Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - FB Informatik

Prozesse und Prozessverwaltung

- Einführung des Prozessmodells
- Datenstrukturen von Prozessen
- Multiprozesssysteme und Scheduling (Prozessumschaltung)

Terminologie

Programm

Ausführungsvorschrift für eine Maschine, i.A. eine lineare Liste von Instruktionen.

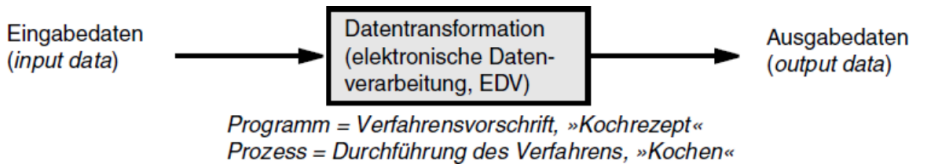
Prozessor

Physische (oder virtuelle) Ausführungseinheit einer Maschine (Rechner) um Instruktionen auszuführen. Maschineninstruktionen sind Daten- und Kontrollanweisungen, d.h. Berechnung und Kontrollflusssteuerung.

Prozess

Ein Orogramm in Ausführung, entweder im Vordergrund durch einen Prozessor, oder im Hintergrund durch einen Speicherzustand (inkl. Auslagerung auf einen Datenträger).

Terminologie



[bsdp]

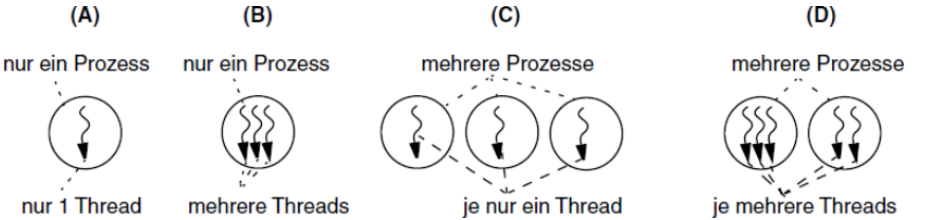
Abb. 1. Unterscheidung Programm zu Prozess

Allgemein gilt: kommt ein Programm zur Ausführung, dann wird ein neuer Prozess gestartet. Der Prozess belegt Hauptspeicherplatz, Prozessorregister, belegt Rechenzeit und nutzt die Peripherie. Ein bestimmtes Programm kann bei Bedarf auch mehrfach parallel ablaufen. Es liegen dann mehrere Prozesse vor, aber immer noch nur ein Programm.

Terminologie

Threads

Threads sind parallel ablaufende Aktivitäten innerhalb einer gemeinsamen Prozessumgebung. Der Code einzelner Threads ist Teil des bereits geladenen Programms. Jeder Thread gehört damit zu einem bestimmten Prozess.



[bsdp]

Abb. 2. Varianten der Software-Parallelität

Terminologie

Nebenläufigkeit

Nebenläufigkeit (concurrency) bedeutet, dass die Software mehrere Programmanweisungen gleichzeitig ausführen kann (abstrakte Parallelität). Parallelität im strikten Sinne bezieht sich hingegen auf die tatsächlich gleichzeitige Ausführung (konkrete Parallelität). Eine weniger strikte Definition der Parallelität schlie0t die Nebenläufigkeit ein, da dies Praxisgebrauch darstellt.

Task

Primär ist diese Bezeichnung bei eingebetteten Systemen (embedded systems) üblich und bezeichnet dort parallel ausgeführte Programmteile. Einige Betriebssysteme benutzen den Begriff Task hingegen als einen Oberbegriff, der Prozesse und Threads umfasst.

Terminologie

Job und Session

Darunter sind voneinander unabhängige und gegenseitig geschützte Anwendungen auf einer Rechneranlage zu verstehen. Eine derartige Anwendung besteht dabei aus einer Gruppe von Prozessen.

Echtzeitbetrieb

Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu vorbestimmten Zeitpunkten auftreten (DIN 44 300).

Man unterscheidet harte und weiche Echtzeitsysteme. Das Überschreiten von Zeitgrenzen (von Prozessen) wird im Softbetrieb toleriert, im Hartbetrieb wird diese als Systemfehler gewertet mit Anhalten der Rechneranlage.

Prozess

Ein Prozess *i* hat einen Zustand Σ_i. Ein Prozess in Ausführung ändert seinen Zustand.

- Der Prozesszustand setzt sich aus zwei Teilzuständen zusammen:

1. Kontrollzustand γ, gegeben durch den Befehlszähler;
2. Datenzustand σ, gegeben durch alle Prozessvariablen.

- Der Prozess ist ein Grundbaustein der Parallelverarbeitung mit einem Betriebssystem.
- Das Prozessmodell ist daher ein zentrales Konzept moderner Betriebssysteme. Ergänzt wird es durch das Thread-Modell, das für die Parallelisierung innerhalb von Applikationen die optimalere Lösung darstellt.

Prozess

- Ein Prozess kann sich in drei wesentlichen Zuständen befinden:

1. **Ready**: Rechenbereit, aber nicht in Ausführung durch einen Prozessor;
2. **Run**: In Ausführung durch einen Prozessor; wesentlich Berechnung;
3. **Wait**: Wartend (auf Ereignisse).

- Das Warten bedeutet i.A. Ein- und Ausgabe, Kommunikation oder die Verzögerung
- Es gibt zwei Arten um auf Ereignisse zu warten:

1. Durch wiederholte Abfrage (Polling);
2. Durch Prozessblockierung (Suspendierung) und Wiederaufnahme (Resume) beim Eintreten des Ereignisses (Interrupt).

Ereignisse

Man kann verschiedene Klassen von Ereignissen unterscheiden, die von Prozessen berarbeitet werden:

1. Daten oder Ereignisse von Kommunikationsgeräten (WIFI/WLAN, Bluetooth, USB, Scanner, Drucker "kein Papier");
2. Zeitablauf (Verzögerung, Timer);
3. Synchronisation zwischen Prozessen, z.B. Steuerung einer Warteschlange, Bereistellen von Daten zur Abholung usw.;
4. Wettbewerbsauflösung durch Sequenzialisierung, d.h., Nutzung geteilter Ressourcen.

Warten auf Ereignisse bedingt das Anhalten des Kontrollflusses eine Prozesses (Prozessblockierung), wenn auch beim Pollingbetrieb nur eine wiedeholende Schleife den Programmfortschritt blockiert.

Prozesszustände

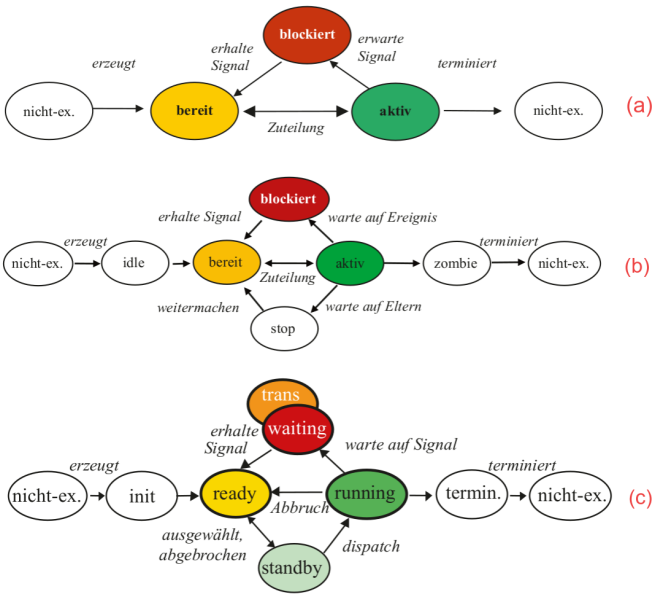


Abb. 3. (a) Allgemein Prozesszustände und Übergänge (b) Prozesszustände und Übergänge bei UNIX (c) Prozesszustände in Windows NT [bsbrause]

Prozesszustände

Die Zustände „bereit“ und „blockiert“ enthalten eine oder mehrere Warteschlangen (Listen), in die die Prozesse mit diesem Zustand eingetragen werden.

- Der bereit-Zustand ist besonders ausgezeichnet: Alle Prozesse, die Ereignisse erhalten und so entblockt werden, werden zunächst in die bereit-Liste (ready-queue) verschoben und erhalten dann in der Reihenfolge den Prozessor zur Ausführung.

Prozessmodell

- Das Prozessmodell ist ein zentrales Konzept der meisten Betriebssysteme.
  - Kerneigenschaft dabei ist, dass jeder Prozess virtuell den ganzen Rechner für sich alleine zur Verfügung hat, d.h., er kann den Adressraum frei belegen (abgesehen von reservierten Systembereichen) und kann die Prozessorregister nach Belieben benutzen.
- Das Prozessmodell realisiert durch die nebenläufige Ausführung mehrerer Prozesse den Mehrprogrammbetrieb.
  - In einem Einprozessorsystem wird eine Aufteilung der Rechenzeit auf mehrere ablaufwillige Prozesse durchgeführt.
  - Es findet dabei ein Zeitmultiplex des Prozessors statt.
  - Dies ergibt für den Benutzer die Illusion der echt parallelen Programmausführung, obwohl zu jedem Zeitpunkt nur immer gerade ein einziges Programm von der CPU abgearbeitet wird.
  - Auf Multiprozessorsystemen ist eine mehrheitlich gleichzeitige Ausführung von Programmen möglich.

