemeinsame Daten)
  ⇒ (Temporäre) Ausmaskierung möglich



- Unterscheidung:
  - Edge-triggered
    ⇒ ereignisorientiert (Interrupt verpufft u.U. ohne Behandlung)
  - Level-triggered
    ⇒ zustandsorientiert (Interrupt liegt an bis zur Behandlung)
- Mehrere Interrupts können gleichzeitig zur Behandlung anstehen
- Während einer Behandlung kann weiterer Interrupt reinkommen
  ⇒ Was nun?

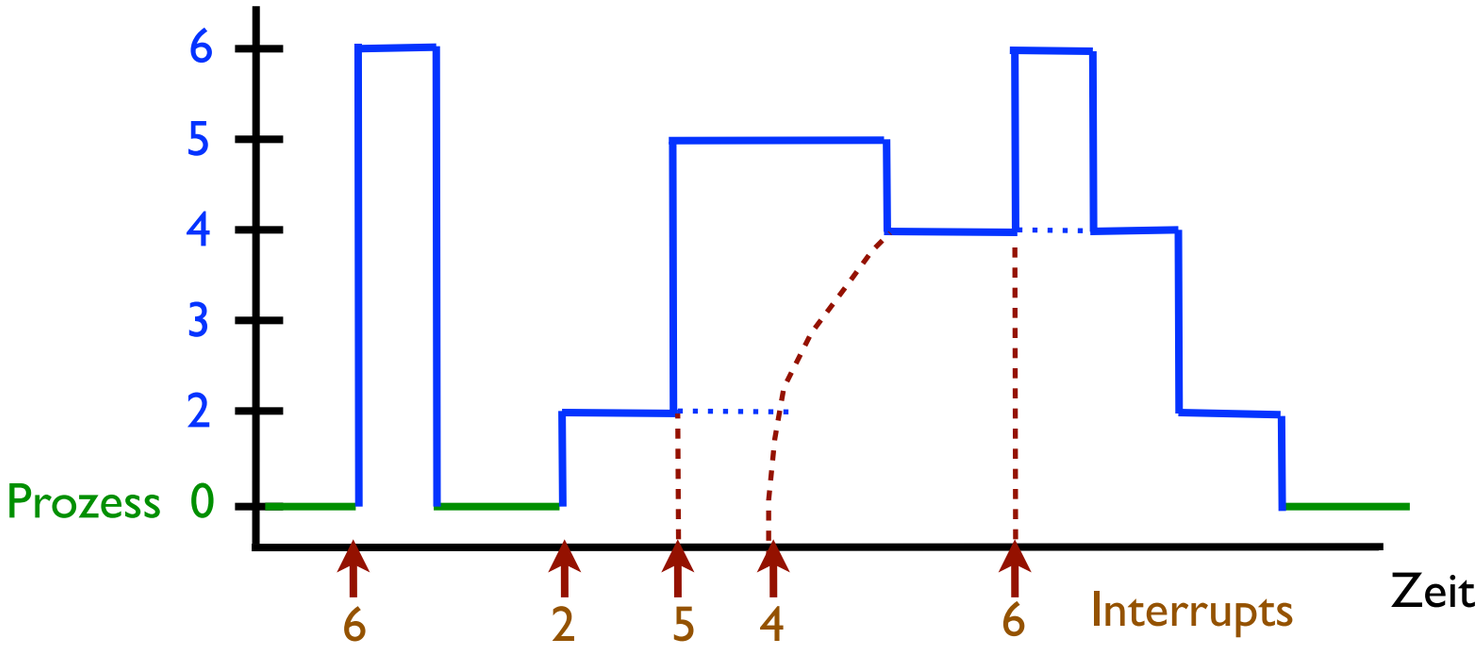
- Interrupts sind unterschiedlich eilig:
  - Uhrtick geht verloren
  - eingelesenes Zeichen geht verloren
  - Platte hat nichts zu tun

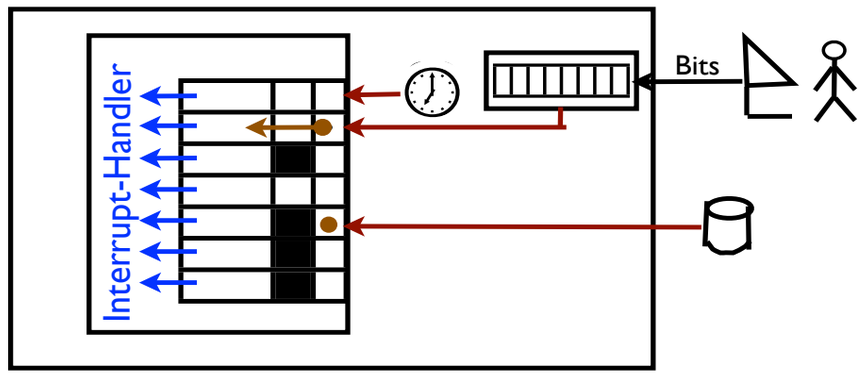
⇒ Interrupts haben unterschiedliche Prioritäten

- Interrupts sind unterschiedlich eilig:
  - Uhrtick geht verloren
  - eingelesenes Zeichen geht verloren
  - Platte hat nichts zu tun

⇒ Interrupts haben unterschiedliche Prioritäten

⇒ Höherpriorisierte Interrupts können Behandlung niederpriorisierter Interrupts unterbrechen (Interrupt-Stack)

