

Betriebssysteme

Prozesse

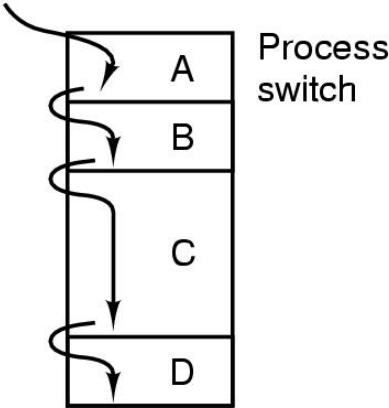
Prof. Dr.-Ing. Torben Weis
Universität Duisburg-Essen

Prozesse

- Prozess =
 - Programm in der Ausführungsphase (Benutzer–Sicht)
 - Einheiten, die die CPU belegen (Betriebssystem–Sicht)
 - Prozesskontrollblock (Systemprogrammierer–Sicht)
 - Basismodell für die Ausführung von Aktionen in einem Rechnersystem
 - Vom Betriebssystem zur Verfügung gestellte und verwaltete Abstraktion
 - Sammlung von Code, Daten, Registern, Stackzeiger, Instruktionszeiger, etc.

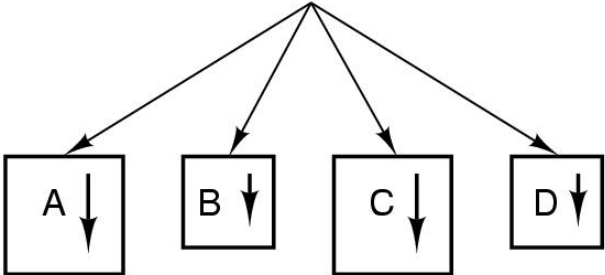
Prozesse

One program counter

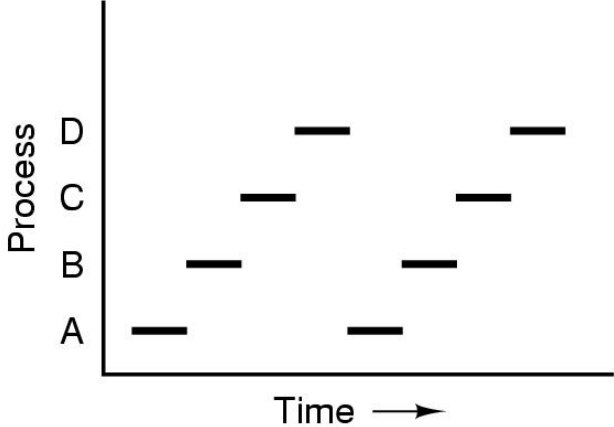


(a)

Four program counters



(b)



(c)

Prozesskontrollblock (PCB)

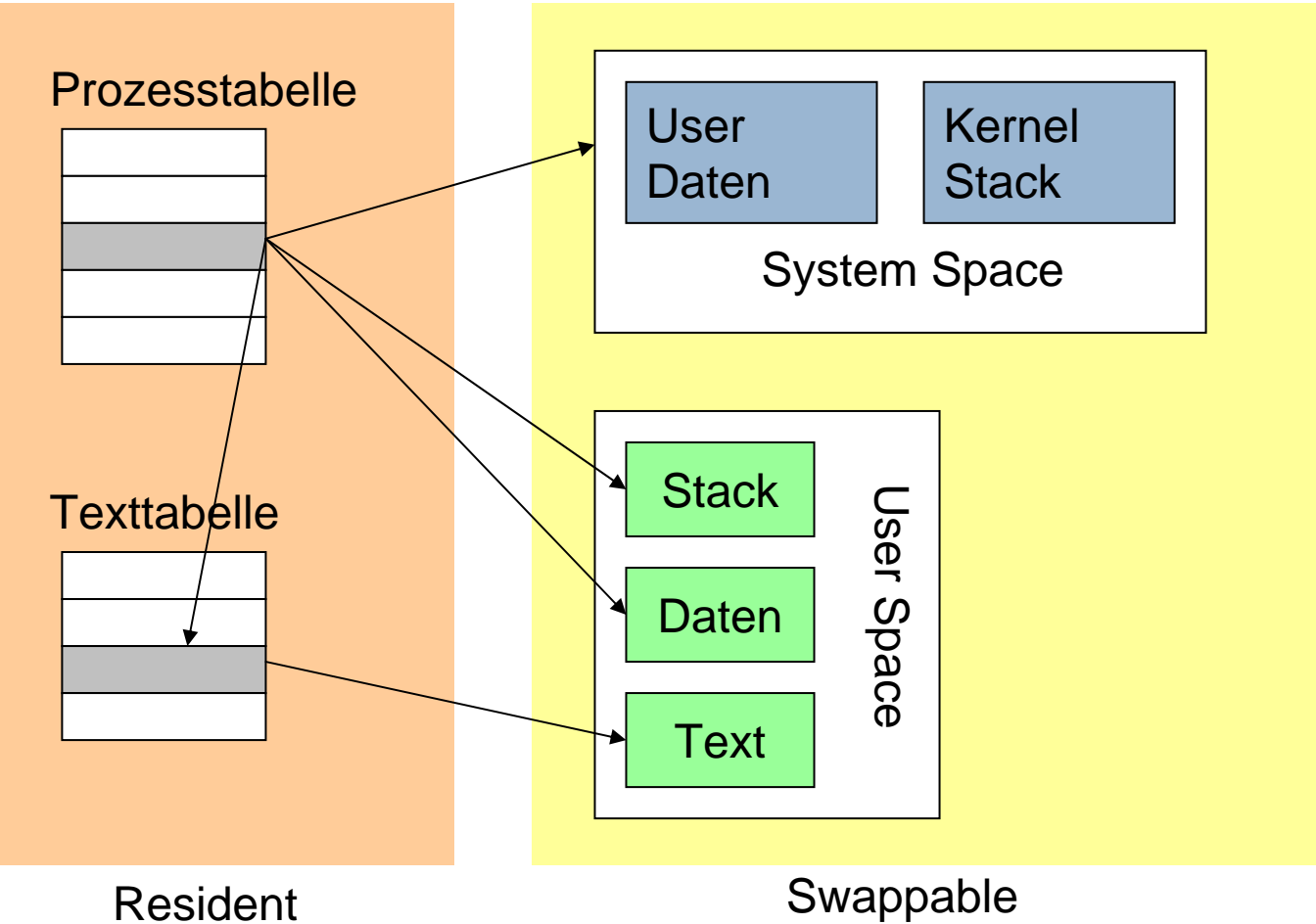
- Datenstruktur mit notwendigen Informationen zu einem Prozess