Prozessverwaltung

- Die Programme und damit die Prozesse existieren nicht ewig, sondern werden irgendwann erzeugt, in eine Wartschlange eingereiht und auch beendet.
- Dabei verwalten die Prozesse aus Sicherheitsgründen sich nicht selbst, sondern das Einordnen in die Warteschlangen wird von einer besonderen Instanz des Betriebssystems, dem **Scheduler**, nach einer **Strategie** geplant.
- Bei einigen Betriebssystemen gibt es darüber hinaus eine eigene Instanz, den **Dispatcher**, der das eigentliche Überführen von einem Zustand in den nächsten bewirkt.

Im Unterschied zu dem Maschinencode werden die Zustandsdaten der Hardware (CPU, FPU, MMU), mit denen der Prozess arbeitet, als Prozesskontext bezeichnet.

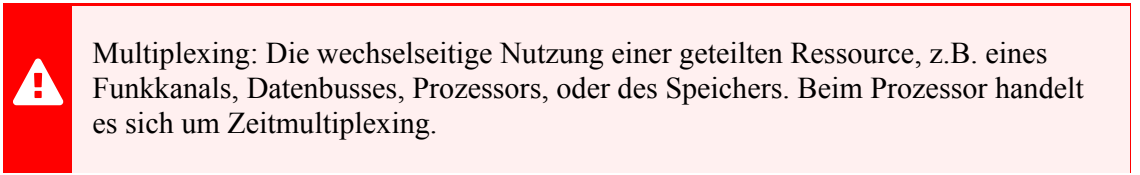
Prozessverwaltung

- Der Teil der Daten, der bei einem blockierten Prozess den letzten Zustand der CPU enthält und damit wie ein Abbild der CPU ist, kann als **virtueller Prozessor** angesehen werden und muss bei einer Umschaltung zu einem anderen Prozess bzw. Kontext (context switch) neu geladen werden.

Die verschiedenen Betriebssysteme differieren in der Zahl der Ereignisse, auf die gewartet werden kann, und der Anzahl und Typen von Warteschlangen, in denen gewartet werden kann. Sie unterscheiden sich auch darin, welche Strategien sie für das Erzeugen und Terminieren von Prozessen sowie die Zuteilung und Einordnung in Wartelisten vorsehen.

Prozessumschaltung

- Synonym für Scheduling oder Prozessmultiplexing



Multiplexing: Die wechselseitige Nutzung einer geteilten Ressource, z.B. eines Funkkanals, Datenbusses, Prozessors, oder des Speichers. Beim Prozessor handelt es sich um Zeitmultiplexing.

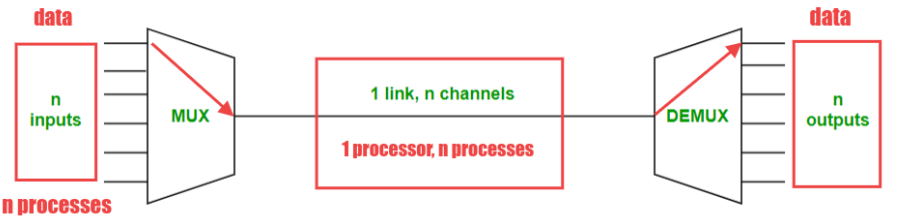


Abb. 4. Kanalmultiplexing (grün) im Vergleich zum Prozessmultiplexing (rot). Im Grunde werden Daten geschaltet.

Prozessumschaltung

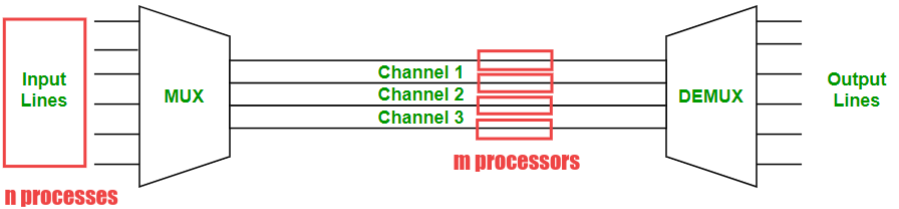


Abb. 5. Multiprozessorsysteme können als multiple Kommunikationskanäle verstanden werden.

- Die Prozessumschaltung bewirkt letztlich eine wechselseitige Verarbeitung von Datenströmen.

Prozessgruppen

Prozesse können einzeln gestartet oder in Gruppen zugeordnet werden.

- In Unix gibt es die Eltern-Kind Prozessgruppe, unter Windows nicht.
- Ein Kindprozess wird i.A. als Kopie des Elternprozesses erzeugt (Forking)

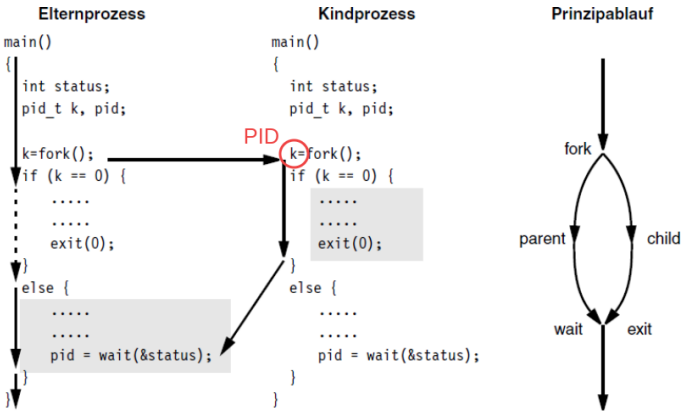
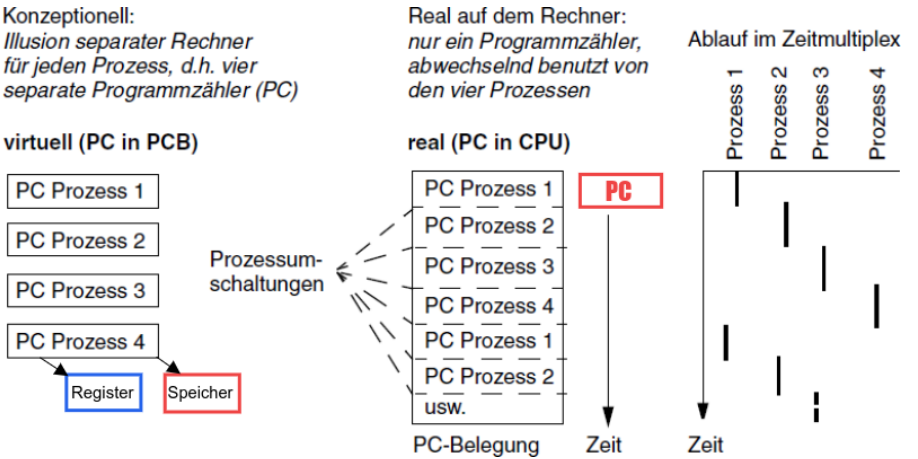


Abb. 6. Prozessverdoppelung bei fork(). PID: Process Identifier Number (numerischer ganzzahliger Wert der einen Prozess im System eindeutig identifiziert)

Prozessverwaltung



[bsdp]

Abb. 8. Mehrprogrammbetrieb mit vier Prozessen (Beispiel)

Prozessumschaltung

Prozesswechsel benötigt eien Verwaltungstabelle für Prozesse mit einem Process Control Block (PCB), kann noch durch erweiterte Tabellen überdeckt sein.

Ablauf für einen Prozesswechse:

1. Der laufende Prozess wird gestoppt (»eingefroren«): Damit übernimmt der Prozesswechselcode des Betriebssystems die Kontrolle.
2. Kontextsicherung des aktuellen Prozesses: Die aktuellen CPU-Registerinhalte werden aus der CPU in den PCB1 kopiert.
3. Auswahl nächster Prozess: Entsprechend einer Prozessorzuteilungsstrategie wird der nächste auszuführende Prozess bestimmt (sog. CPU-Scheduling).
4. Kontextwiederherstellung neuer Prozess: Die gesicherten CPU-Registerinhalte werden aus dem PCB2 in die CPU kopiert.
5. Neuer Prozess wird gestartet (»aufgetaut«): die Kontrolle wird neuem Prozess übergeben, der am letzten Unterbrechungspunkt fortfährt.

Prozessumschaltung

[bsdp]

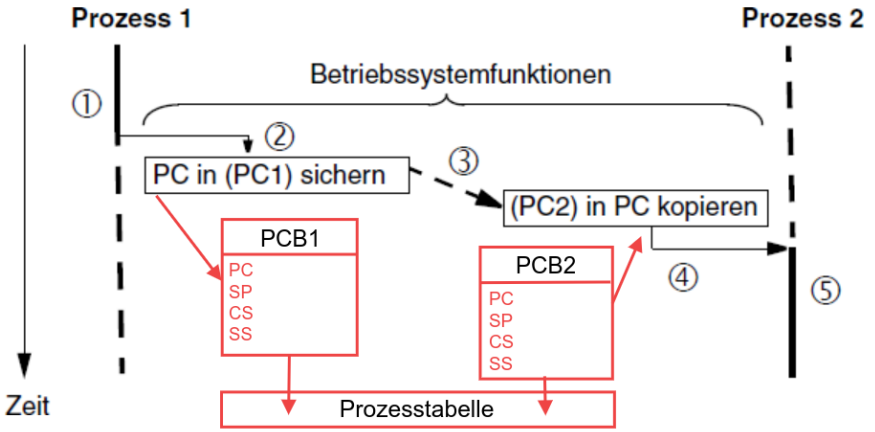


Abb. 9. Prozessumschaltung und Rolle des PCB in fünf Schritten (vereinfachtes Beispiel)

Vordergrund/Hintergrund System

Das einfachste Betriebssystem ist keines (im Sinne eines eigenständigen Programms). vor allem in Eingebetteten Systemen (Einplatinenrechner, Mikrokontroller) kommt das Einprogrammsystem zum Einsatz.