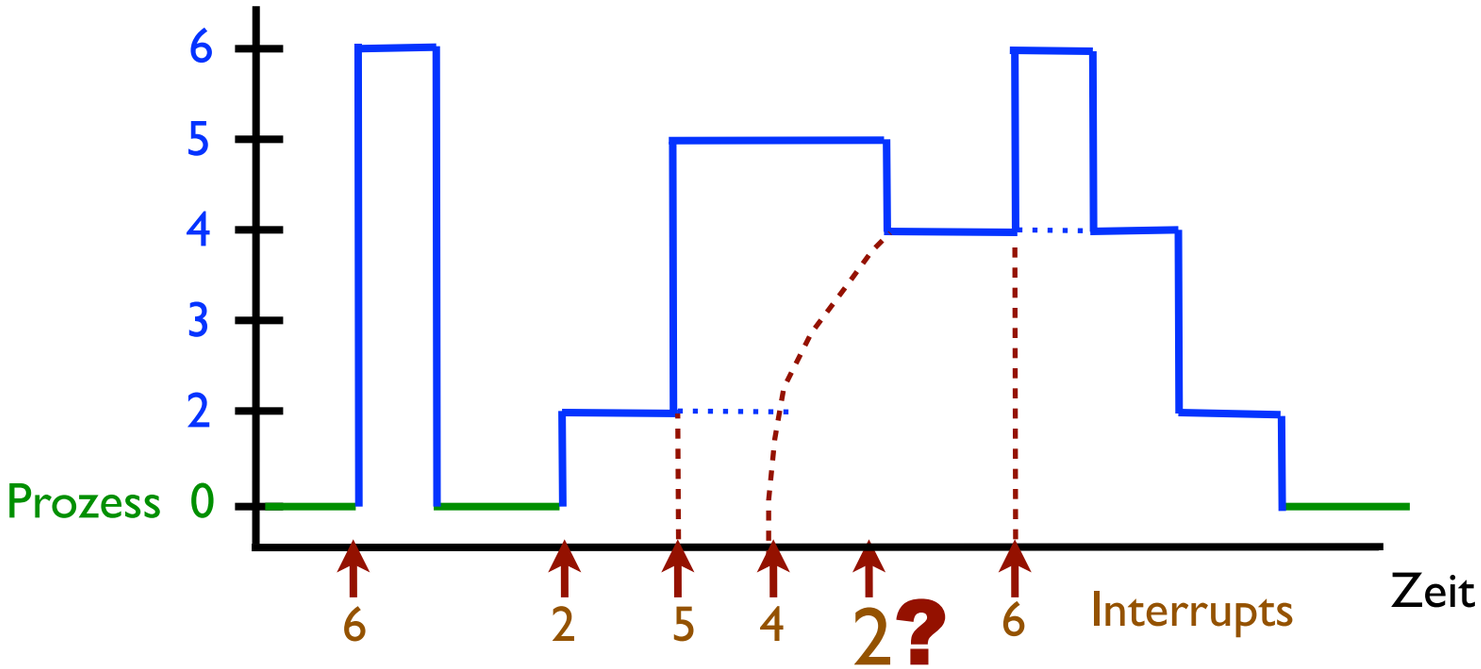


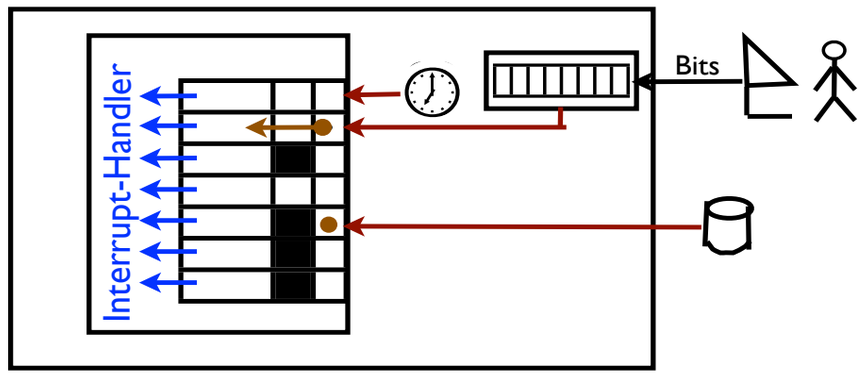


- Interrupts sind unterschiedlich eilig:
  - Uhrtick geht verloren
  - eingelesenes Zeichen geht verloren
  - Platte hat nichts zu tun

⇒ Interrupts haben unterschiedliche Prioritäten

⇒ Höherpriorisierte Interrupts können Behandlung niederpriorisierter Interrupts unterbrechen (Interrupt-Stack)



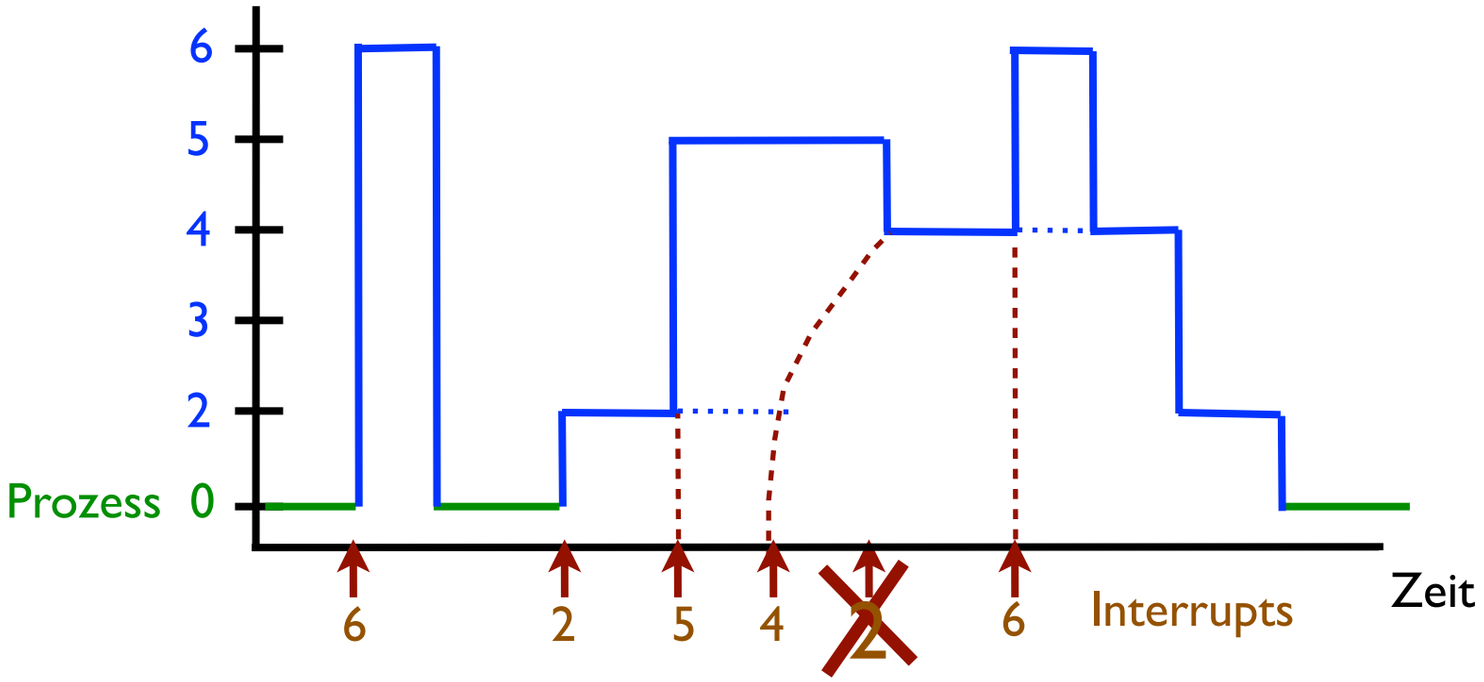


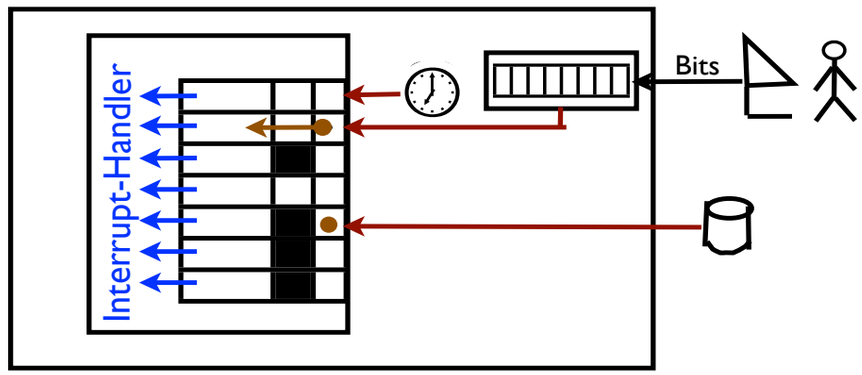
- Können mehrere gleiche Interrupts gleichzeitig anliegen?

  a) Falls ja: Setzt entsprechenden Speicherbereich voraus

- Interrupts sind unterschiedlich eilig:
  - Uhrtick geht verloren
  - eingelesenes Zeichen geht verloren
  - Platte hat nichts zu tun

⇒ Interrupts haben unterschiedliche Prioritäten
⇒ Höherpriorisierte Interrupts können Behandlung niederpriorisierter Interrupts unterbrechen (Interrupt-Stack)





- Können mehrere gleiche Interrupts gleichzeitig anliegen?

  a) Falls ja: Setzt entsprechenden Speicherbereich voraus

  b) Falls nein: De facto „verlorengegangene“ Interrupts möglich

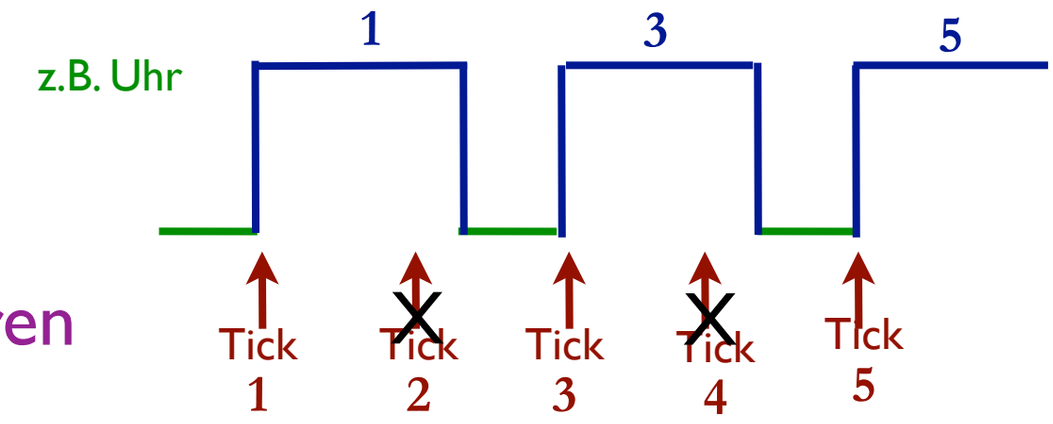
  ⇒ nachfolgend vereinfachend b) angenommen)

