Kapitel 1

Prozessverwaltung

Übersicht

Kern der Arbeit mit Computern ist die Ausführung von Programmen zur Verarbeitung von Daten. Betriebssysteme sind hierbei nur (sinnvolle) Hilfsarbeiter. Programme werden durch das Betriebssystem als Prozesse verwaltet. Das Betriebssystem teilt dabei den Prozessen die notwendigen Ressourcen zu. Gerade in einer Zeit, in der man auf einem Computer mehrere Aufgaben am besten gleichzeitig bewältigen möchte, kommt dem Betriebssystem eine besondere Aufgabe zu. Es muss verhindern, dass sich verschiedene Prozesse gegenseitig behindern und alle Prozess den notwendigen und möglichen Zugriff auf die vom Betriebssystem verwalteten Ressourcen erhalten.

Lernziele
Nach Abschluss dieses Kapitels,

1. wissen Sie, was ein Prozess ist
2. kennen Sie einfache Verfahren nach denen der Computer Ressourcen an Prozesse verteilt
3. können Sie Prozesse auf einem UNIX-System verwalten.

1.1 Grundlagen der Ressourcenverwaltung

Wie wir anfangs bereits erläutert haben, ist das Betriebssystem eine Schicht zwischen Hardware und Anwendungsprogrammen, die verschiedene Vorteile mit sich bringt:

- Nicht jedes Programm muss Funktionen zum Umgang mit der Hardware mitbringen -> Hardware wird Zentral durch OS verwaltet.
- Mehrere Programme können parallel ausgeführt werde (ggf. Zugriff auf die gleichen Ressourcen möglich)
- Zugriffe unterschiedlicher Programme und Nutzer können vom OS zueinander abgegrenzt werden und falsche oder ungewollte Zugriffe vermiden werden.

Um diese Regelung des Zugriffs auf die Hardware-Ressourcen des Computers zu verstehen, müssen wir einige grundlegende Konzepte betrachten.

Betrachten wir als erstes das Betriebssystem alleinig und verstehen wir es als eine Anwendung, so haben wir eine Anwendung vor uns, die:

- über Funktionen, um Hardware anzusprechen und zu nutzen
- exklusiv alle Ressourcen nutzen
- vollen Zugriff auf alle Ressourcen hat, insbes. CPU und Speicher
- sich einteilt, wann Ressourcen (z. B. Rechenzeit, Speicher) für welche Aufgabe herangezogen wird.

Wir hatten ebenfalls bereits festgelegt, dass das Betriebssystem für weitere Anwendungen die Ressourcenverwaltung übernimmt. Für jede weitere Anwendung, die unter Nutzung des Betriebssystems auf der gegebenen Hardware laufen soll, entstehen dadurch aber folgende Fragen:

- Wie kann eine Anwendung Zugriff erhalten, wenn eigentlich das Betriebssystem exklusiv zugreift?
- Was passiert z. B. wenn eine Anwendung in einer Schleife festhängt und sich selbst nicht mehr beenden kann?
- Wie kann das Betriebssystem selbst wieder die Kontrolle erhalten, wenn eine Anwendung die Nutzung der Hardware überlassen bekommen hat?
- Wie kann das Betriebssystem vermeiden, dass eine Software in den Speicherbereich einer anderen Software/Anwendung schreibt?
- Und wie kann damit ggf. eine Manipulation des Betriebssystems vermieden werden?

Abb.: Anschauungsmodell zum Zusammenspiel Anwendung, Betriebssystem und Hardware

1.1.1 System Calls

Da das Betriebssystem die Hardware von den Anwendungen abschirmt, muss der Zugriff auf Systemressourcen über das Betriebssystem ermöglicht werden. Das Betriebssystem stellt sogenannte SystemCalls zur Verfügung. Dies sind definierte Schnittstellen, die es erlauben eine Systemfunktion des Betriebssystems aufzurufen, dieser Parameter zu übergeben. Das Betriebssystem prüft dann die Systemaufruf, gibt ihn zur Ausführung an die entsprechende Hardware und gibt die Ergebnisse dann zurück.

Ein möglicher Systemcall wäre der Zugriff auf eine Datei und könnte wie folgt ablaufen:

- Eine Anwendung ruft den System Call zum Öffnen einer Datei auf (in C: fopen())
- Das Betriebssystem unterbricht für diesen Aufruf andere Aufgaben (Prozesse) und prüft den System Call z. B. ob die Datei existiert, Zugriffsrechte bestehen,...
- Während der System Call / das OS auf die Rückmeldung der Hardware wartet, wird zwischenzeitlich ggf. ein anderer Prozess behandelt.
- Nach Rückmeldung der angefragten Ressource werden vom OS Daten gelesen und im Speicherbereich d. Prozessors abgelegt.