Prozessnummer (ID)
Programmzähler (PC)
Datensegment-Zeiger
Stack-Zeiger (SP)
Programmstatuswort
Verbrauchte Zeit
Erzeugungszeitpunkt
Aktuelles Verzeichnis
Signalstatus
Zeiger auf Nachrichten
Register

Prozessverwaltung

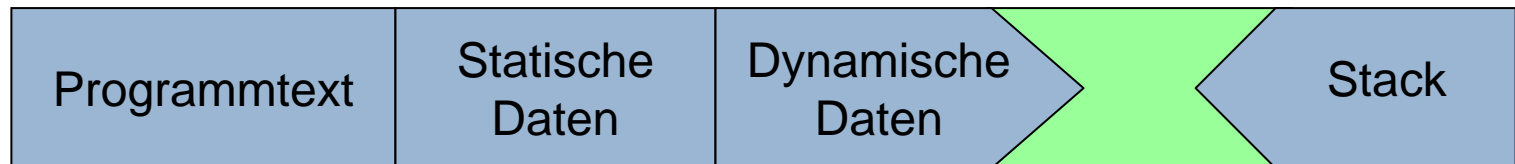
- Prozesstabelle
 - Liste aller Prozesse mit Zeiger auf ihre PCBs
 - „Tabelle“ ist hier eher ein konzeptioneller Begriff
 - Implementierung durch effizientere Datenstruktur
- Speicherbereiche
 - Programm-Text
 - Instruktionen, statische Daten, read-only
 - Programm-Daten (Heap)
 - Dynamische Daten
 - Stack
 - Für Prozeduraufrufe
 - BS-interne Daten
 - Rechte, User-Informationen, Kernel Stack

Prozessverwaltung



Prozesse & Adressräume

- MMUs ermöglichen getrennte Adressräume
 - Vorteil: Unumgänglich für Sicherheit
 - Nachteil: Inter-Prozess-Kommunikation komplizierter
- Normalerweise
 - Ein Prozess hat einen Adressraum
- Andere Möglichkeiten
 - Mehrere Prozesse mit einem Adressraum -> Threads