Software-Interrupts (≠ Traps)

- Interruptbehandlung kann u.U. recht lange dauern

  ⇒ zu starke Verzögerungen

  ⇒ Ticks 2 und 4 gehen verloren

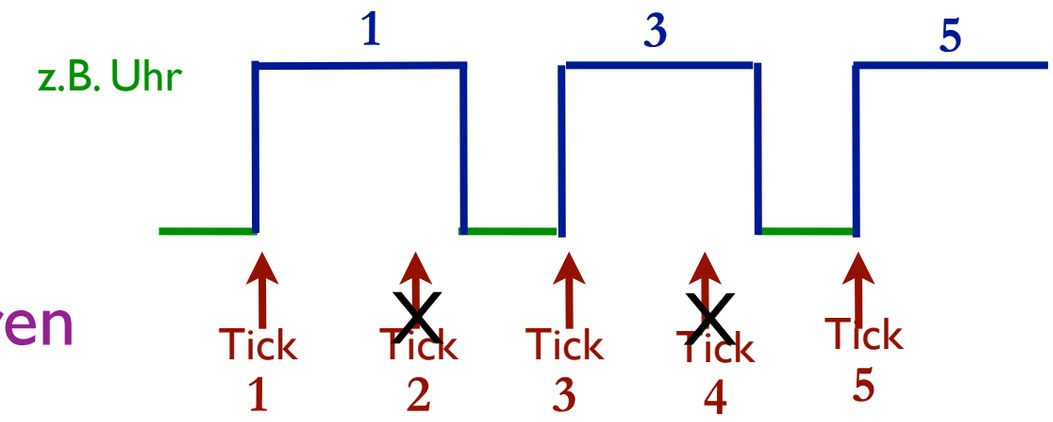


Software-Interrupts (≠ Traps)

- Interruptbehandlung kann u.U. recht lange dauern

  ⇒ zu starke Verzögerungen

  ⇒ Ticks 2 und 4 gehen verloren



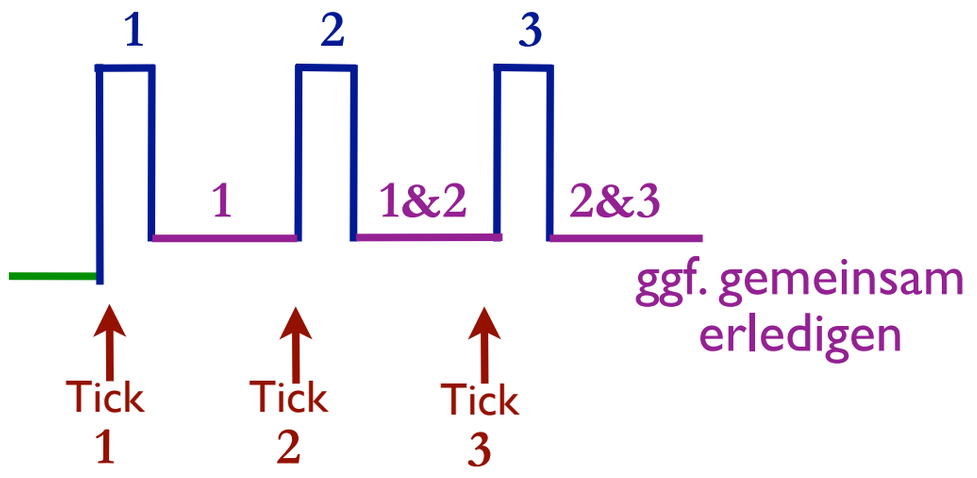
- Nicht alle Aufgaben der Interruptbehandlung sind zeitkritisch

  ⇒ können z.T. verzögert durchgeführt werden

- Abspalten von nicht zeitkritischen Aufgaben

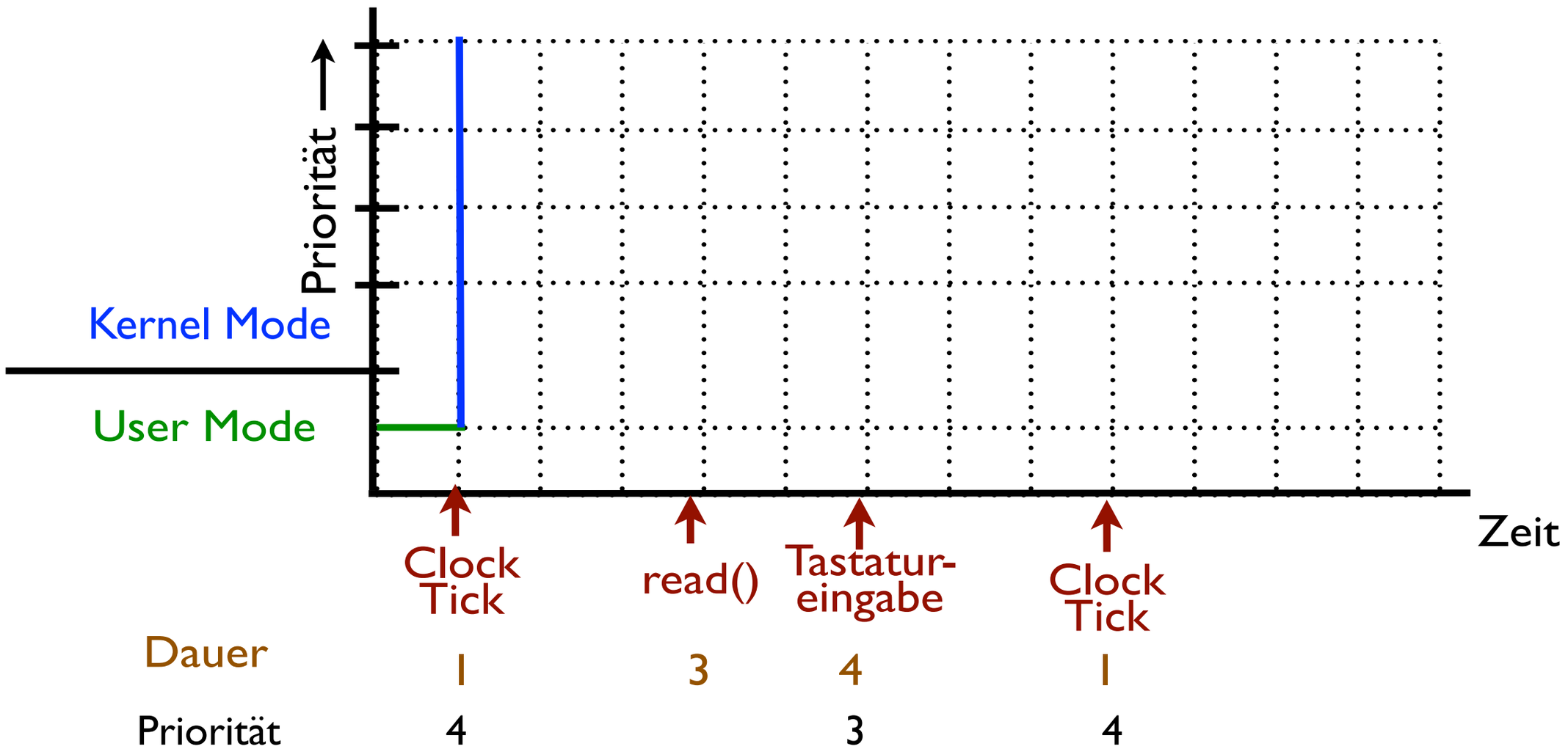
  ⇒ Aktivieren eines „Software-Interrupts“ (niedrigste Priorität)

  (heute u.U. modelliert über „Kernel Threads“)



Kleine Aufgabe

Gegeben sei ein fiktiver Prozessablauf sowie die angegebenen Ereignisse (deren Behandlungsdauer ebenfalls angegeben ist). Wann läuft was?