SystemCalls bieten damit den Vorteil, dass die Implementierung der tatsächlichen Funktion vor dem Anwender versteckt wird. Wenn wir auf das Beispiel des Dateizugriffs zurückkommen, dann muss weder der Programmierer zum Zeitpunkt der Programmierung noch der Anwender zur Laufzeit die Information haben, um welche Art von Dateisystem es sich handelt, in dem die Datei hinterlegt. Diese Information muss zur Laufzeit dem Betriebssystem vorliegen, das zum Zeitpunkt des Systemcalls dann die korrekten internen Routinen wählt, um die Daten im vorliegenden Dateisystem korrekt zu handhaben. Ebenso kann damit gewährleistet werden, dass die Nutzung des Betriebssystems auf unterschiedlichen Hardware-Plattformen möglich wird, da der SystemCall Systemspezifisch compiliert oder ggf. implementiert wird, um den Besonderheiten der Plattform bzgl. Maschinenbefehle und Mikroarchitektur gerecht zu werden. Auch dieses Problem entbindet den Programmierer als auch den Anwender von der Aufgabe eine Entscheidung bzgl. der verwendeten Hardware zu treffen.

1.1.2 Kernel- und User-Mode

Neben den Zugriff auf die CPU muss auch der Speicher entsprechend vom Betriebssystem verwaltet werden. Insbesondere muss sichergestellt werden, dass Anwendungen nicht in den Speicher des Betriebssystems schreiben können. Dies würde es erlauben aus einer Anwendung heraus das Betriebssystem zu manipulieren. Um dies zu verhindern unterstützen viele Prozessorarchitekturen und Betriebssysteme das Konzept des Kernel- und User-Modes.

Kernel-Mode:

- Im KM ist ein Vollzugriff auf alle Hardware inkl. ges. Speicher
  und aller Befehlssätze des Prozessors möglich.
  Befehle im KM haben uneingeschränkten Zugriff.
  KM ist für den Kernel bzw. dessen Module / Treiber reserviert.

User-Mode:

- Prozesse bzw. Anweisungen im User Mode können nur auf ausgewählte Bereiche des Speichers und einen eingeschränkten Befehlssatz zugreifen. Alles darüber hinaus muss über System Calls durchgeführt werden.

Dies bedeutet, dass Anwendungen im User-Mode ausgeführt werden. Sind Zugriffe auf Hardware erforderlich, übergibt die Anwendung die Kontrolle wieder an das Betriebssystem, das dann im Kernel-Modus den Hardware-Zugriff realisiert und nach Abschluss wieder in den User-Mode schaltet und die Kontrolle an die Anwendung zurück gibt.

1.1.3 Interrupts

Die Frage, die noch offen bleibt, ist, wie und wann eine Anwendung Zugriff zu Systemressourcen erhält, wie sich das Betriebssystem wieder den Zugriff sichert und wie reagiert wird, wenn Meldungen der Hardware eintreffen und wie diese in der Bearbeitung einbezogen werden.

Hierzu dient das Konzept der Interrupts. Ein Interrupt ist eine Aufforderung (InterruptRequest IRQ), die aktuelle Aufgabe zu unterbrechen und auf ein Ereignis zu reagieren. Als Analogie hierzu kann eine Wohnungstür und eine Klingel darstellen. Die Klingel ist ein Interrupt, der uns bei einer laufenden Arbeit unterbricht und auffordert die Türe zu öffnen.

Beispiele:

Timer-Interrupt:

- Zur Vermeidung, dass Prozesse den Rechner blockieren, sendet die Hardware Uhr in regelmäßigen Abständen einen Interrupt, damit das Betriebssystem übernimmt und anstehende Aufgaben prüft

Illegal Adress Interrupt:

- Versucht ein Prozess auf einen nicht zugewiesenen Adressraum zuzugreifen wird ein Interrupt ausgelöst und der Prozess beendet.

Hardware-Interrupt:

- Wird durch Hardware (z. B. Tastatur od. Maus) ein Interrupt ausgelöst, so wird der laufende Prozess unterbrochen und die Daten von der HW entgegengenommen.

Ein alternatives Konzept ist das sogenannte Polling, bei dem von der Hardware der Status abgefragt wird. Dies erfordert aber regelmäßige Abfragen und bindet damit Systemressourcen.

Die Reaktionsfähigkeit wird eingeschränkt, je größer die Abstände zwischen den Pollings gewählt werden. In Analogie zu unserer Türklingel würde Polling bedeuten, dass wir regelmäßig zur Tür müssten, um zu prüfen, ob jemand dort auf uns wartet.

Damit Interrupts korrekt abgearbeitet werden können, sind aber weitere Schritte notwendig, damit die Bearbeitung eines Interrupts nicht zu Problemen in der zuvor laufenden Anwendung führt. Solange eine Anwendung in der CPU bearbeitet wird, sind folgende Daten in der CPU abgelegt:

- Daten im Datenregister
- Befehle im Befehlregister
- Status d. Befehlszählers
- Verweise auf Speicherbereiche
- In Bearbeitung befindliche Befehle

Bevor einer Anwendung deshalb die CPU entzogen werden kann, müssen folgende Schritte ausgeführt werden:

- Sicherung der Daten
- Sicherung der Befehlssätze
- Sicherung des Befehlszählers

Zwei Konzepte des Interrupts können unterschieden werden:

- Hardware Interrupt: HW sendet ein Signal an einen Eingang des Prozessors (i.d.R. vorgeschalteter Interrupt Controller vor Prozessor)

- Softwarwe Interrupt: Programmbefehl einen Interrupt bewirkt.