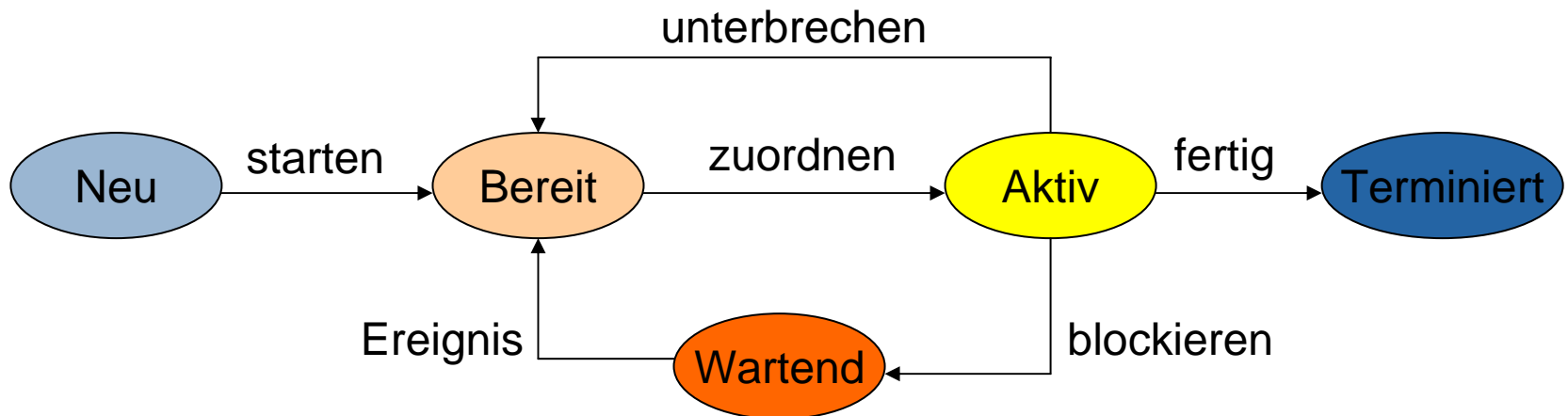
Niedrige Adresse

Hohe Adresse

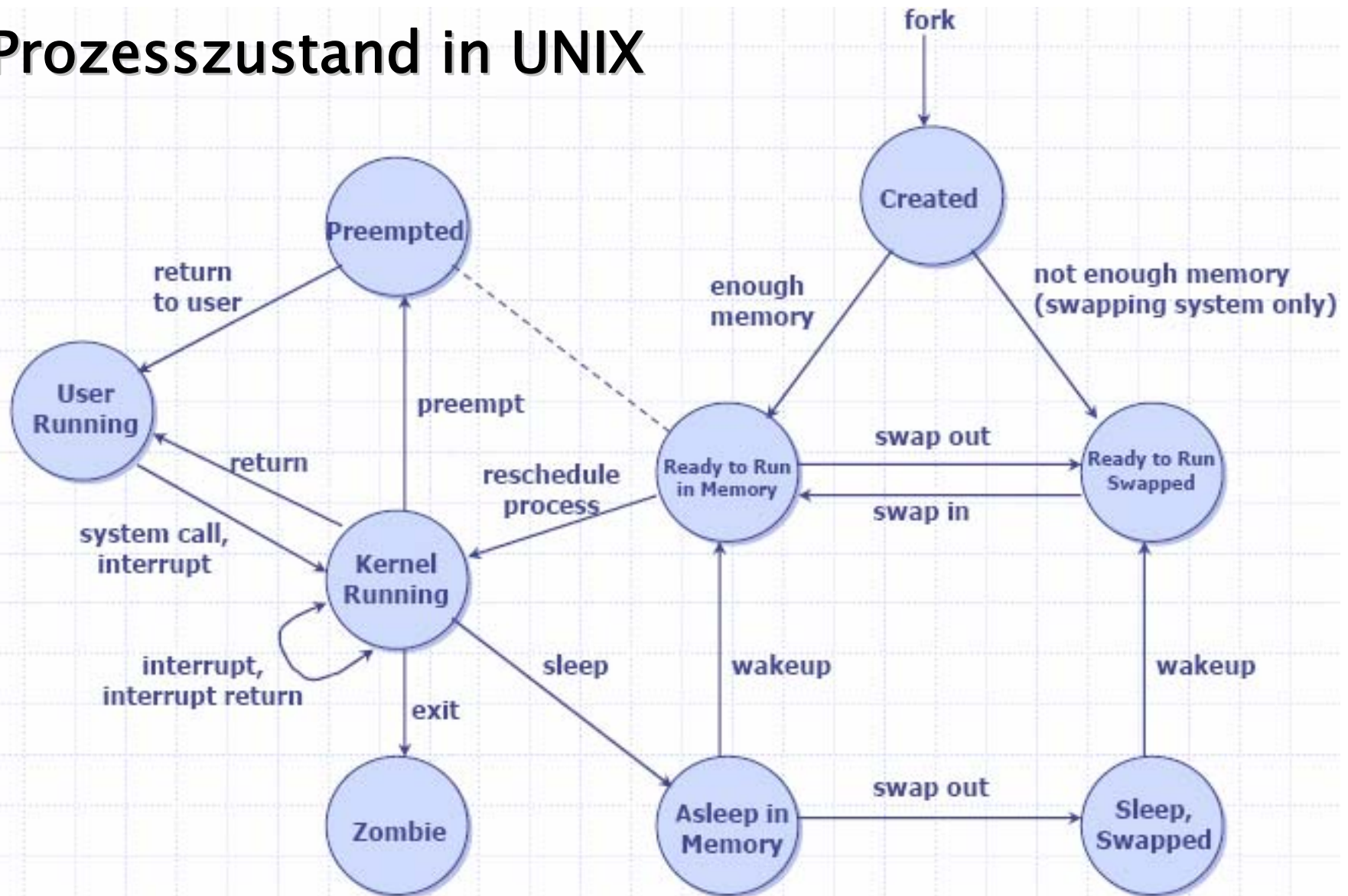
Prozesszustand

- Zustände eines Prozesses

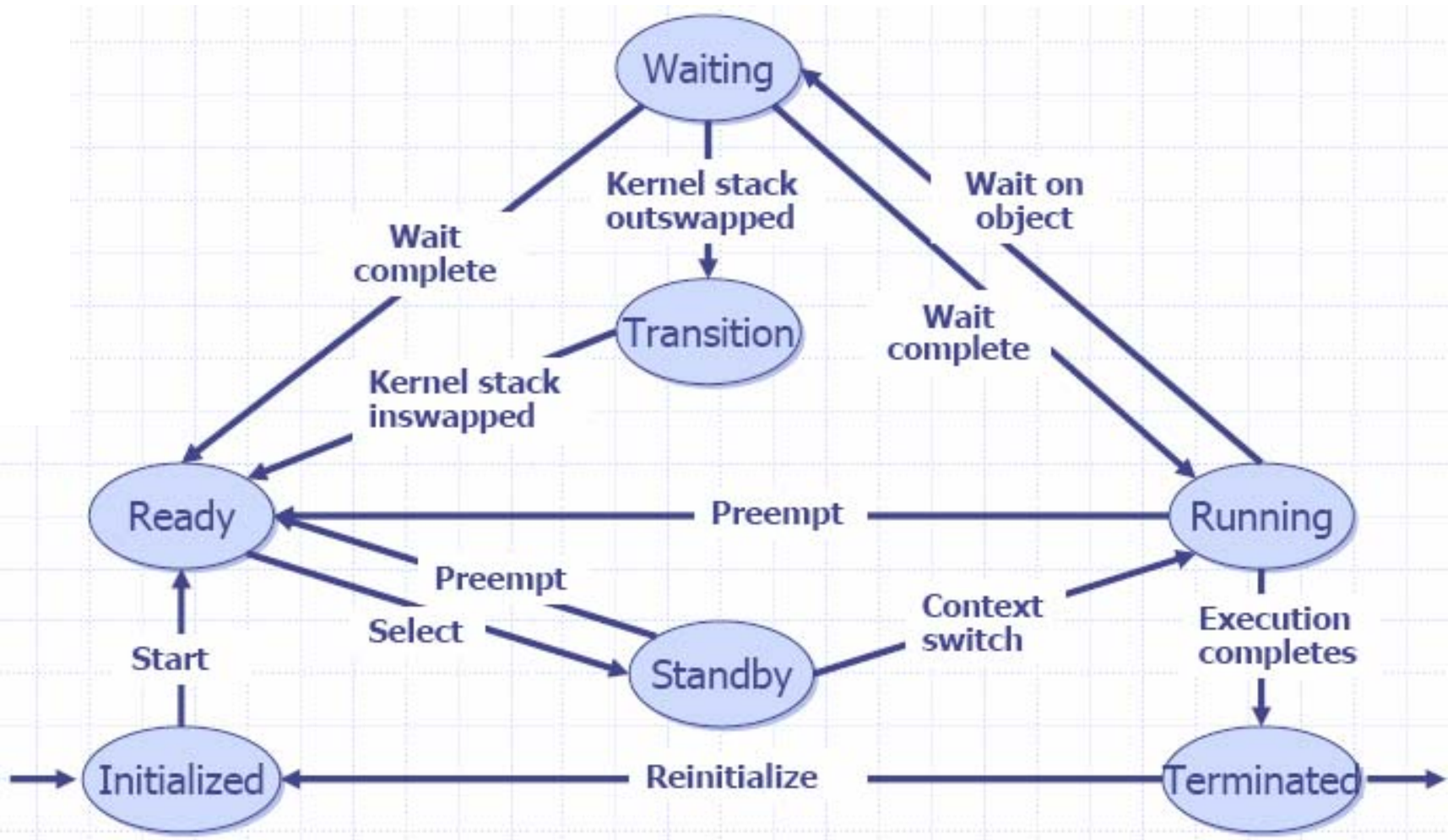
- Neu Aufgenommen im BS
- Bereit Kann CPU zugeordnet werden
- Aktiv Ist CPU zugeordnet
- Wartend Wartet auf ein Ereignis
- Terminiert Abgearbeitet, keine CPU mehr nötig



Prozesszustand in UNIX



Prozesszustand in Windows NT



Prozess-Erzeugung

- Wer erzeugt einen neuen Prozess?
 - Bootloader (erzeugt den `init` Prozess unter UNIX)
 - Booten des Systems (erzeugt Daemons)
 - DNS, Timer-Server, etc.
 - Login-Shell für Terminals
 - Systemaufruf durch einen anderen Prozess (`fork` unter UNIX)
 - Anfrage an das System von außen
 - Der Init Prozess kann neue Prozesse starten, wenn IP Pakete auf einem entsprechenden Port ankommen
- Zwei Methoden zur Erzeugung eines Prozesses
 - Erzeugen (Windows) oder Kopieren (UNIX)