Fragen – Teil 3

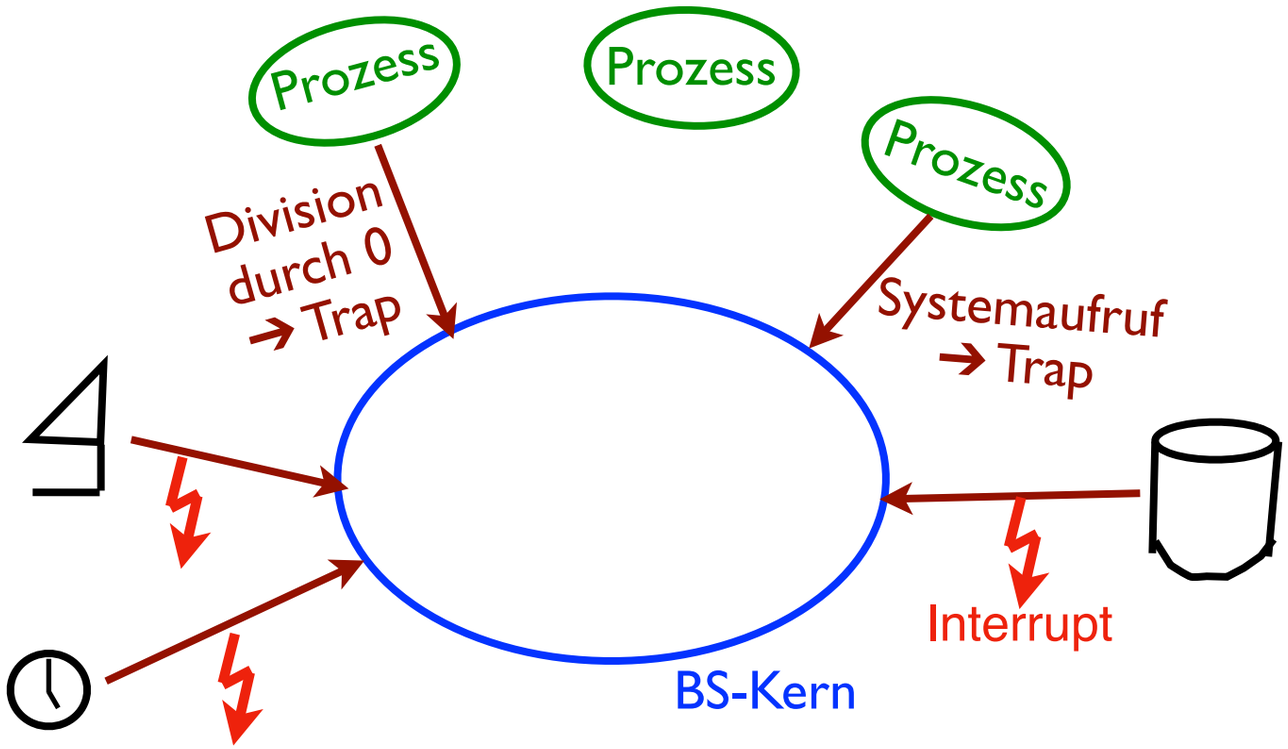
- Was ist ein *Interrupt*? Nenne Beispiele für mögliche Interrupt-Quellen. Warum werden sie unterschiedlich priorisiert? Wie wird ein Interrupt in etwa behandelt?
- Inwiefern unterscheiden sich Traps von Interrupts?