- Es gibt nur ein Programm welches monolithisch verschiedene Aufgaben ausführt:
  - Berechnung
  - Ein- und Ausgabe
  - Kommunikation
- Jetzt gibt es aber ein Problem: Wie soll ein Programm ohne Prozesse diese verschiedenen Aufgaben nebenläufig (quasi parallel) ausführen?
  - Das Warten auf Ereignisse müsste "parallel" im Polling geschehen
  - Wenn in Berechnung dann können keine Ereignisse überwacht und verarbeitet werden.
  - Besser: Hardwareunterstützung durch den Prozessor mit Interruptbetrieb

Vordergrund/Hintergrund System

Durch den Interruptbetrieb kann das Programm rechnen und beim Eintreten einen Ereignisses (Daten verfügbar, Ressource verfügbar, Fehler usw.) verzweigt der Prozessor in ein Unterpogramm welches das Ereignis bearbeitet.

- Kurios: Das Hauptprogramm ist ein Hintergrundprozess, das Interrupt Unterprogramm (Interrupthandler) ist ein Vordergrundprozess mit Kontextwechsel!
  - Grund: Interrupt Unterprogramme werden **präemptiv** aufgerufen, d.h. es wird das Hauptprogramm "sofort" unterbrochen und in das Unterpogramm verzweigt.
  - Der Interrupthandler hat die höhere Priorität und ist daher (wenn auch selten aktiv) der Vordergrundprozess.
- Aber: Präemption bedeutet Gefahr der Dateninkonsistenz. Beim (partiellen) Kontextwechsel werden nur einige Register gesichert, einige werden vom Hauptprogramm weiter verwendet und dürfen **nicht modifiziert** werden.
  - Daher sollte in den Interrupt Unterprogrammen nur das Ereignis (beim Gerät) bestätigt werden und das Hauptprogramm über Speichervariablen über das Ereignis informiert werden. Den Rest muss das Hauptprogramm bearbeiten.

Main IO Loop

- Wenn wir ein Einprogrammsystem betrachten dann muss es ungeachtet von Interruptbetrieb eine Hauptschleife geben die Berechnung und Ereignisverarbeitung "fair" aufteilt.
  - Ansonsten müsste das Hauptprogramm ständig nach Ereignissen fragen (Verteiltes Polling)

```
void setup() {
  Startcode
}
void loop () {
  if (Ereignis) handle(Ereignis)
  if (Berechnung) process(Berechnung)
}
setup();
while (1) loop();
```

Code 1. Zwei Funktionen (Arduino Style!): `setup` wird einmal zu Beginn, und `loop` "endlos" immer wieder aufgerufen.

Main IO Loop

Damit die Hauptschleife durchläuft muss die eigentliche Berechnung gestückelt sein. I.a. folgt die Berechnung auf Ereignisse.

```
int timeout=0;
struct list { int type; struct list *next };
void add(struct list* l, int ev) { add event ev to list l };
void loop () {
  int now=milliseconds();
  struct list events, eventp=events;
  if (Serial1.available) add(events,EV1);
  if (Serial2.available) add(events,EV2);
  if (timeout!=0 && timeout<now) { timeout=0; add(events,EV3); }
  ...
  while (eventp) {
    switch (eventp->type) {
      case EV1: procev1(); domore(); break;
      case EV2: procev2(); break;
      ...
    }
    eventp=eventp->next;
  }
}
```

Main IO Loop

- Event Handler (Ein- und Ausgabe usw.) haben nur kurze Laufzeit, können aber häufig aufgereufen werden. Die Laufzeit kann durch Monitoring abgeschätzt werden (Min., Mittelwert, Max.).
- Berechnungsfunktionen können beliebig lange laufen, die Laufzeit ist i.A. nicht über Monitoring abschätzbar.

Ein großes Problem (nicht nur in Echtzteitbetriebssystemen): Eine lange Laufzeit in einer Berechnung kann die Ereignsiverarbeitung derart einschränken dass es zu Verlusten von Ereignissen und Daten kommen kann. Es können sogar Hardware Geräte in einen blockierten/unbrauchbaren Zustand übergehen!

- Und noch schlimmer: Die meisten Programmiersprachen erlauben verschachtelte Funktionen. Eine lineare Liste von Instruktionen können unterbrochen werden, eine tief aufgerufene Funktion nicht ohne weiteres.

Main IO Loop

Es gibt zwei Lösungen:

1. Präemptives Zeitscheibenverfahren sofern es möglich ist aus Funktionen heraus zu springen und später zurück zukehren ("Long Jumps");
2. Transformation einer ganzen Berechung (funktionale/prozedural) in eine lineare Liste von abstrakten Instruktionen die von einer Virtuellen Maschine sukzessive verarbeitet werden kann.

Main IO Loop

Eine einfache VM

> Jedes zusammenhängende (atomare) Syntaxelement einer fiktiven Programmiersprache, sagen wir Basic, wird einen Sequenz von Tokens (Symbolen) transformiert.

```
a=1             LHS VAR a 1 EOL
b=2             LHS VAR b 2 EOL
x=3             LHS VAR x 3 EOL
for i = 1 to 10 FOR VAR i 1 10 EOL
  x=x+i+a+b     LHS VAR ADD VAR x ADD VAR a ADD VAR b
next i          NEXT i
print x         PRINT VAR X
```

Bsp. 1. Transformation eines Programms in eine lineare Folge von Symbolen

Eine einfache VM

- Die einzelnen Tokens können nun linear von der VM abgearbeitet werden und jederzeit unterbrochen werden: **Mikrostepping**.
- Die Tokens können direkt in einem Speicherarray abgelegt (kompiliert) werden

```
// VM state
int pc; int sp; uint16_t mem[SIZE]; // .....
// VM Execution Loop
int vmloop(steps) {
  for (int step=0;step<steps;step++) {
    int next=mem[pc++];
    switch (next) {
      case LHS:    xassign(); break;
      case VAR:    xvar();    break;
      case PRINT:  xprint();  break;
      case DELAY:  xdelay();  return -1;
      case END:    xend();    return 0;
      ...
    }
    if (interrupt) return 1;
  }
  return 1;
}
```

Bsp. 2. VM Execution Loop

```
// Interrupt Handler
int interrupt=0,
    pendingtokens=0,
    runnable=0;
void handleIRQn() { interrupt++; }
// IO Loop
void loop () {
  if (interrupt) handle interrupts;
  interrupt=0;
  if (events) handle events;
  if (pendingtokens && runnable) {
    status=vmloop(STEPSMAX);
    if (status!=0) pendingtokens=1;
    else           pendingtokens=0;
    if (status>0)  runnable=1;
    else           runnable=0;
  }
  if (newcode) {
    tokenize(newcode);
    pendingtokens=1;
    runnable=1;
  }
  ...
}
while (1) loop();
```

Bsp. 3. Einbindung einer VM Execution Loop in eine IO Loop

Prozesserzeugung und Terminierung

Prozessstart

In sehr einfachen Systemen, z.B. in vielen eingebetteten Systemen, existieren alle Prozesse von Anfang an und laufen ewig, d.h., solange das System läuft. Auf Universalrechnern (general-purpose computers) ist es hingegen notwendig, Prozesse bei Bedarf zu erzeugen und gelegentlich wieder zu terminieren. Auslösende Ereignisse für die Prozess- bzw. Thread-Erzeugung können dabei sein:

- Systemstart (Initialisierung)
- Systemdienstaufruf zur Prozesserzeugung durch irgendeinen laufenden Prozess
- Benutzeranforderung zum Starten eines neuen Prozesses (Applikationsstart)
- Auslösung eines Stapelauftrags (batch job)

Prozessstart

Es gibt drei grundlegende Möglichkeiten für den Start eines neuen Prozesses:

a. Prozessverkettung (chaining): Der laufende Prozess startet einen neuen Prozess und terminiert sich damit selbst. Der Code des neuen Prozesses ist unabhängig (z.B. in separater ausführbarer Datei vorliegend).
b. Prozessvergabelung (forking): Der laufende Prozess startet einen neuen Prozess, läuft selbst aber weiter. Der Code beider Prozesse ist gemeinsam.
c. Prozesserzeugung (creation): Der laufende Prozess startet einen unabhängigen neuen Prozess. Der Code beider Prozesse ist unabhängig (z.B. in zwei separaten ausführbaren Dateien vorliegend).

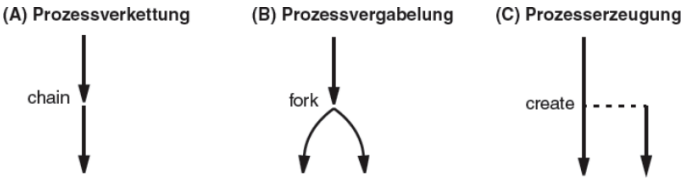


Abb. 10. Prozessstartformen: (a) Prozessverkettung, Sequenz (b) Prozessvergabelung, Forking (c) Unabhängige Prozesserzeugung, Create

Prozessbeendigung

Neben der Erzeugung interessiert auch die Terminierung von Prozessen. Übliche Formen sind:

- Normale Beendigung (freiwillig)
- Vorzeitige Beendigung bei einem durch den Prozess selbst erkannten Fehler (freiwillig)
- Vorzeitige Beendigung bei einem katastrophalen Fehler, erkannt durch das System (unfreiwillig)
- Terminierung durch einen anderen Prozess (unfreiwillig)

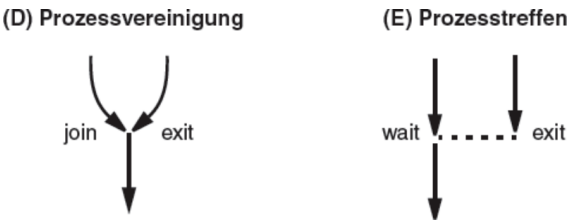


Abb. 11. Prozessvereinigungsformen: Prozesse können auf andere warten.