Prozess-Erzeugung in UNIX

- Erzeugen eines Prozesses durch klonen
 - UNIX bietet System Call: `pid = fork()`
 - Erzeugt identische Kopie des Aufrufenden mit neuem Adressraum, falls ausreichend Ressourcen verfügbar sind
 - Im Kind-Prozeß ist `pid == 0`
 - Im Mutter-Prozeß ist `pid == 'ID des Kind-Proz.'`
 - Typischerweise folgt im Kind-Prozeß ein `exec()`, wodurch die Text- und Datensegmente ersetzt werden
 - Durch weitere `fork()`-Operationen entsteht eine Prozeßhierarchie

Prozess-Erzeugung in UNIX

```
int pid = fork( ); // Prozeß abspalten
if ( pid == -1 ) printf( "Fehler!!!" );
else if ( pid == 0 ) printf( "Kind" );
else printf( "Mutter, KindID=%d\n", pid);

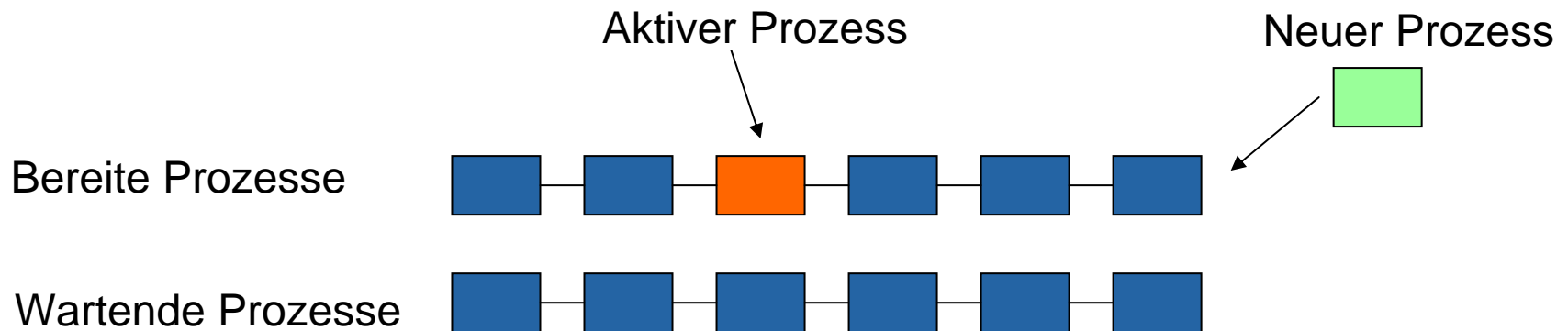
// blockiert bis ein Kind-Prozess terminiert
int wait( int *status);
// blockiert bis Kind-Prozess "pid" terminiert
int waitpid( int pid, int *status, int
             options );
```