Teil 4:
Betriebssystemkomponenten

05 Traps vs. Interrupts vs. Signale, Ute Bormann

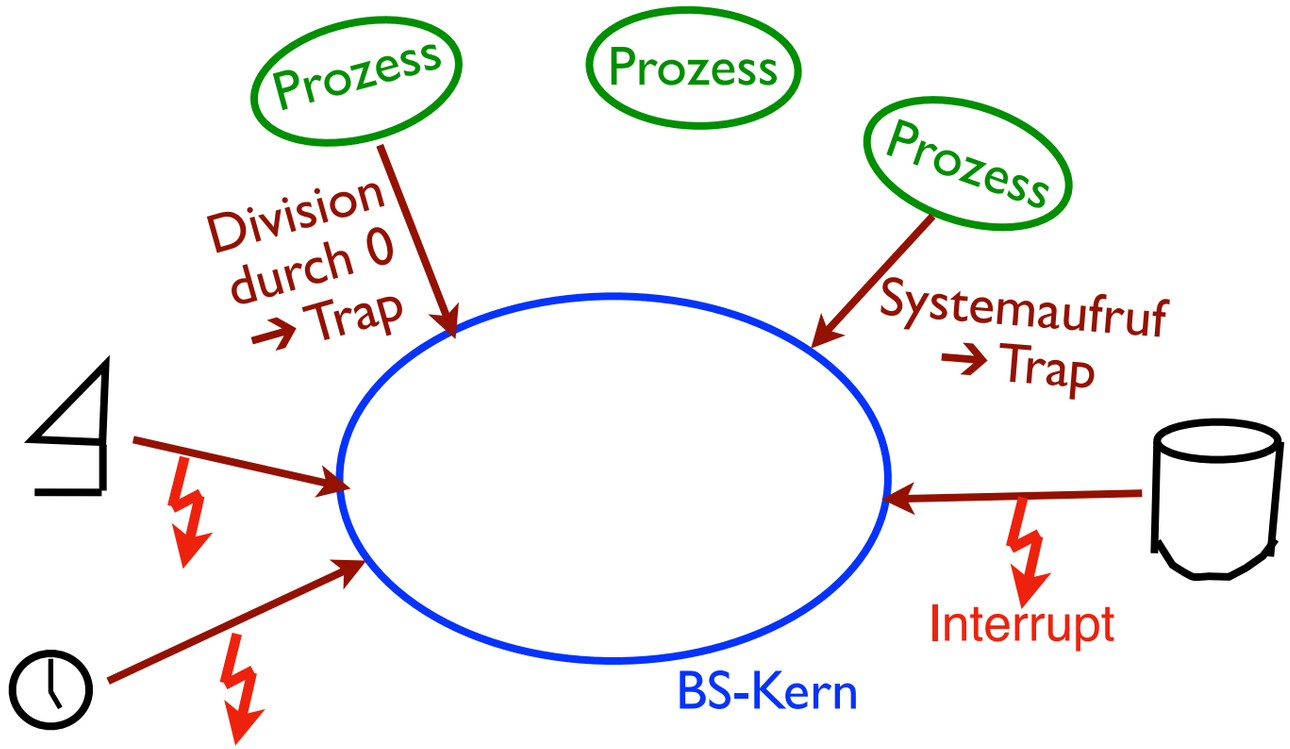
Zusammenfassend

a) Eingänge in den Kern



Zusammenfassend

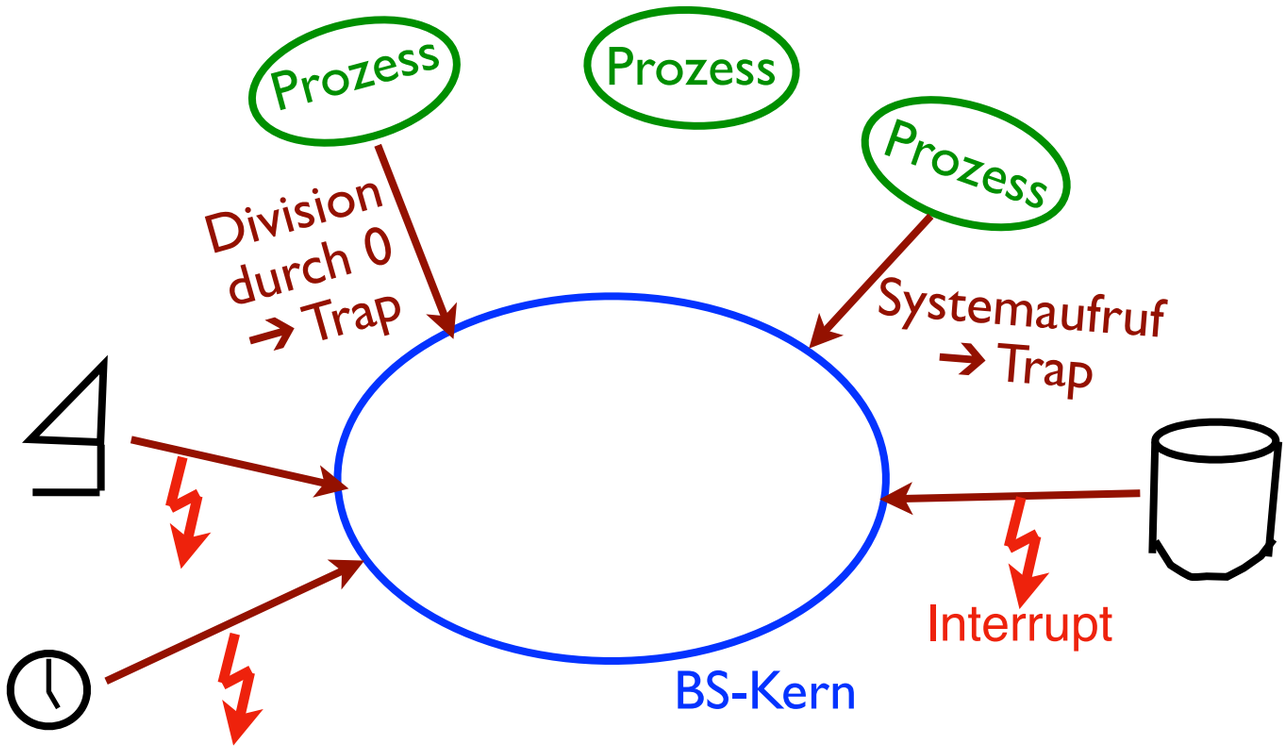
a) Eingänge in den Kern



- Retten des aktuellen (Prozess)Zustands
- (Umschalten in den Kernel-Mode)
- Abarbeiten von Betriebssystem-Code zur Behandlung (privilegierte Instruktionen)
- Ggf. Rückkehr in den alten (Prozess)Zustand

Zusammenfassend

a) Eingänge in den Kern



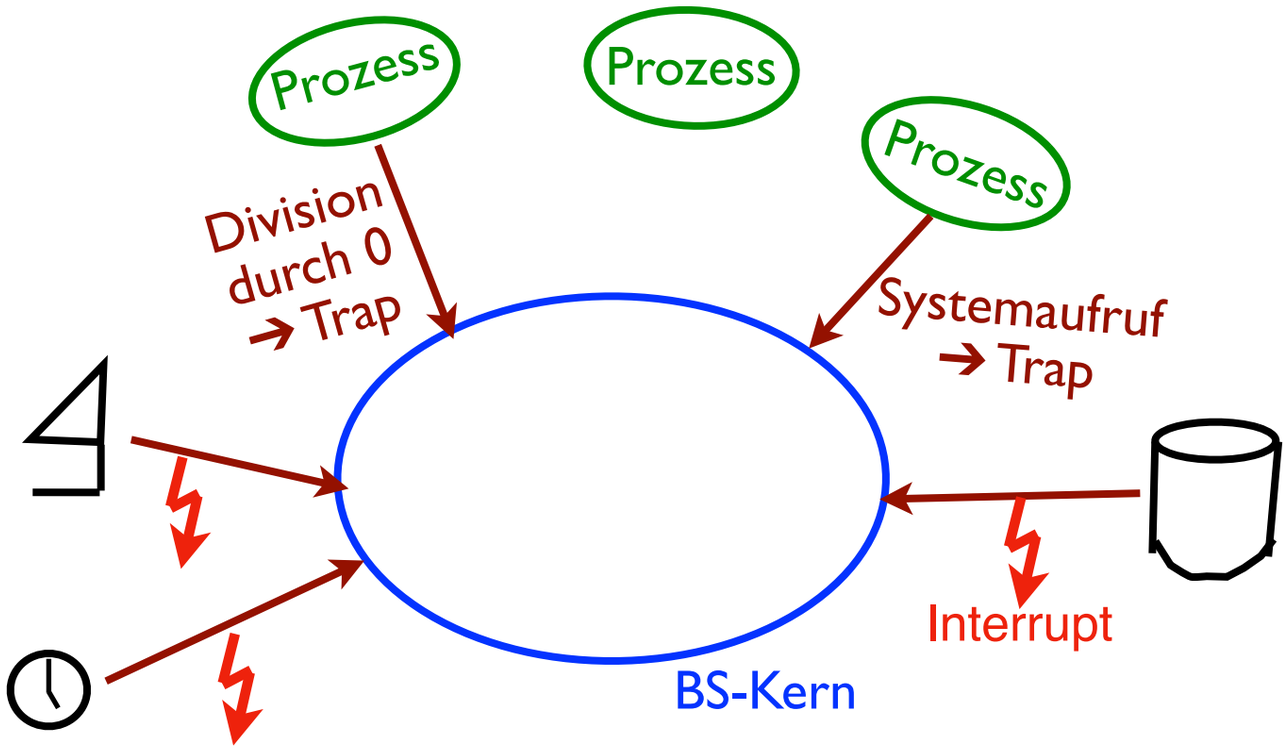
- Retten des aktuellen (Prozess)Zustands
- (Umschalten in den Kernel-Mode)
- Abarbeiten von Betriebssystem-Code zur Behandlung (privilegierte Instruktionen)
- Ggf. Rückkehr in den alten (Prozess)Zustand

b) Betriebssystemaktivitäten

- Nach Trap im Kernel-Mode eines „normalen“ Prozesses (⇒ Traphandler)
- Im Interrupthandler (im Kernel-Mode)

Zusammenfassend

a) Eingänge in den Kern



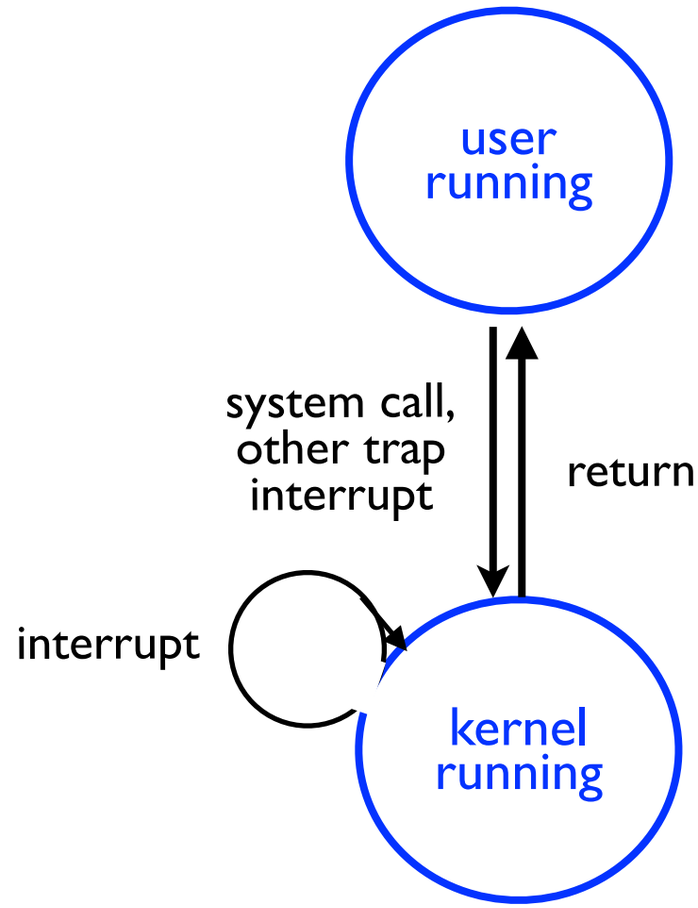
- Retten des aktuellen (Prozess)Zustands
- (Umschalten in den Kernel-Mode)
- Abarbeiten von Betriebssystem-Code zur Behandlung (privilegierte Instruktionen)
- Ggf. Rückkehr in den alten (Prozess)Zustand

b) Betriebssystemaktivitäten

- Nach Trap im Kernel-Mode eines „normalen“ Prozesses (⇒ Traphandler)
- Im Interrupthandler (im Kernel-Mode)
- In Systemprozessen (i.d.R. im User-Mode) bzw. Kernel Threads

Prozesszustände

(vereinfacht)



- Betriebssystem verwaltet Betriebsmittel:
  - Zugriff auf die Geräte ⇒ Geräteverwaltung
    - Wie geht's?
    - Wer darf wann?
  - Zugriff auf Dateien ⇒ Dateiverwaltung
  - Abbildung Adressraum → Hauptspeicher ⇒ Speicherverwaltung
  - Dabei: Unterscheidung mehrerer Aktivitäten (Prozesse)
    ⇒ Prozessverwaltung
  - Nachrichten an andere Prozesse
    ⇒ Interprozesskommunikation (zu anderen Sytemen)

- Betriebssystem verwaltet Betriebsmittel:

④
- Zugriff auf die Geräte ⇒ Geräteverwaltung
  - Wie geht's?
  - Wer darf wann?