Prozessverwaltung: API

(a)

	Unix-Prozesse	POSIX-Threads	Windows-Prozesse	Windows-Threads
chain	`exec()`	–	–	–
fork	`fork()`	–	–	–
create	–	`pthread_create()`	`CreateProcess()`	`CreateThread()`

(b)

	Unix-Prozesse	POSIX-Threads	Windows-Prozesse	Windows-Threads
Vererbung	`fork()`	–	`CreateProcess()`	–
Interprozess-kommunikation	`popen()`	–	–	–
Initialpara-meter	–	`pthread_create()`	–	`CreateThread()`

(c)

	Unix-Prozesse	POSIX-Threads	Windows-Prozesse	Windows-Threads
join	`wait()`, `waitpid()`	`pthread_join()`	–	–
wait	–	–	`WaitForSingleObject()`	`WaitForSingleObject()`

Abb. 12. (a) Systemdienstaufrufe für Prozess-/Thread-Start (b) Weitergabe von Daten an neue Prozesse/Threads (c) Systemdienstaufrufe für die Prozess-/Thread-Vereinigung

Prozessorzuteilungsstrategien

Für die Ausführung paralleler Prozesse wäre idealerweise pro Prozess ein eigener Prozessor nötig. Da dies technisch nicht sinnvoll ist und nicht skalierbar ist (oder sehr aufwendig ist), hat man Verfahren entwickelt, die es erlauben, mehrere Prozesse quasi gleichzeitig auf dem gleichen Prozessor im Zeitmultiplexverfahren auszuführen.

Aber wie sollen die Prozesse der Ressource Prozessor zugeteilt werden, und wie lange sollen oder dürfen sie den Prozessor belegen? Reicht kooperatives Scheduling (Prozesse geben freiwillig die Ressource zurück, yielding), oder muss das Betriebssystem dieses erzwingen?

- Die Prozesszuteilung erfolgt nach einer Strategie und einem Algorithmus.
- Die Prozesszuteilungsstrategie gilt gleichermaßen für voll Prozesse und dür Subprozesse (Threads).

Prozessorzuteilungsstrategien

[bsdp]

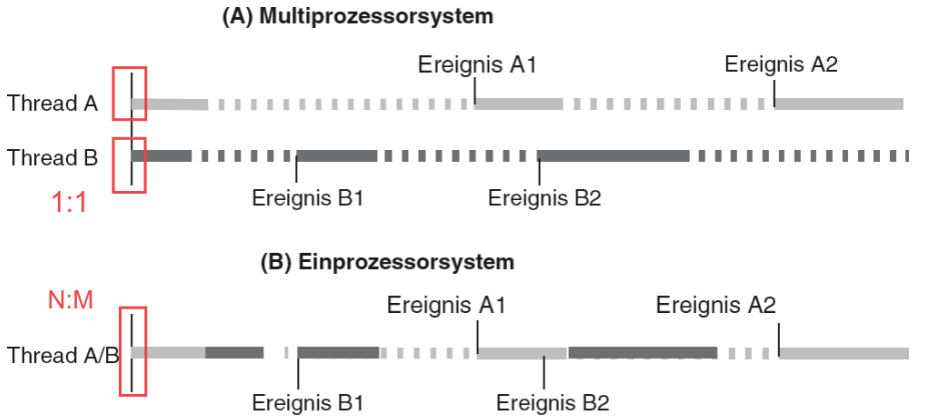


Abb. 13. Vergleich echte 1:1 Prallelitäz versa "simulierte" N:M Parellelität durch Zeitmultiplexing

Welchen Nachteil hat die 1:1 Zeuteilung (von der nicht vorhandenen Skalierbarkeit abgesehn)?

Prozessorzuteilungsstrategien

- Ein wichtiger Faktor ist die Auslastung einer Rechneranlage, die wie bei einer Heizungsanlage 100% betragen sollte (d.h. durchgehender Rechenbetrieb).
  - Bei einer festen 1:1 Zuteilung würden Prozessoren beim Warten von Prozessen auf Ereignisse untätig sein - keine Auslastung von 100%!

Wie können nun *N* Prozesse auf *M* Prozessoren verteilt werden?

Priorität & Präemption

- Die Auswahl kann von der Wichtigkeit von Prozessen beeinflusst werden, d.h. die **Priorität**, eine Ganzzahl.
  - Je größer die Priorität, desto eher könnte der Prozess ausgewählt werden oder
  - ein Prozess niedrigere Priorität könnte beim Bereitsein eines Prozesse höherer Priorität unterbrochen werden, d.h. es findet **Präemption** statt.
- Pritoritäten haben Einfluss auf das Zustandsmodell von Prozessen

Priorität & Präemption

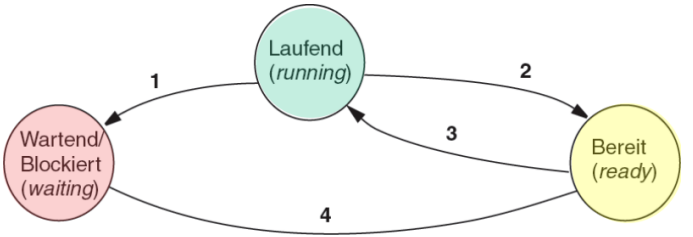


Abb. 14. Grundmodell mit drei Zuständen

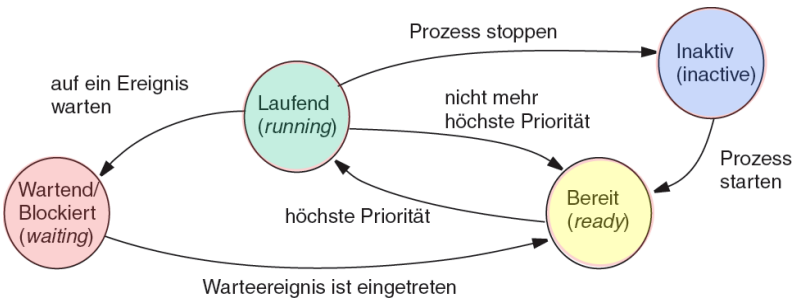


Abb. 15. Erweitertes Modell der Prozesszustände vier Zuständen und Priorität

Warteschlangen

- Für die Prozesszuteilung und dem Prozessscheduling gibt es mehrere Warteschlangen, d.h. im i.A. eine Warteschalnge pro Prozesszustand:
  - Bereit
  - Laufend
  - Blockiert
  - ...

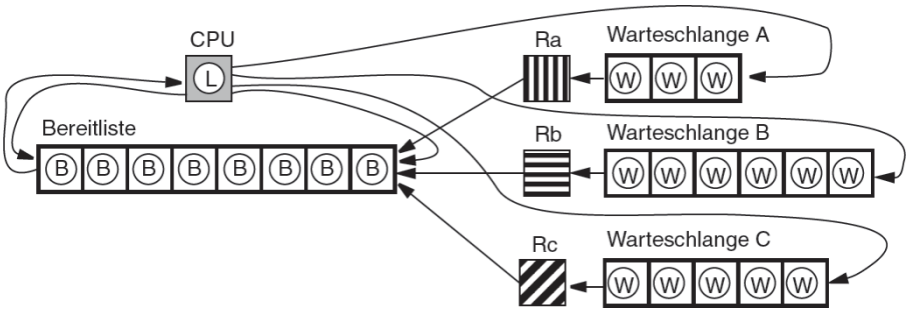


Abb. 16. Konzeptionelle Prozessverwaltung (Prozessorganisation)

Warteschlangen

- Ein Prozess ist entweder der CPU zugeteilt oder in einer der Ressourcen-Warteschlangen
  - Es findet ein Wechsel der Prozesse zwischen den Warteschlangen je nach Prozesszustand statt.
- Die Prozessorzuteilungsstrategie (CPU scheduling) legt fest, welcher Prozess als nächster die CPU zugeteilt erhält.
  - Der ausgewählte Prozess wird aus der Bereitliste entfernt und der CPU zugeordnet.
- Die Reihenfolge der Prozesse in der Bereitliste wird hierbei strategieabhängig unterschiedlich berücksichtigt.

  - Gibt ein laufender Prozess freiwillig die CPU ab, so wechselt er in die Bereitliste an die vorderste Position.
  - Wird einem Prozess die CPU entzogen, weil er sein Zeitquantum erschöpft hat (Präemption), dann wechselt er in die Bereitliste an die hinterste Position seiner Priorität.
  - Wenn der laufende Prozess auf eine Ressource/ein Ereignis zu warten beginnt, so wechselt er in die zugehörige Warteschlange.

- Welcher Prozess aus einer Ressourcen-Warteschlange die frei werdende Ressource erhält, ist eine Frage der E/A-Zuteilungsstrategie (I/O scheduling).
  - Diese kann **ressourcenabhängig unterschiedlich sein**.

Wie sollen nun Prozesse aus der Bereitwarteschlange der CPU zugeteilt werden? Was sind die Ziele?

Prozesszuteilung

Man unterscheidet zunächst:

1. Process based scheduling: Findet die Prozessorzuteilung nur zu ganzen Prozessen statt, so kennt der Systemkern keine Threads (single-threaded process). Dies bedingt ein Multithreading vollständig im Benutzermodus (user level threads).