Prozess-Terminierung

- Gründe für eine Prozess-Terminierung
 - Programm beendet (freiwillig)
 - Vom Programm erkannter Fehler (freiwillig)
 - Vom BS erkannter schwerer Fehler (unfreiwillig)
 - Beenden durch anderen Prozess (unfreiwillig)
- Prozess-Terminierung unter UNIX
 - Prozess erhält ein Signal
 - kill, segfault, etc.
 - Signale haben Standard-Behandlung
 - Meist Terminierung des Prozesses
 - Manche Signale können von der Anwendung abgefangen werden

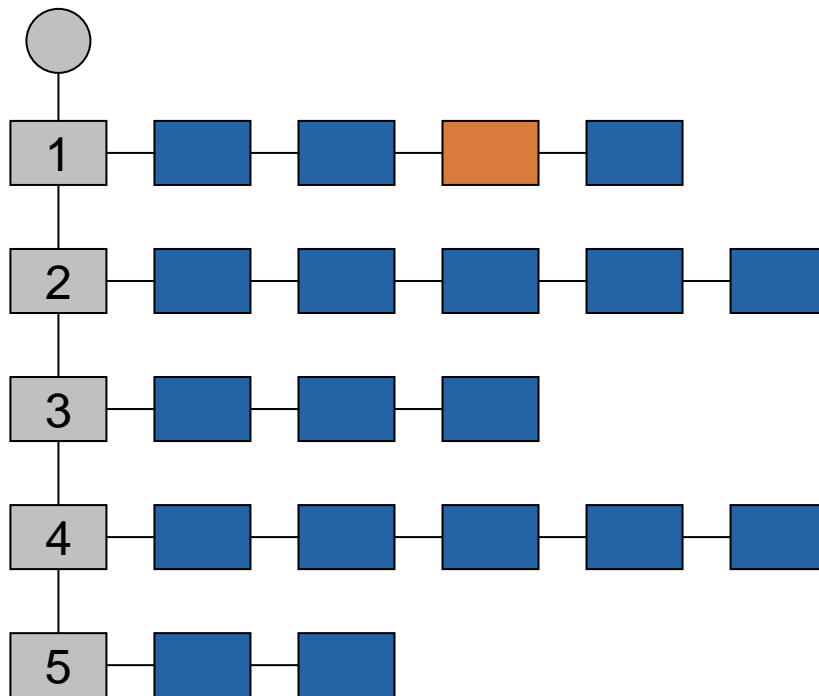
Effiziente Prozesstabellen

- Finden des nächsten aktiven Prozesses muss sehr effizient sein
 - Durchsuchen der Prozesstabelle ist nicht akzeptabel, denn Laufzeit ist $O(n)$
- Erste Idee: Sortieren nach Prozesszustand



Effiziente Prozesstabellen

- Berücksichtigung von Prioritäten durch 2-dimensionale Struktur



Leerlaufprozess

- Es gibt immer einen aktiven Prozess
 - Das vereinfacht die Scheduling Algorithmen
 - Was tun, wenn alle Prozesse warten?
- Der Leerlaufprozess ...
 - ... läuft immer
 - ... wartet nie
 - ... konsumiert überflüssige CPU Zeit
- Energiesparen
 - Mobile Rechner drosseln die CPU wenn sie nicht gebraucht wird
 - Das ist genau dann der Fall, wenn der Leerlaufprozess dauernd oder zumindest fast durchgehend läuft

Prozess-Umschaltung

- Gründe für eine Prozess-Umschaltung
 - Der Prozess muss warten und gibt die CPU ab
 - Beispielsweise `sleep(1000);`
 - Alternativ durch blockierenden Systemaufruf
 - Der Prozess hat seine Zeitscheibe aufgebraucht
 - Der Timer löst einen Interrupt auf
 - Der Kernel übernimmt und teilt die CPU neu zu
 - Ein Prozess mit höherer Priorität ist bereit oder wurde gerade gestartet
 - Dem geht ein Interrupt voraus
 - Während der Interrupt-Behandlung wird der höher priorisierte Prozess erzeugt oder wieder bereit

Prozess-Umschaltung

- **Automatisch**
 - Das System verfügt über Unterbrechungen
 - Timer löst einen Interrupt aus
 - Vorteil: Völlig transparent für den Prozess
 - Nachteil: Mit einem gewissen Overhead verbunden
- **Manuell**
 - Prozesse rufen sich regelmäßig gegenseitig auf
 - Windows 3.x: „kooperatives Multitasking“
 - Wird kaum noch verwendet
 - Macht heute nur Sinn in abgeschlossenen Systemen
 - D.h. ein Programmiererteam hat alle kooperierenden Prozesse entwickelt