③
- Zugriff auf Dateien ⇒ Dateiverwaltung

②
- Abbildung Adressraum → Hauptspeicher ⇒ Speicherverwaltung

①
- Dabei: Unterscheidung mehrerer Aktivitäten (Prozesse)

⑤
⇒ Prozessverwaltung

⑥
- Nachrichten an andere Prozesse
  ⇒ Interprozesskommunikation (zu anderen Sytemen)

- Gebiete eng miteinander verzahnt

  ⇒ keine natürliche Reihenfolge

- Am Beispiel Unix

Prozessverwaltung in Unix

Verschiedene Teilaufgaben:

- Prozesserzeugung und -termination
- Prozesswechsel (Scheduling)
- Nebenläufigkeit
- Interprozesskommunikation (Spezialform: Signale)

- Datenstrukturen zur Prozessverwaltung

Prozessverwaltung in Unix

Verschiedene Teilaufgaben:

④ • Prozesserzeugung und -termination

② • Prozesswechsel (Scheduling)

⑤ • Nebenläufigkeit

⑥ • Interprozesskommunikation (Spezialform: Signale)

①

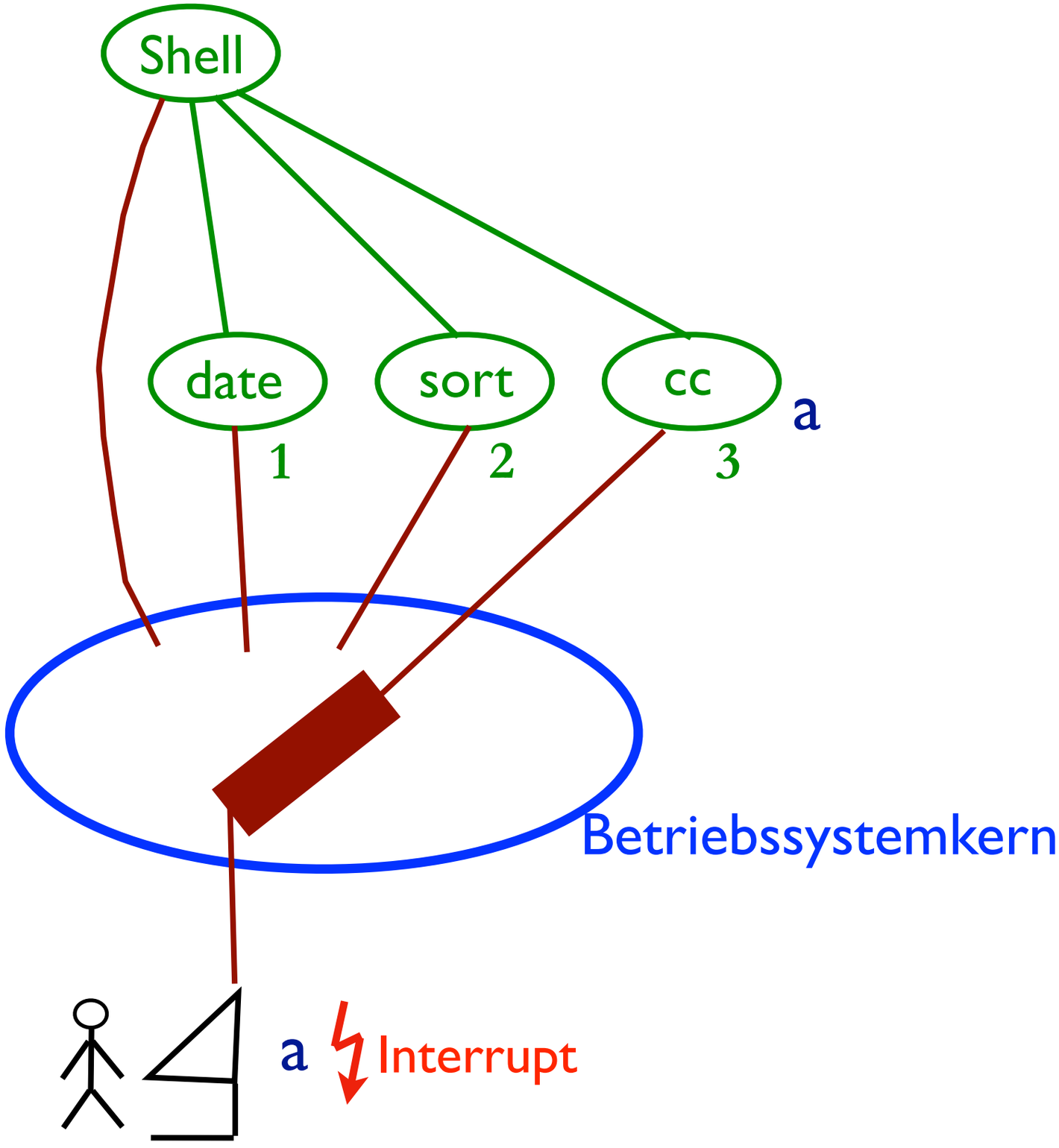
③ • Datenstrukturen zur Prozessverwaltung

Fragen – Teil 4

- In welchen Kontexten finden Betriebssystemaktivitäten statt?
- Aus welchen Komponenten besteht ein typisches Betriebssystem?