2. Thread based scheduling: Findet die Prozessorzuteilung zu einzelnen Threads statt, so sind unterschiedliche Varianten des Multithreading möglich. Sowohl ein Multithreading auf der Benutzerebene als auch die direkte Nutzung von KL-Threads (kernel level threads) sowie verschiedene Zwischenformen sind denkbar.

Prozesszuteilung

Es gibt wesentlich zwei Arten von Prozessausführung (auch abwechselnd):

Betrachtet man typische Prozessabläufe, so kann man zwei wichtige Grundmuster (pattern) unterscheiden (siehe auch Abb. 4–31):

1. CPU-lastig (CPU-bound): Die Aktivität nutzt viel Rechenzeit und wartet im Vergleich dazu selten auf die Ein-/Ausgabe.
2. E/A-lastig (I/O-bound): Die Aktivität rechnet wenig, wartet hauptsächlich auf die Ein-/Ausgabe (Input/Output), d.h. nutzt Peripherie.

Generell können Wartezeiten auf die Ein-/Ausgabe für die Ausführung parallel laufender Berechnungen genutzt werden. Besonders lohnend ist dies für »E/A-lastige« Aktivitäten. Grundprinzip der asynchronen E/A, z.B. in virtuellen Maschinen.

Prozesszuteilung

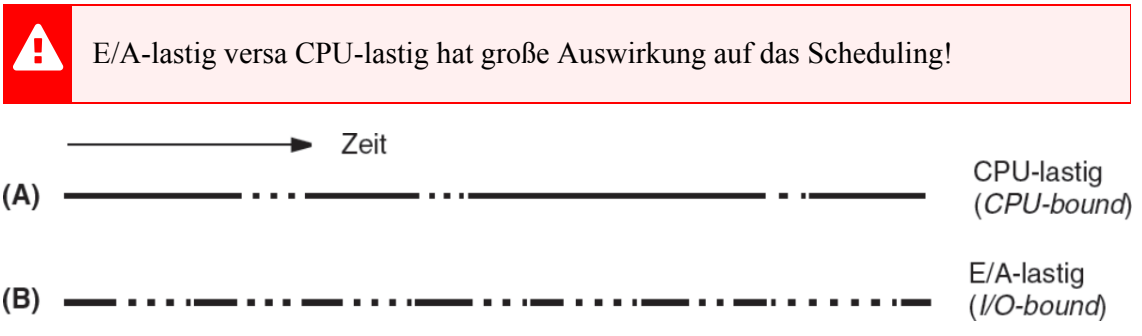


Abb. 17. Prozessablaufmuster (E/A- bzw. CPU-lastig)

- Bei einer reinen Berechnung spielen Verzögerung i.A. ein nicht so große Rolle als wie bei der Verarbeitung von E/A Ereignissen!

Optimierungsziele

Für eine optimale Zuteilung der CPU zu rechenwilligen Prozessen wurde eine Vielzahl an Strategien entwickelt. Diese Vielfalt kommt daher, dass je nach verfolgten Optimierungszielen unterschiedliche Lösungen optimal sind. Aus Sicht des Anwenders können folgende Ziele wichtig sein:

1. Durchlaufzeit (turnaround time): Gesamtzeit von Prozessstart bis Prozessbeendigung
2. Antwortzeit (response time): Zeit zwischen einer Eingabe und der Reaktion des Systems darauf
3. Endtermin (deadline): Zeitpunkt, zu dem eine vorgegebene Aktion erfolgt sein muss

> Aus Sicht eines optimalen Ressourceneinsatzes können weitere Ziele interessant sein, wie Vorhersagbarkeit, Durchsatz (Anzahl erledigter Aufträge pro Zeiteinheit) und Prozessorauslastung.

Verdrängende und nicht verdrängende Zuteilungsstrategien

- Eine Zuteilungsstrategie wählt aus der Menge aller Prozesse, die sich im »Bereit«-Zustand befinden, einen bestimmten Prozess aus. Zuteilungsstrategien unterscheiden sich durch das maßgebende Auswahlkriterium.

Wichtig ist dabei, zu verstehen, dass ein zum Auswahlzeitpunkt im Zustand »Laufend« befindlicher Prozess an dieser Auswahl ebenfalls teilnimmt.

- Damit kann der Auswahlentscheid auch lauten, dass kein Prozesswechsel stattfindet, sondern dem gleichen Prozess weiter die CPU zugeteilt wird.
- Eine weitere spezielle Situation liegt vor, wenn der laufende Prozess in den Zustand »Wartend« gewechselt hat.
  - In diesem Fall ist das Ziel der Auswahl auf jeden Fall die Zuteilung der CPU zu einem neuen Prozess.
  - Eine solche Auswahl eines Prozesses, der als nächster laufen soll, wird als **Neuzuteilung** bezeichnet (rescheduling). Eine Neuzuteilung kann auch die Auswahl des bereits laufenden Prozesses bedeuten.

Zuteilungszeitpunkte

1. Ein Prozess wechselt vom Zustand »Laufend« in den Zustand »Wartend« (dadurch wird die CPU frei und kann neu zugeteilt werden).
2. Ein Prozess wechselt vom Zustand »Wartend« in den Zustand »Bereit« (dadurch könnte eine Prioritätsregel verlangen, dass dieser Prozess Vorrang vor dem laufenden Prozess erhält).
3. Ein Prozess wechselt vom Zustand »Laufend« in den Zustand »Bereit« (dies könnte nach Ablauf einer bestimmten Zeit von einer Zeitquotenregel erzwungen werden).
4. Ein Prozess wechselt von einem der Zustände »Laufend«/»Bereit«/»Wartend« in den Zustand »Inaktiv« (dies wäre eine freiwillige oder erzwungene Prozessbeendigung).
5. Ein Prozess wechselt vom Zustand »Inaktiv« in den Zustand »Bereit« (dies trifft für einen neu gestarteten Prozess zu, der durch eine Prioritätsregel den Vorrang vor einem laufenden Prozess erhalten könnte).

Zuteilungszeitpunkte

Eine Zuteilungsstrategie gilt als nicht verdrängend (nonpreemptive), wenn die Neuzuteilung nur zum 1. und 4. Zeitpunkt stattfindet. Eine nicht verdrängende Neuzuteilung hat die Eigenschaft, dass sie kooperatives Verhalten der laufenden Prozesse erwartet. Beansprucht ein laufender Prozess für sehr lange Zeit fortlaufend immer die CPU, so haben ablaufbereite Prozesse keine Möglichkeit, dies zu ändern. Dies gilt nicht für verdrängende (preemptive) Zuteilungsstrategien, da bei ihnen eine Neuzuteilung zu allen fünf oben aufgeführten Zeitpunkten stattfindet.

FIFO Strategie

- Die FIFO-Strategie (**First In First Out**) ist auch unter dem Namen FCFS (First Come First Served) bekannt. Die Reihenfolge der Prozessausführung richtet sich streng nach der Reihenfolge, in der die Prozesse ablaufbereit wurden (d.h. Wechsel in den Zustand »Bereit«).
- Es handelt sich um eine nicht verdrängende Strategie, sodass neu ablaufbereite Prozesse keine Neuzuteilung zur Folge haben.
- Man bezeichnet dies auch als kooperative Zuteilung, da eine Neuzuteilung erst möglich wird, wenn der laufende Prozess die CPU freiwillig freigibt.
- Diese einfache Strategie erlaubt einen minimalen Verwaltungsaufwand. Damit eine Neuzuteilung des Prozessors stattfinden kann, muss der laufende Prozess entweder zu warten beginnen oder seine Ausführung terminieren.
- Alle Prozesse besitzen die gleiche Priorität. Dieses Verfahren funktioniert nur so lange, bis ein Prozess die CPU nicht mehr abgibt.
- Die Reaktionszeiten hängen zudem direkt von den durch die einzelnen Prozesse bedingten Laufzeiten ab (sog. Konvoieffekt).

FIFO Strategie

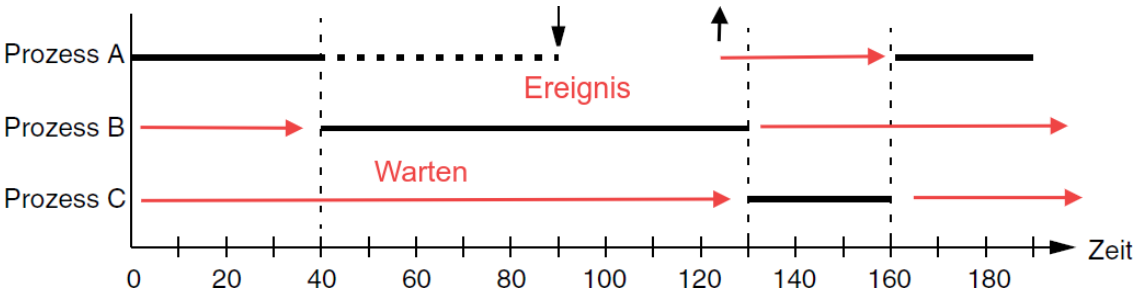


Abb. 18. Beispielszenario mit FIFO-Strategie

1. Ein Prozess A startet zum Zeitpunkt 0 und läuft für 40 Zeiteinheiten. Dann wartet er auf eine Ein-/Ausgabe, die nach 50 Zeiteinheiten abgeschlossen ist. Dadurch wird er wieder ablaufbereit und belegt seine restlichen 30 Zeiteinheiten.
2. Die Wartezeit (hier 50) für den Prozess A startet von dem Zeitpunkt an, wenn der Prozess blockiert, weil ein Prozess eine Ein-/Ausgabe anfordert und dann für die Dauer der Ein-/Ausgabe warten muss. Die Verzögerung (hier 40) startet ab dem Eintreten des Eireignisses.
3. Prozess B startet nach 20 Zeiteinheiten und will dann die CPU für 90 Zeiteinheiten belegen.
4. Prozess C startet nach 30 Zeiteinheiten und benötigt ab dann die CPU für 30 Zeiteinheiten.