Automatische Prozess-Umschaltung

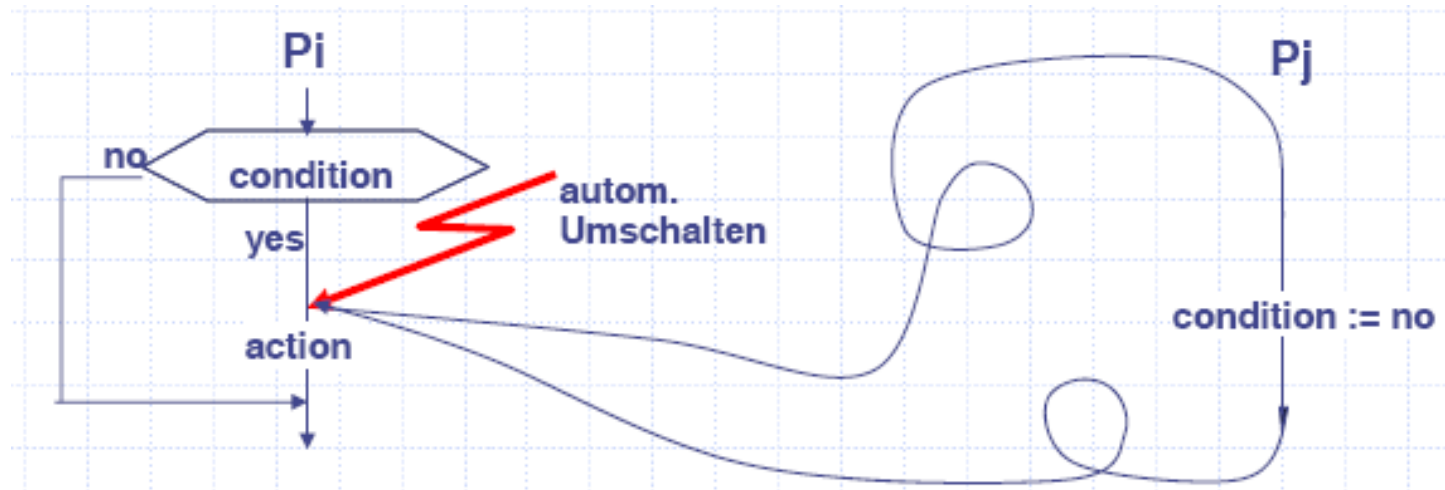
1. Interrupt wird durch Hardware ausgelöst
2. Hardware sichert Befehlszähler (PC)
3. Hardware holt neuen PC aus Interrupt-Vektor
4. Assemblerfunktion speichert Register
5. Assemblerfunktion setzt SP auf Kernel-Stack
6. C-Unterbrechungsroutine läuft (optional)
7. Scheduler sucht neuen aktiven Prozess
8. Rückkehr in die Assemblerfunktion
9. Laden der Register & Setzen des SP, PC, etc.
10. Sprung in den Code des jetzt aktiven Prozesses

Umschaltverhinderung

- Automatischen Umschalten kann jederzeit erfolgen
- Dadurch kann beispielsweise ein Umschalten ausgelöst werden, wenn gerade ein (freiwilliges) Umschalten im Gange ist. Dies kann zu Fehlern führen
- Während des Umschaltens muss verhindert werden, dass ein weiteres Umschalten ausgelöst wird
- Grundsätzlich kann es zu Fehlern kommen, wenn eine Kernoperation mitten in ihrer Ausführung durch eine andere Kernoperation unterbrochen wird, da Kernoperationen auf gemeinsame Datenstrukturen zugreifen
- Wenn das Umschalten zu jedem Zeitpunkt eintreten kann, dann ist eine beliebige verzahnte Ausführung von Prozessen und damit auch von Kernoperationen möglich

Probleme durch verzahnte Ausführung

- In Kernoperationen gibt es eine Vielzahl von Stellen, bei denen abhängig von einer Bedingung eine Aktion durchgeführt wird
- Es kann sein, dass zwischen der Abfrage der Bedingung und der Aktion ein Umschalten stattfindet und wo die Bedingung geändert wird



Kern als kritischer Abschnitt

- Kritische Abschnitte sind dann ungefährlich, wenn eine solche Verzahnung ausgeschlossen wird.
- Wenn ein Prozess einen kritischen Abschnitt betritt, darf kein weiterer Prozess einen dazu in potentiellm Konflikt stehenden kritischen Abschnitt betreten.
- Man spricht von einem **gegenseitigen Ausschluss (mutual exclusion)**.
- Im Betriebssystemkern müssten alle möglichen Konfliktstellen identifiziert und entsprechend geschützt werden.
- Da im Kern eine Vielzahl dieser kritischen Abschnitte vorhanden ist, kann man es sich einfach machen und den **gesamten Kern als kritischen Abschnitt** auffassen.
- Die Konsequenz ist, dass der **komplette Kern unter gegenseitigen Ausschluss** zu stellen ist.
- Es muss sichergestellt werden, dass Kernoperationen nicht verzahnt ablaufen können, sondern vollständig “an einem Stück” ausgeführt werden.