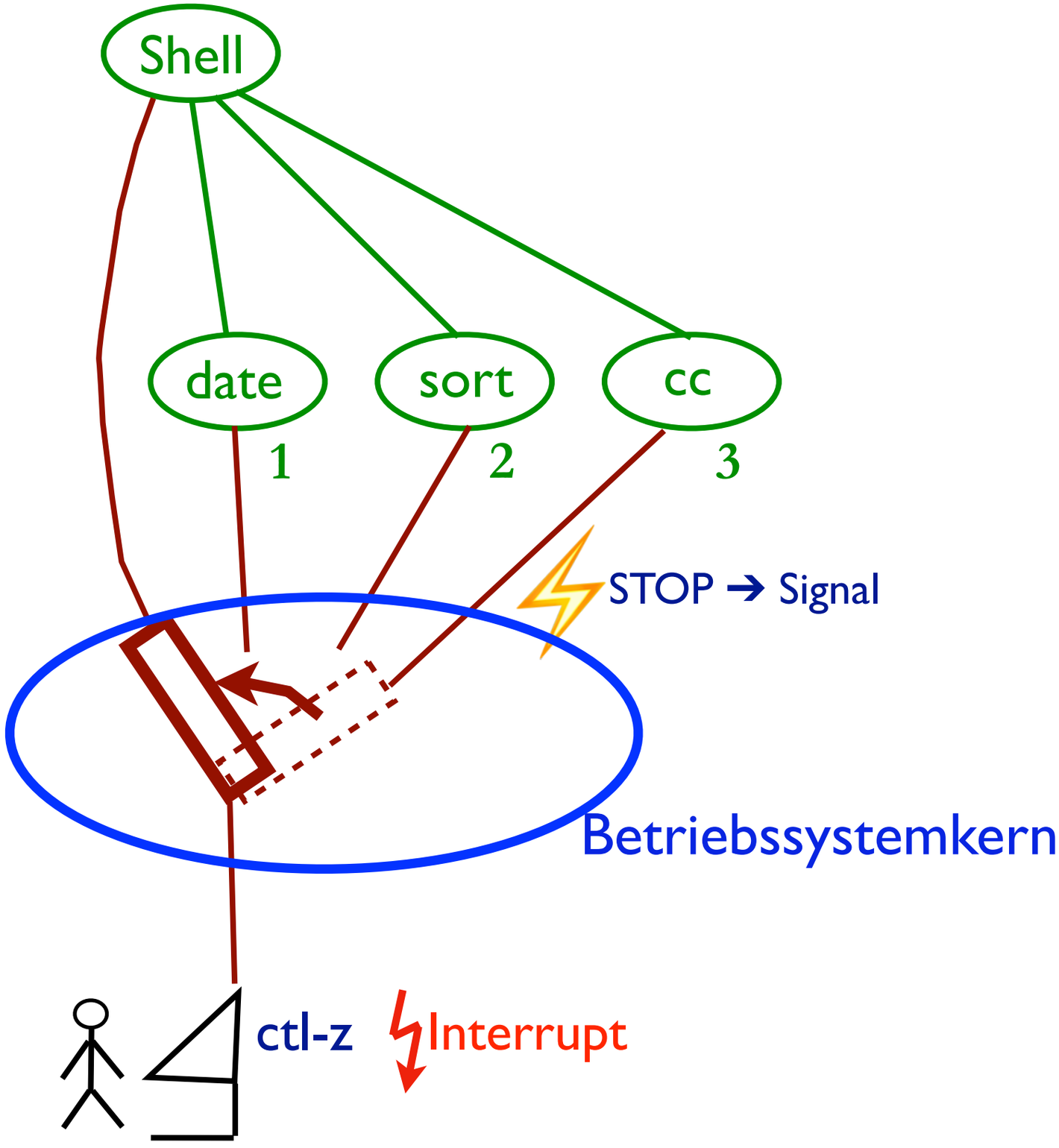
Teil 5:
Signale

05 Traps vs. Interrupts vs. Signale, Ute Bormann



Tastatureingabe
erzeugt Interrupt

Signale



Auch Eingabe von ctl-z löst Interrupt aus;
im Betriebssystem i.d.R. Umwandlung in STOP-Signal

Signale

Melden von „Ausnahmesituationen“ an Prozesse:

- Aus einem Interrupthandler (i.d.R. bei Tastatureingabe), z.B. Eingabe von

  ctl-z ⇒ SIGSTOP

  ctl-c ⇒ SIGINT

Signale

Melden von „Ausnahmesituationen“ an Prozesse:

- Aus einem Interrupthandler (i.d.R. bei Tastatureingabe), z.B. Eingabe von

  ctl-z ⇒ SIGSTOP

  ctl-c ⇒ SIGINT

- Von einem anderen Prozess (per Systemaufruf):

  kill (pid, sig)

  kill (4711, SIGKILL)

  kill (42, SIGCHLD)

  (ggf. vom Benutzer aus per Shell-Kommando, z.B. kill -KILL 4711 (benötigt Rechte))

Signale

Melden von „Ausnahmesituationen“ an Prozesse:

- Aus einem Interrupthandler (i.d.R. bei Tastatureingabe), z.B. Eingabe von

  ctl-z ⇒ SIGSTOP

  ctl-c ⇒ SIGINT

- Von einem anderen Prozess (per Systemaufruf):

  kill (pid, sig)

  kill (4711, SIGKILL)

  kill (42, SIGCHLD)

  (ggf. vom Benutzer aus per Shell-Kommando, z.B. kill -KILL 4711 (benötigt Rechte))

- Aber auch aus einem Traphandler:

  Division durch 0 ⇒ SIGFPE

  Illegale Instruktion ⇒ SIGILL

  Illegaler Speicherzugriff ⇒ SIGSEGV, SIGBUS

  ⇒ warum?

  - Traphandler innerhalb des Kernel implementiert
  - Prozesse möchten u.U. eigene Trapbehandlung machen

    ⇒ schicken „sich selbst“ Signal

- Mögliche Reaktionen auf empfangenes Signal:
  - Prozess terminiert
  - Prozess behandelt Signal
    - u.U. im Kern (SIGSTOP)
    - u.U. im User-Mode (anwendungsspezifisch)
  - Prozess ignoriert Signal

⇒ Default meist Termination (abhängig vom Signal, z.B. *nicht* bei SIGSTOP/SIGCHLD)

⇒ Bei SIGKILL immer

- Mögliche Reaktionen auf empfangenes Signal:
  - Prozess terminiert
  - Prozess behandelt Signal
    - u.U. im Kern (SIGSTOP)
    - u.U. im User-Mode (anwendungsspezifisch)
  - Prozess ignoriert Signal

⇒ Default meist Termination (abhängig vom Signal, z.B. *nicht* bei SIGSTOP/SIGCHLD)

⇒ Bei SIGKILL immer

- Prozess kann Signalbehandlung beim System anmelden (per Systemaufruf):

  signal (sig, func)

  welches Signal — welcher Signalhandler (oder ignorieren: SIG-IGN)

  ⇒ wird beim „Ausliefern“ berücksichtigt

- Mögliche Reaktionen auf empfangenes Signal:
  - Prozess terminiert
  - Prozess behandelt Signal
    - u.U. im Kern (SIGSTOP)
    - u.U. im User-Mode (anwendungsspezifisch)
  - Prozess ignoriert Signal

⇒ Default meist Termination (abhängig vom Signal, z.B. *nicht* bei SIGSTOP/SIGCHLD)

⇒ Bei SIGKILL immer

- Prozess kann Signalbehandlung beim System anmelden (per Systemaufruf):

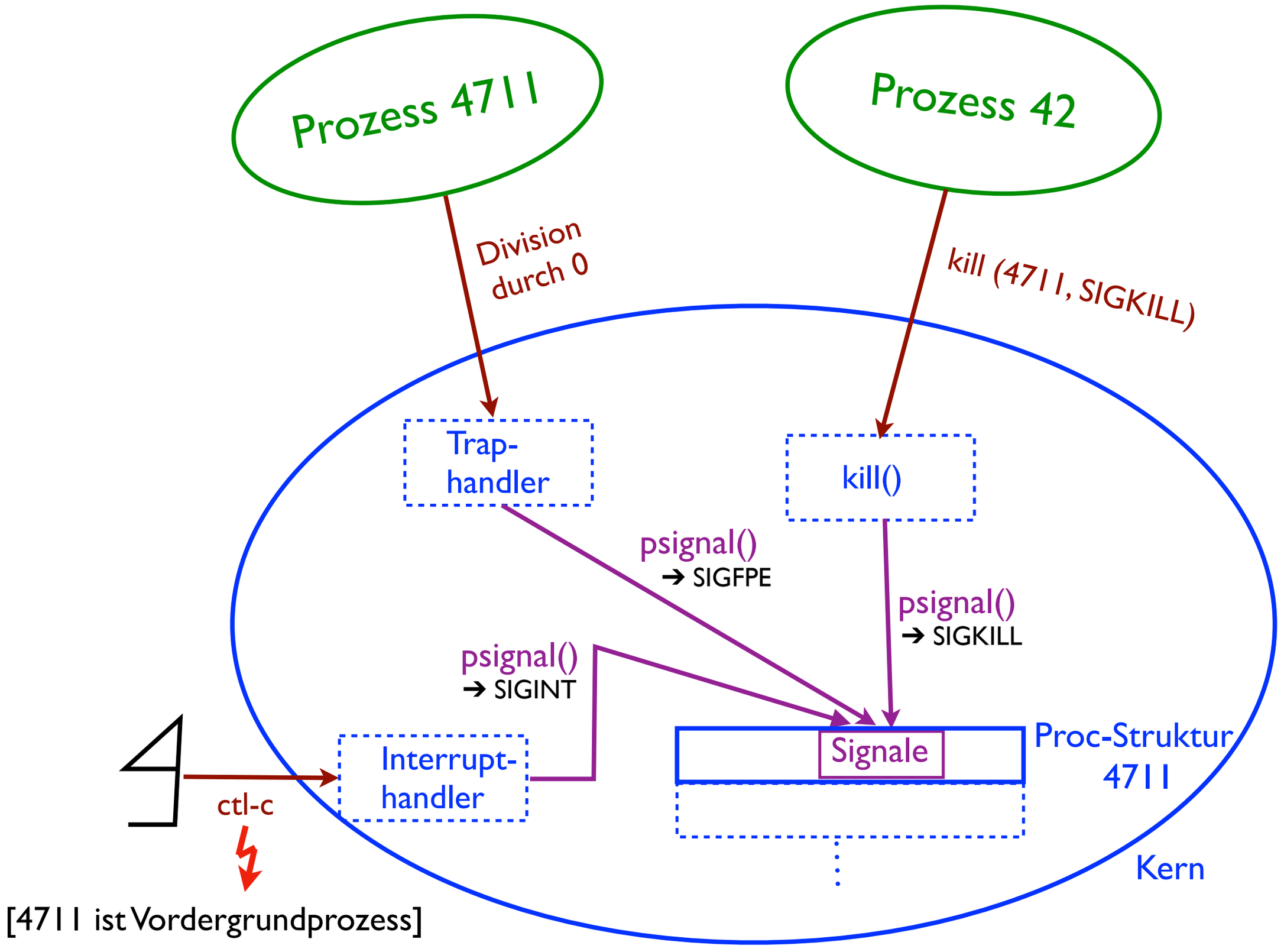
  signal (sig, func)