SJF Strategie

Bei der nicht verdrängenden Strategie SJF (**Shortest Job First**), die auch unter dem Namen SPN (Shortest Process Next) fungiert, wird bei einer Neuzuteilung derjenige Prozess ausgewählt, der den kleinsten erwarteten Rechenzeitbedarf hat.

- Diese Strategie lässt sich naturgemäß nur anwenden, wenn Angaben über den Rechenzeitbedarf zum Entscheidungszeitpunkt dem System zugänglich sind.
  - Der Benutzer kann Angaben zur Laufzeit machen
  - Aaus der Vergangenheit kann eine Vorhersage gemacht werden (durch Messung). Besonders bei E/A-lastigen Prozessen kann das System repräsentative Daten über den vergangenen Rechenzeitbedarf sammeln. Diese Daten ermöglichen dann eine Hochrechnung, deren Qualität letztlich durch die Vorhersagbarkeit des zukünftigen Prozessverhaltens bestimmt wird.

SJF Strategie

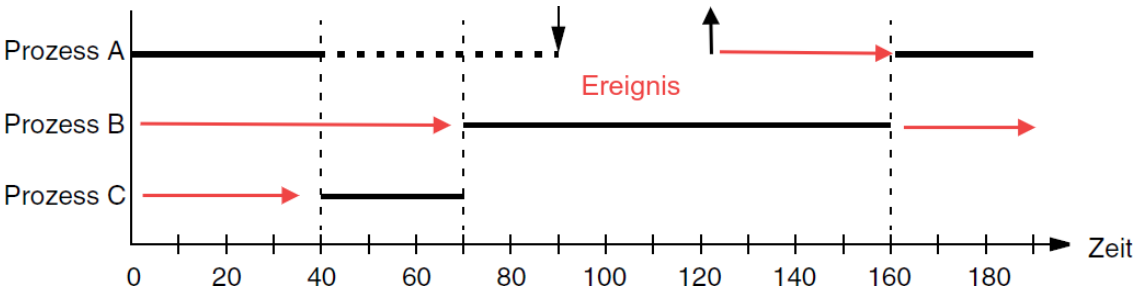


Abb. 19. Beispielszenario mit SJF-Strategie

E/A Reaktionszeiten können auch hier schlecht sein (lange Verzögerung)

SRT Strategie

Die Strategie SRT (**Shortest Remaining Time**) stellt eine Abwandlung des SJF-Verfahrens dar. Bei einer Neuzuteilung wird derjenige Prozess ausgewählt, der den kleinsten verbleibenden Rechenzeitbedarf hat.

- Da es sich um eine verdrängende Strategie handelt, kann eine Neuzuteilung einem laufenden Prozess die CPU entziehen, wenn ein anderer Prozess weniger Rechenzeit benötigt. - Im Übrigen gelten die gleichen Einschränkungen wie bei der SJF-Strategie.

SRT Strategie

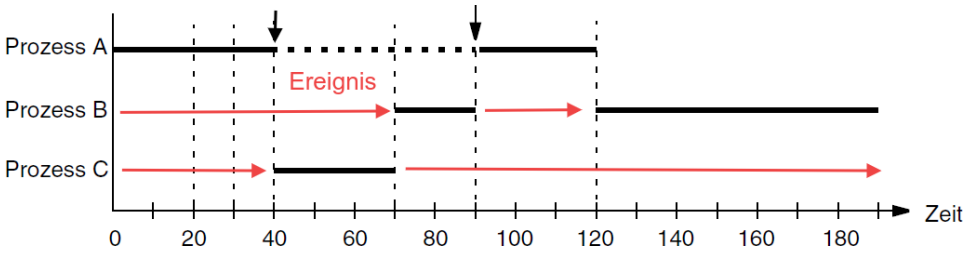


Abb. 20. Beispielszenario mit SRT-Strategie

1. Es fällt auf, dass zu den Zeitpunkten 20 und 30 jeweils eine Neuzuteilung stattfindet. Da aber Prozess A in beiden Fällen den geringsten verbleibenden Rechenzeitbedarf hat, wird er weder durch B noch durch C verdrängt.
2. Erst zum Zeitpunkt 40 findet ein Prozesswechsel statt, da A zu warten beginnt. Zu diesem Zeitpunkt erhält Prozess C den Vorrang, da er weniger Rechenzeit benötigt als B.
3. Nach dem Prozessende von C wird die CPU frei und kann daher dem einzigen dann ablaufbereiten Prozess B zugeteilt werden.
4. Zum Zeitpunkt 90 wird A wieder ablaufbereit, da die Ein-/Ausgabe abgeschlossen ist.
5. Da Prozess A einen geringeren Restzeitbedarf aufweist als B, erfolgt eine Verdrängung. Prozess B kann erst nach dem Prozessende von A seine restliche Rechenzeit beanspruchen.

RR Strategie

Bei der RR-Strategie (**Round Robin**, im Kreis herum) erhält jeder Prozess ein **Quantum an Rechenzeit** zugeteilt, für das er ununterbrochen die CPU belegen kann.

- Periodisch (z.B. alle 10 ms, "tick") wird der Zeitquantumszähler des laufenden Prozesses dekrementiert.
- Fällt damit sein Zeitquantum auf null, so wird eine Neuzuteilung veranlasst.
- Enthält die Liste der ablaufbereiten Prozesse mindestens einen Eintrag, so wird der laufende Prozess verdrängt. Er wechselt damit in den Zustand »Bereit« und wird zuhinterst in der Liste ablaufbereiter Prozesse eingetragen.
- Danach wird der vorderste Prozess der Liste entnommen, sein Zeitquantumszähler auf ein volles Zeitquantum gesetzt und ihm die CPU zugeteilt.
- Ist die Liste der ablaufbereiten Prozesse leer, wird der Zeitquantumszähler des laufenden Prozesses wieder auf ein volles Zeitquantum gesetzt und belegt die CPU weiter.
- Neu gestartete Prozesse werden anfänglich in den »Bereit«-Zustand versetzt und zuhinterst in die Liste der ablaufbereiten Prozesse eingereiht.
- Diese Strategie versucht die Rechenzeit möglichst gleichmäßig auf alle ablaufbereiten Prozesse zu verteilen, indem sich diese zyklisch abwechseln.

RR Strategie

- Da jedoch das Dekrementieren des Zeitquantumszählers nur in einem festen Zeitraster erfolgt, werden Prozesse benachteiligt, die erst kurz vor dieser periodischen Prüfung die CPU zugeteilt bekamen.
  - Ihr Zeitquantum ist damit um maximal eine Zeitscheibe verkürzt.
  - Diesem Effekt kann begegnet werden, indem ein volles Zeitquantum mehrere Zeitscheiben umfasst (z.B. 10 ticks).
- Die Implementierung des RR-Verfahrens kann unterschiedlich aufwendig sein.
- Eine einfache Implementierung wird einen festen Wert für das volle Zeitquantum nutzen. Wird ein volles Zeitquantum nur eine einzige Zeitscheibe groß gewählt, so stellt dies ebenfalls eine Vereinfachung dar.
- *Andererseits lässt sich das Verfahren verfeinern, indem prozessabhängig oder lastabhängig der Wert für ein volles Zeitquantum variiert wird.*
- *Eine weitere Variante teilt Vordergrundprozessen ein größeres Zeitquantum zu und erlaubt ihnen so, möglichst gut auf Benutzereingaben zu reagieren.*

RR Strategie

- Das RR-Verfahren geht davon aus, dass alle Prozesse zu jedem Zeitpunkt **gleich wichtig** sind, womit sich nur einfachere Systeme bauen lassen.
- Vorteilhaft ist der vergleichsweise bescheidene Verwaltungsaufwand.
- Ein weiterer Vorteil ist, dass einzelne Prozesse mit Busy Waits (E/A Polling, Schleifen) das System nicht lahmlegen können.
- Neuzuteilungen finden immer dann statt, wenn der laufende Prozess die CPU freigibt (zu warten beginnt oder terminiert) oder wenn sein Zeitquantum erschöpft ist.
- **Dieses Verfahren baut auf dem FIFO-Prinzip (kooperative Zuteilung) auf und erweitert es um die Zeitquantenumschaltung**.

RR Strategie

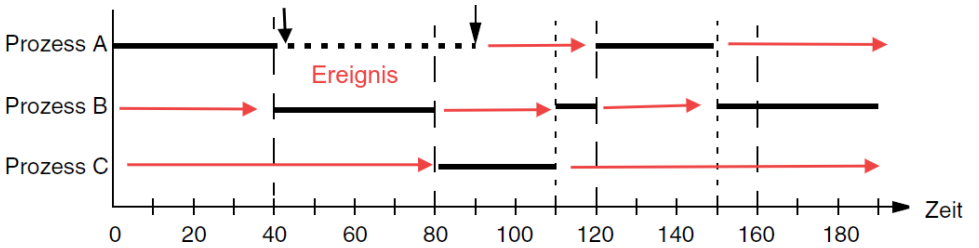


Abb. 21. Beispielszenario mit RR-Strategie: Zeitscheibengröße 40. Erreicht ein Prozess laufend das Zeitscheibenende, so wird ihm die CPU entzogen und dem nächsten bereiten Prozess zugeteilt.