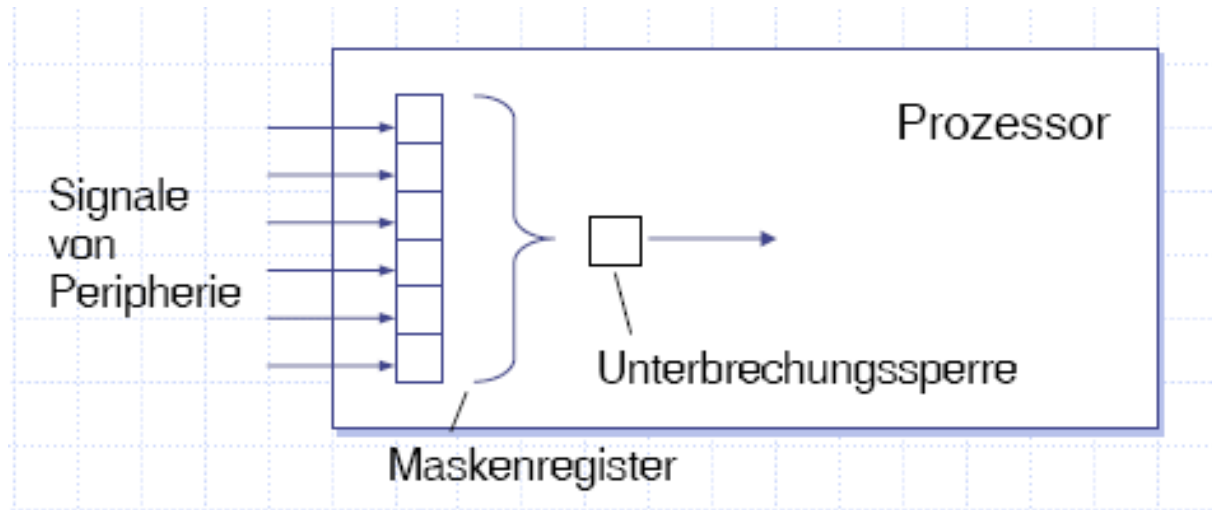
Realisierung des Kernausschlusses

- Kein Unterbrechungsmechanismus
–> keine Unterbrechung der Kernoperation
- Besitzt die CPU einen Unterbrechungsmechanismus, so kann für die Dauer der Kernoperation die Unterbrechungssperre gesetzt werden
- Dadurch haben wir das Problem auf den vorigen Fall zurückgeführt.
- Dies gelingt aber nur bei Einprozessormaschinen
 - Im Mehrprozessorsystem kann es trotz Unterbrechungsverbot zu einer verzahnten Ausführung von Kernoperationen kommen, wenn sie eben simultan auf zwei Prozessoren bearbeitet werden, deren Speicherzugriffe dann verzahnt ablaufen.
 - Für diesen Fall müssen wir den gegenseitigen Ausschluss am Kern durch eine sogenannte **Kernsperre** erzwingen.

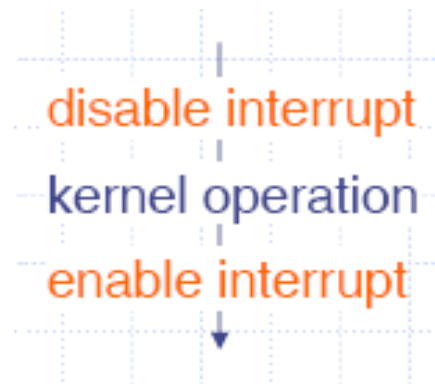
Kernausschluss

- Die Realisierung des Kernausschlusses ist davon abhängig, ob Unterbrechungen möglich sind und ob mehrere Prozessoren vorhanden sind.
- Daher haben wir vier Fälle zu unterscheiden:
 - Fall 1: Einprozessorsystem ohne Unterbrechungen
 - Fall 2: Einprozessorsystem mit Unterbrechungen
 - Fall 3: Mehrprozessorsystem ohne Unterbrechungen
 - Fall 4: Mehrprozessorsystem mit Unterbrechungen
- Der Fall 1 erfordert keinerlei spezielle Sicherungsmaßnahmen, da es keinen Anlass gibt, eine Kernoperation zu verlassen

Fall 2: Einprozessorsystem mit Unterbrechung



- Die Kernoperation wird dann durch ein *disable interrupt* und *enable interrupt* geklammert

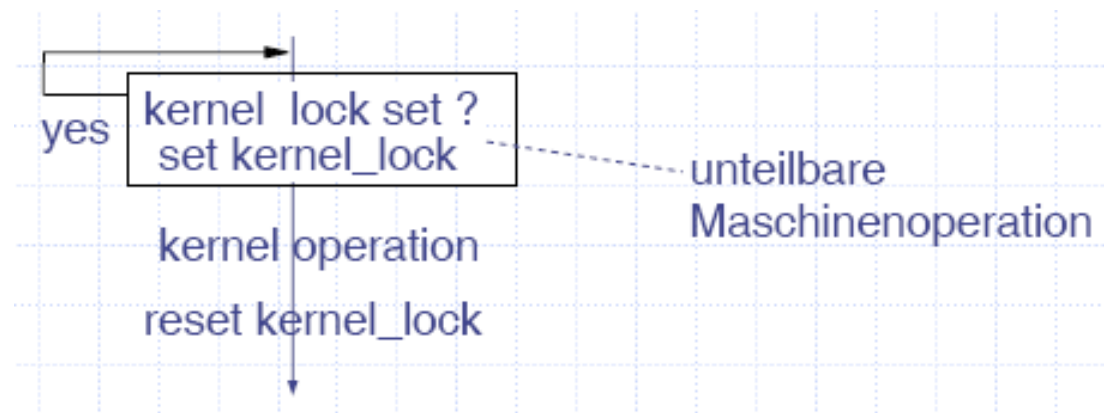


Fall 3: Mehrprozessorsystem ohne Unterbrechnung

- Prozessoren könnten parallel Kernoperationen ausführen
- Um den gegenseitigen Ausschluss von Kernoperationen zu gewährleisten, wurden Sperroperationen eingeführt
 - Die sind jedoch selbst wiederum kritische Abschnitte
- Um diese Rekursion aufzulösen, gibt es i.d.R. einen Maschinenbefehl, der in einer **unteilbaren Operation** den Wert einer Bedingungsvariable abfragt und sie gleichzeitig setzt
 - `test_and_set_lock(reg, x) := {load reg, x; x:=1}`
 - Unabhängig von ihrem Wert wird die Variable x auf 1 gesetzt
 - War $x=0$, so wird x auf 1 gesetzt, andernfalls bleibt der Wert unverändert bei 1