  welches Signal welcher Signalhandler (oder ignorieren: SIG-IGN)

  ⇒ wird beim „Ausliefern“ berücksichtigt

- Absetzen von Signalen

  psignal (Prozess, Signal) (betriebssystemintern)
  - Soll ignoriert werden? ⇒ Fertig
  - Sonst vermerken in Proc-Struktur des Prozesses



[4711 ist Vordergrundprozess]

- Wann ausliefern? Verschiedene Fälle (Einprozessorsystem):

a) Prozess war eigentlich gerade in Besitz der CPU
(bei Trap, evtl. bei Interrupt)

- Auslieferung bei Rückkehr in den User-Mode

Globale Systemaufrufroutine (kernintern)

- Retten des User-Stack-Pointers (usp) 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 ➔ später)
- (ggf. Signalauslieferung ➔ später)
- Kernel-Stack aufräumen
- Zurück in Bibliotheksroutine write() ⇒ User-Mode

  ⇒ dort User-Stack aufräumen

- Wann ausliefern? Verschiedene Fälle (Einprozessorsystem):

a) Prozess war eigentlich gerade in Besitz der CPU
(bei Trap, evtl. bei Interrupt)

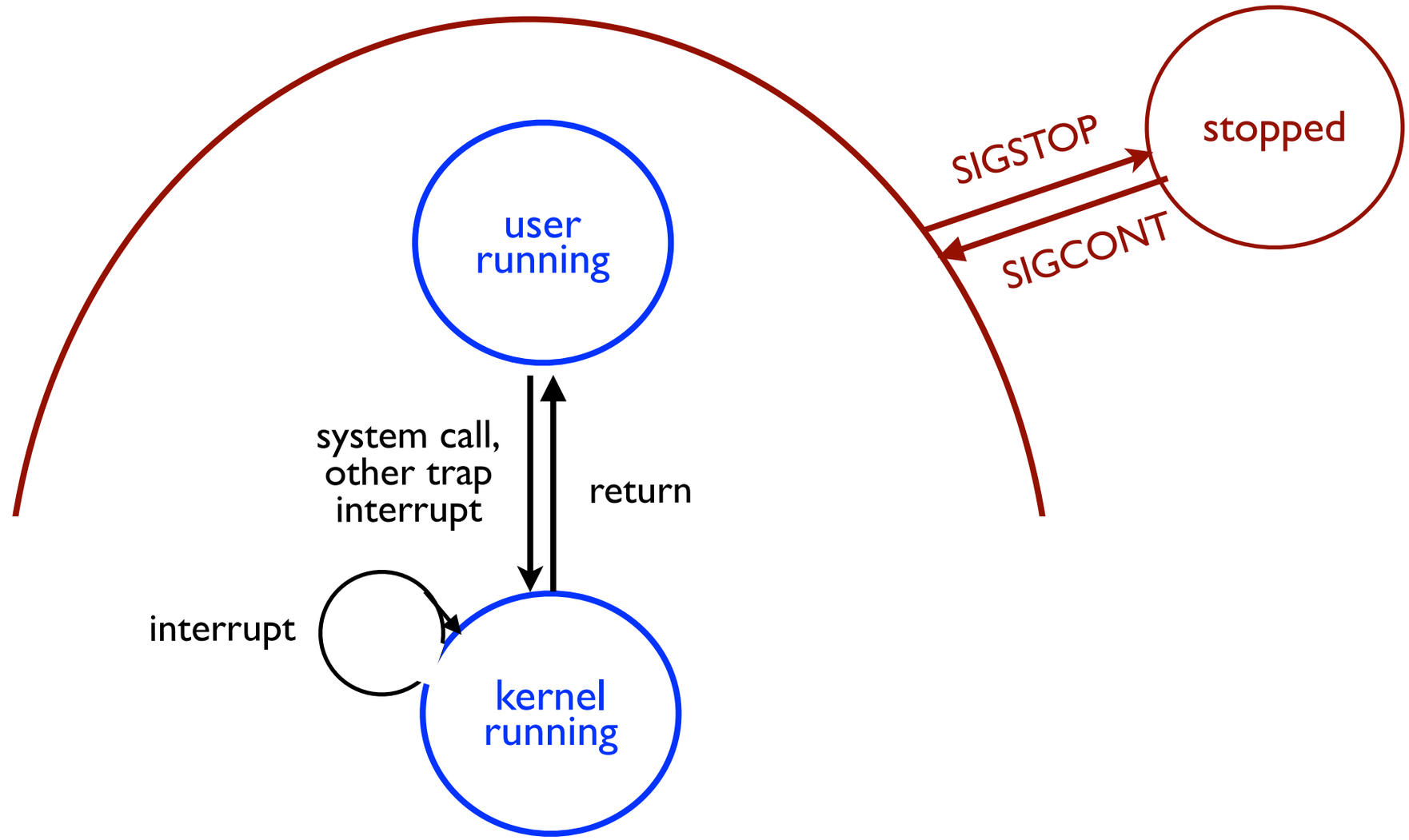
  - Auslieferung bei Rückkehr in den User-Mode

b) Anderer Prozess hat(te) gerade CPU
(bei kill(), oft bei Interrupt)

  - Auslieferung, wenn empfangender Prozess CPU wiedererhält
  - Evtl. beschleunigen
    ⇒ u.U. vorzeitiges Reaktivieren nach SSLEEP (→ später)

Prozesszustände

(vereinfacht)



Fragen – Teil 5

- Was ist ein *Signal*? Nenne Beispiele für mögliche Signalquellen. Wie kann ein Prozess auf ein Signal reagieren?

Zusammenfassung

- Aktivierung des Betriebssystemkerns
- User-Mode vs. Kernel-Mode
- Systemaufrufe und andere Traps
- Interrupts
- Betriebssystemkomponenten
- Signale

Traps vs. Interrupts vs. Signale – Fragen

1. Worin unterscheidet sich der *Kernel-Mode* vom *User-Mode* (in Unix)? Warum wird diese Unterscheidung getroffen?

2. Was passiert in etwa bei einem *Systemaufruf*? (Reihenfolge der Arbeitsschritte.)

3. Was ist ein *Trap*? Nenne Beispiele.

4. Was ist ein *Interrupt*? Nenne Beispiele für mögliche Interrupt-Quellen. Warum werden sie unterschiedlich priorisiert? Wie wird ein Interrupt in etwa behandelt?

5. Inwiefern unterscheiden sich Traps von Interrupts?

6. In welchen Kontexten finden Betriebssystemaktivitäten statt?

7. Aus welchen Komponenten besteht ein typisches Betriebssystem?

8. Was ist ein *Signal*? Nenne Beispiele für mögliche Signalquellen. Wie kann ein Prozess auf ein Signal reagieren?