1. Die Zeitscheibengröße wird exemplarisch auf 40 Zeiteinheiten festgelegt und das volle Zeitquantum ist eine Zeitscheibe. Implementierung durch periodischen Uhreninterrupt.
2. Am Anfang ist Prozess A lauffähig und erhält die CPU zugeteilt. Nach 20 bzw. 30 Zeiteinheiten werden B und C ablaufbereit. Dies ändert jedoch nichts, da Verdrängung nur bei Ablauf eines Zeitquantums möglich ist.
3. Zum Zeitpunkt 40 beginnt Prozess A zu warten, wodurch die CPU frei wird. Da Prozess B vor Prozess C ablaufbereit wurde, kann jetzt B laufen (erhält ein volles Zeitquantum von eins).
4. Zum Zeitpunkt 80 ist das Zeitquantum von B erschöpft, womit C die CPU zugeteilt bekommt.
5. Kurz darauf zum Zeitpunkt 90 wird A wieder ablaufbereit nach abgeschlossener Ein-/Ausgabe. Prozess A wird hinter B in der Bereitliste eingereiht.

RR Strategie

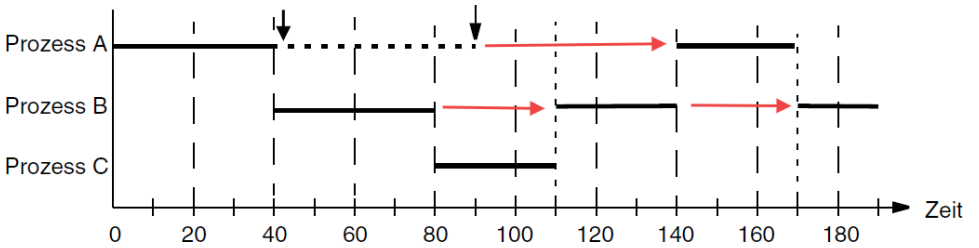


Abb. 22. Beispielszenario mit erweiterter RR-Strategie: Zeitscheibengröße 20, wobei jeder Prozess jeweils über ein Zeitquantum von 2 verfügt. Erreicht ein Prozess laufend zweimal das Zeitscheibenende, so wird ihm die CPU entzogen und dem nächsten bereiten Prozess zugeteilt.

Kleinere Zeitscheibengrößen bei höherer Anzahl von Zeitscheiben kann zu einer schlechteren E/A Reaktionszeit führen. Die Wartezeit von bereiten Prozessen kann sich verbesssern. Das hängt aber vom aktuellen Zustand der Prozesse und der Wartelisten ab!.

ML Strategie

Die ML Strategie (**Multi-Level Priority**) basiert auf Prioritäten, die den einzelnen Prozessen zugeordnet sind. Diese Prioritätszuordnung obliegt dem Anwender, der damit die Wichtigkeit der einzelnen Prozesse bestimmt.

- Die Priorität wird meistens mit Zahlenwerten umschrieben, z.B. 0..100. Ob eine hohe Zahl einer hohen Priorität entspricht oder umgekehrt, ist eine Implementationsfrage.

Typischerweise ist diese Strategie verdrängend realisiert (Präemtpion), sodass stets derjenige ablaufbereite Prozess die CPU besitzt, der die höchste Priorität hat.

- Sind mehrere Prozesse auf dieser Prioritätsstufe ablaufbereit, so gilt sekundär das FIFO-Prinzip.

Nachteilig bei dieser Strategie ist, dass die CPU von höherpriorisierten Prozessen monopolisiert werden kann. Dies ist dann der Fall, wenn diese zu oft ablaufbereit sind, sodass niederpriore Prozesse nie die CPU erhalten. Man sagt dann, dass diese niederprioren Prozesse verhungern (starvation).

ML Strategie

- Gründe für das Auftreten dieser Situation können falsch ausgelegte Prozesse, unerwartete Anwendungssituationen mit erhöhtem Rechenzeitbedarf oder eine zu leistungsschwache CPU sein.
- Die ML Strategie wird gerne für Echtzeitsysteme genutzt, da sie bei nur einem einzigen Prozess auf der höchsten Prioritätsstufe diesem stets sofort die CPU zuteilt, wenn er ablaufen will.
  - Eine Neuzuteilung des Prozessors findet nicht nur statt, wenn der aktuelle Prozess die CPU freigibt (zu warten beginnt oder terminiert), sondern auch dann, wenn ein anderer Prozess ablaufbereit wird.
  - In der Praxis wird diese Strategie oft mit der RR-Strategie kombiniert, d.h., existieren ablaufbereite Prozesse mit gleicher Priorität wie der laufende Prozess, so wechseln sie zyklisch ab (round robin).

Die Priorität kann auch dynamisch sein, d.h. sie ändert sich mit der Laufzeit von Prozessen. So werden Prozesse die häufig die Zeitscheiben ausschöüfen degradiert, welche die häufig auf E/A warten und nur kurz rehchen befördert.

ML Strategie

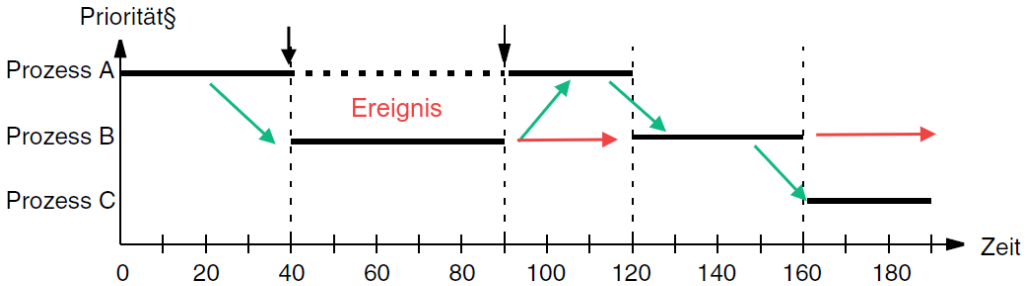


Abb. 23. Beispielszenario mit ML Strategie. Das Diagramm wurde um eine Prioritätsrangierung erweitert in der Art, dass A die höchste, B eine mittlere und C die tiefste Priorität zugeordnet bekommt.

Da Prozess A als einziger die höchste Priorität besitzt, erhält er stets sofort die CPU zugeteilt, wenn er ablaufen will. So verdrängt er beispielsweise zum Zeitpunkt 90 den Prozess B, wenn er die Ein-/Ausgabe abgeschlossen hat. Prozess B kann immer dann laufen, wenn A die CPU nicht besetzt. Prozess C schließlich muss warten, bis die CPU für ihn als Letzten frei wird.

MLF Strategie

Die MLF Strategie (**Multi-Level Feedback**) begründet ihre Scheduling-Entscheide auf der aufgelaufenen Rechenzeit der ablaufbereiten Prozesse. Dazu verwendet sie eine feste Anzahl von Prioritätsstufen, z.B. 1..10.

- Anfänglich ist einem Prozess die höchste Priorität zugeteilt.
- Die Auswahl des nächsten Prozesses bei einer Neuzuteilung berücksichtigt primär die Priorität und sekundär das FIFO-Prinzip, wenn mehrere wählbare Prozesse der gleichen Priorität vorliegen.
- Jeder Prioritätsstufe ist zudem ein festes Zeitquantum zugeordnet.
- Hat ein Prozess auf einer Prioritätsstufe das geltende Zeitquantum ausgeschöpft, so wird seine Priorität um eins erniedrigt und eine Neuzuteilung ausgelöst.

MLF Strategie

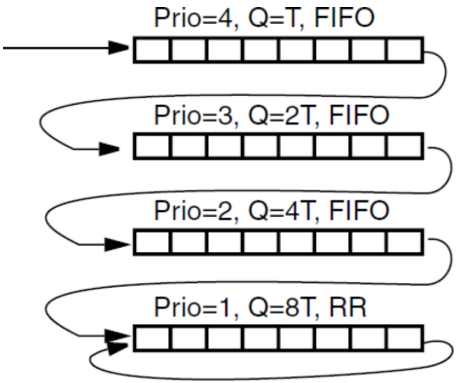


Abb. 24. Warteschlagenorganisation für die MLF-Strategie (Beispiel). Prio: Priorität, Q: Zeitquantum. T: Zeitscheibengröße

Damit werden Prozesse mit kleinem Rechenzeitbedarf bevorzugt, da lang laufende Prozesse schrittweise in ihrer Priorität heruntergestuft werden.

MLF Strategie

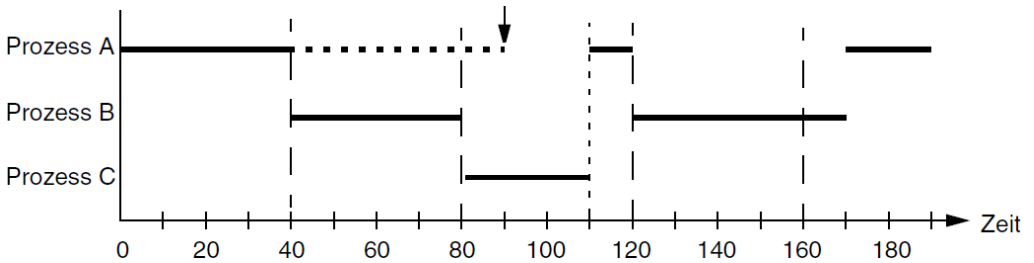


Abb. 25. Beispielszenario mit MLF-Strategie

Vorteilhafte Eigenschaften besitzt diese Strategie für interaktive Systeme, da einerseits keine Vorkenntnisse über erwartete Rechenzeiten nötig sind und andererseits E/A-lastige Prozesse bevorzugt werden. Dies sind typischerweise auch Prozesse, die einen intensiven Benutzerdialog besitzen und mit dieser Strategie kurze Antwortzeiten realisieren können. Länger laufende Hintergrundprozesse nutzen so die verbleibende Rechenzeit, ohne dass sie die Reaktionszeit der Vordergrundprozesse negativ beeinflussen.