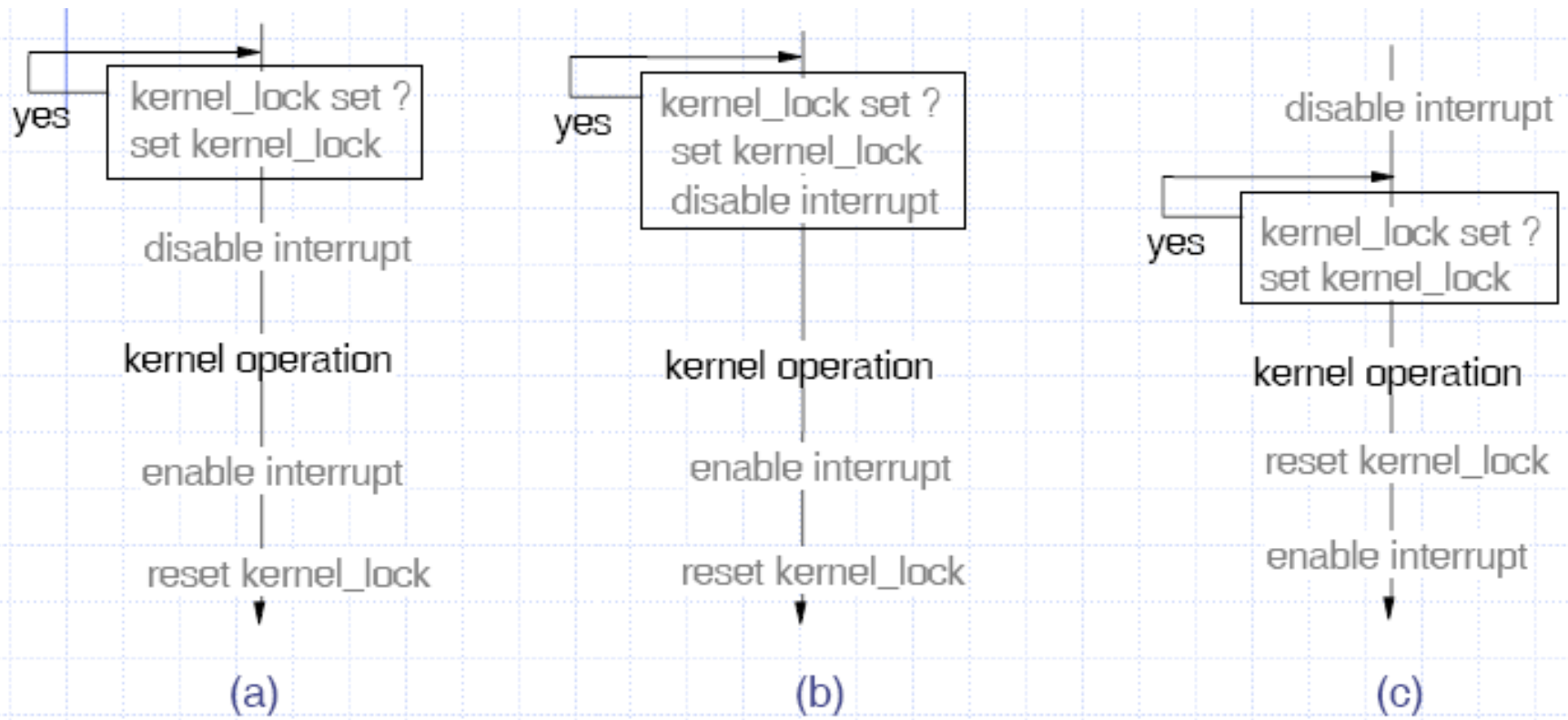
Fall 3: Mehrprozessorsystem ohne Unterbrechnung

- Lösung mit test-and-set-Befehl
- Ist die Kernsperre belegt, so wird in einer Schleife der Wert immer wieder abgefragt
 - Man nennt dies „aktives Warten“ (busy waiting) oder „spin lock“
- Aktives Warten ist zwar Verschwendung von Rechenkapazität, kann aber toleriert werden, da Kernoperationen relativ kurz sind



Fall 4: Mehrprozessorsystem mit Unterbrechungen

- Es müssen nun beide Techniken, d.h. Unterbrechungssperre und Kernsperre gemeinsam zum Einsatz kommen



Fall 4: Mehrprozessorsystem mit Unterbrechungen

- Lösung A
 - Eine Unterbrechungsbehandlung ist ebenfalls eine Kernoperation, für deren Durchführung also die Kernsperrung benötigt wird
 - Gibt es nun direkt nach dem Setzen der Kernsperrung eine Unterbrechung, so würde in der Unterbrechungsbehandlung vergeblich versucht, die Kernsperrung zu setzen
 - Der Prozess würde an dieser Stelle „hängenbleiben“.
- Lösung B
 - Ideal wäre deshalb eine Operation, die unteilbar sowohl die Kernsperrung als auch die Unterbrechungssperre setzt
 - Leider findet man solche Operationen bei keinem Prozessor
- Lösung C
 - Es muss daher zuerst die Unterbrechungssperre gesetzt werden und dann die Kernsperrung
 - Lösung C ist also korrekt

Softwarelösung für gegenseitigen Ausschluss

- Gegenseitiger Ausschluss ist auch ohne Hardware realisierbar
- Die auf Peterson (1981) zurückgehende Lösung sei für den Spezialfall von nur 2 Prozessen hier skizziert
 - Sie kann auf n Prozesse erweitert werden.
- Mit *interested_x* gibt der jeweilige Prozess x seine Eintrittsabsicht kund
- Die Variable *favored_process* wird nur wirksam, wenn beide Prozesse interessiert sind und bewirkt eine Entscheidung
 - In diesem Fall für den, der zuerst kommt

Softwarelösung für gegenseitigen Ausschluss