RM Strategie

Das RM-Verfahren (**Rate Monotonic**) bietet sich für periodisch ablaufende Prozesse an, wie sie oft in Echtzeitsystemen anzutreffen sind. Dabei wird von einer festen Periodenlänge ausgegangen. Für periodische Aktivitäten gilt typischerweise als spätester Endtermin der Anfang der nächstfolgenden Periode.

- Besitzt ein Prozess die Periodenlänge T, so wird er alle T Zeiteinheiten starten und muss seine Berechnung vor dem Start der nächsten Zeiteinheit (also nach T Zeiteinheiten) abschließen.
- Es handelt sich um ein verdrängendes Zuteilungsverfahren, bei dem derjenige ablaufbereite Prozess gewählt wird, dessen Periodenlänge am kürzesten ist.

RM Strategie

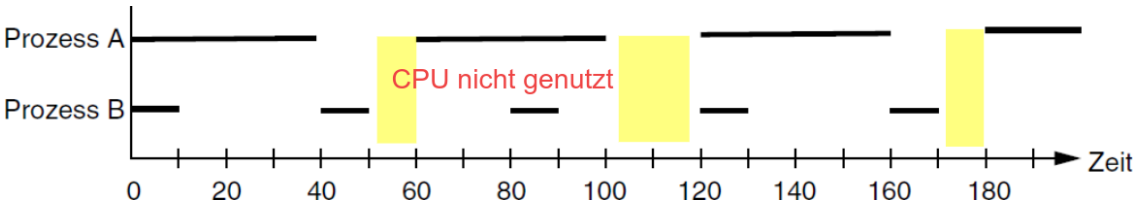


Abb. 26. Beispielszenario mit RM Strategie (periodische Prozessabläufe) ohne Verdrängung/Präemption. Gelbe Bereiche: CPU nicht genutzt.

Prozess A läuft periodisch alle 60 Zeiteinheiten und er braucht für jeden Ablauf 40 Einheiten an Rechenzeit. Prozess B hat eine Periode von 40 und einen Rechenzeitbedarf von 10 Zeiteinheiten pro Aktivierung.

EDF Strategie

Bei der EDF-Strategie (**Earliest Deadline First**) wird derjenige Prozess ausgewählt, dessen Restzeit bis zum Ablauf seines Endtermins am kleinsten ist.

- Wie bei dem RM-Verfahren gehen wir von periodisch ablaufenden Prozessen aus, die mit festen Intervallen arbeiten. Der Endtermin einer anstehenden Bearbeitung ist stets der Anfang der nächstfolgenden Periode.
- Es handelt sich um ein verdrängendes Verfahren, das sich gut für Echtzeitsysteme eignet.
- Wie bei dem RM-Verfahren gehen wir von periodisch ablaufenden Prozessen aus, die mit festen Intervallen arbeiten. Der Endtermin einer anstehenden Bearbeitung ist stets der Anfang der nächstfolgenden Periode. Es handelt sich um ein verdrängendes Verfahren, das sich gut für Echtzeitsysteme eignet.

EDF Strategie

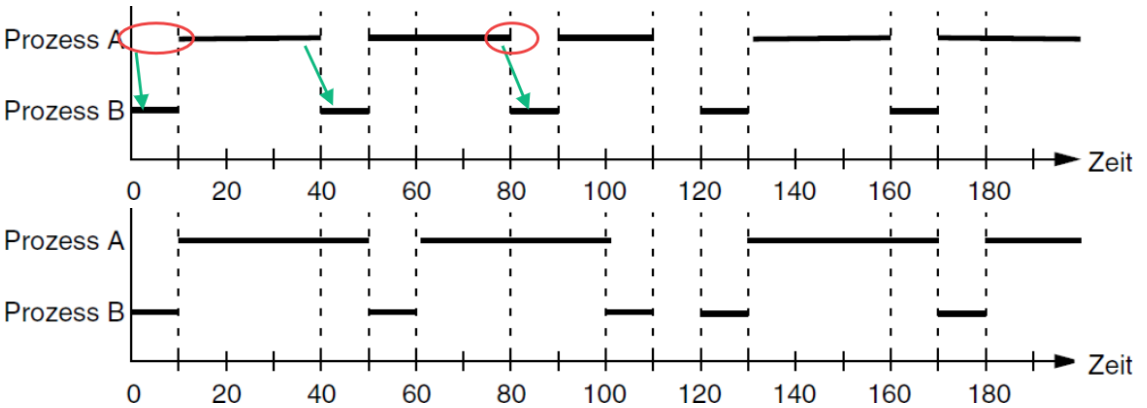


Abb. 27. (Oben) Beispielszenario für RM Strategie mit Unterbechung/Verdrängung von Prozess A durch Prozess B (Unten) EDF-Strategie mit Verdrängung

Bei EDF ist die Laufzeitunterbrechung von länger laufenden Prozesse etwas günmstiger.

- Eigentlich verpasst Prozess A hier seine Deadline da er mit B überlappt!

Vergleich der Strategien

Um verschiedene Strategien miteinander zu vergleichen, sind folgende Größen zur Beurteilung hilfreich:

- Wartezeit (wait time): Summe aller Zeiträume, in denen der Prozess am Warten war (auf irgendeine Ressource, das heißt auch auf die CPU).
- Bedienzeit (service time): Summe aller Zeiträume, in denen der Prozess am Rechnen war (d.h. CPU belegt hat, also inklusive aktiver Warteschleifen).
- Durchlaufzeit (turnaround time): Gesamter Zeitbedarf (verstrichene Zeit), bis ein Prozess vollständig abgearbeitet ist (Zeitpunkt Prozessbeendigung minus Zeitpunkt Prozessstart). Es gilt: Durchlaufzeit = Wartezeit plus Bedienzeit.
- Antwortzeit (response time): Zeit zwischen dem Auftreten eines Ereignisses bis zur ersten Reaktion des Prozesses darauf. Entweder wird die Situation bei einem Einzelereignis bestimmt oder ein Mittelwert über eine Vielzahl an Ereignissen ermittelt.
- Durchsatz (Throughput): Anzahl vollständig abgearbeiteter Prozesse pro Zeiteinheit. Zur Bestimmung wird ein Betrachtungszeitraum gewählt.
- CPU-Auslastung (cpu usage): Zeiträume, während der die CPU irgendeinen Prozess ausgeführt hat, im Verhältnis zum gesamten Beobachtungszeitraum. Typischerweise erfolgt eine prozentuale Angabe.

Vergleich der Strategien

[bsdp]

	FIFO	SJF	SRT	RR	ML	MLF	(MP)
Wartezeit (Prozess A)	120	120	50	80	50	120	50
Bedienzeit (Prozess A)	70	70	70	70	70	70	70
Durchlaufzeit (Prozess A)	190	190	120	150	120	190	120
Antwortzeit (Prozess A)	70	70	0	30	0	20	0
Durchsatz (Zeit: 0..150)	1	1	2	2	1	1	3
CPU-Auslastung (Zeit: 0..150)	100%	100%	100%	100%	100%	100%	42%

Tab. 1. Vergleich ausgewählter Scheduling-Strategien. Teils am Beispiel eines Prozesses, teils für alle Prozesse.

Zusammenfassung

- Wichtigste Konzepte für Multiprozessausführung (Multiprogramming) sind
  1. Prozessblockierung
  2. Prozessunterbrechung (Präemption)
  3. Prioritäten
- Es wird verdrängende (präemptive) und nichtverdrängende Prozessumschaltung unterschieden
- Wichtigstes Beurteilungskriterien:
  1. Antwortzeit auf Ereignisse (Verzögerung)
  2. Faire Prozessauswahl, Zeitscheiben
  3. Auslastung des Prozessors (optimal 100%)
  4. Overhead durch Prozesswechsel!
  5. Möglichkeiten Prozesse messbar zu machen

Zusammenfassung

- Man unterscheidet beim Scheduling zwei wesentliche Prozessklassen:
  1. Rechenlastig
  2. Ereignislastig (E/A)
- Die Prozessauswahl hängt ab von:
  1. Prozessklasse (s.o)
  2. Statistiken der laufende Prozesse
  3. Prioritäten
  4. Aktuell laufender Prozess!

. Das einfachste Multiprozessystem ist das Vordergund-Hintergrund System

1. Hauptprogramm ist Hintergrundprozessn (niedrigste Priorität)
2. Interrupthandler werden präemotiv aufgerufen und sind zum Vordergundprozess zusammengefasst (höchste Priorität).

Zusammenfassung

- Prozesse besitzen einen Daten- und Kontrollzuständ (Programmzähler), aber viel wichtiger werden sie für die Prozessverwaltung auf Metazustände abgebildet und in Wartelisten verarbeitet:

  1. Laufend
  2. Bereit
  3. Wartend/Blockiert
  4. ...

- Prozesswechsel bedeutet:

  1. Overhead!
  2. Umladung PCB und MMU Tabellen
  3. Verwaltungsstrukutren, Wartelisten
  4. Kontextwechsel! Das Betriebssystem ist selber ein Prozess.