1.2 Der Prozess

Bisher haben wir im Kontext der Verarbeitung von Daten immer von Anwendungen und Prozessen gesprochen. Die Anwendung als Begriff ist weitestgehend klar. Der Zusammenhang zwischen Anwendung und Prozess scheint vorhanden, die Abgrenzung ist aber noch unklar.

Es gibt dabei für den Begriff des Prozesses unterschiedliche Definitionen:

Ein Prozess ist ein Vorgang der durch ein Programm kontrolliert wird welches zur Ausführung einen Prozessor benötigt.
Ein Prozess ist ein Programm in Bearbeitung.

Betrachten wir diese Definitionen näher, so sind sie kurz und beinhalten implizit doch eine Reihe von Punkten, die der Erläuterung bedürfen:

- Wird eine Anwendung / ein Programm gestartet, so wird sie (in der Regel) auf Ebene des Betriebssystems als Prozess behandelt.
- Der Prozess umfasst dabei alle Daten, die zur Ausführung notwendig sind: Daten im Speicher, Befehlssätze im Speicher, Daten und Befehlssätze in Registern und Werken -> Prozesskontext
- Systeminformation über Speicherbedarf, verbrauchte Rechenzeit, Rechte des Prozess
- alle Ressourcenrelevante Daten

Das Betriebssystem verwaltet alle Daten der Prozesse in einer eigenen Tabelle, der Prozesstabelle. Die darin enthaltenen Daten werden auch als

- Verkette Listen -> Process Controll Block (PCB)
- Jeder Prozess über eine ID -> PID identifiziert

1.3 Prozessverwaltung

Ein Prozessor kann immer nur einen Prozess verarbeiten. Es sind deshalb Verfahren notwendig, die Abarbeitung mehrere Prozesse zu strukturieren. Gleichzeitig gilt es sicherzustellen, dass auch das Betriebssystem Ressourcen erhält und nicht durch die Prozesse von der Nutzung von Ressourcen, wie dem Prozessor, ausgegrenzt wird.

Daher wurde die Möglichkeit geschaffen, Prozesse nur teilweise auszuführen, zu unterbrechen, und später wieder aufzusetzen und fortzuführen, um in der Zwischenzeit einen anderen Prozess bearbeiten zu können oder dem Betriebssystem ein Zeitfenster zu verschaffen. Dadurch können mehrere Prozesse für den Nutzer quasi gleichzeitig ausgeführt werden, obwohl sie in Wirklichkeit serialisiert sind. Das beispielhafte Prozessmodell beschreibt die vier wesentlichen Prozesszustände, die je nach Ausgestaltung um weitere ergänzt werden können:

Abb.: Einfacher Zustandsautomat für Prozesse

- BEREIT: Der Prozess besitzt alle Ressourcen (mit Ausnahme der CPU) und wartet auf die Zuteilung des Prozessors durch das OS.
- LAUFEND: Der Prozess ist einem Prozessor zugeteilt und läuft ab.
- WARTEND: Der Prozess wurde durch das Betriebssystem unterbrochen und wartet auf eine Ressource

(1) OS wählt den Prozess aus (aktivieren)
(2) OS wählt einen anderen Prozess aus und stoppt den laufen Prozess (deaktivieren bzw. Vorrangsunterbrechung)
(3) Prozess wird blockiert (wartet auf Input, Betriebsmittel wird angefordert)
(4) Blockadegrund fällt weg (Betriebsmittel verfügbar)
(5) Reguläre Beendigung od. Behandlung eines schwere Fehlers

Prozesse werden in ihrem Lebenszyklus damit durch einen Zustandsautomaten beschrieben, für den die oben dargestellten Zustände nur ein einfaches Beispiel darstellen und je nach Betriebssystem anders oder umfangreicher ausfallen können (siehe unten Android).

Das Betriebssystem ändert den Zustand eines jeden Prozesses zwischen BEREIT und LAUFEND hin und her, bis alle abgearbeitet sind. Einzelne Zeitabschnitte des Prozessors werden den Prozessen zugeordnet, die ablaufen wollen.

Durch das Betriebssystem werden den einzelnen Prozessen Zeitscheiben der Prozessorzeit zugewiesen. Dies erfolgt mit Hilfe eines Schedulers und eines Dispatchers.

Scheduler: Der Prozess-Scheduler ist eine Arbitrationslogik,
die die zeitliche Ausführung mehrerer Prozesse regelt
Es wird unterschieden zwischen:
- preemptiv (vorwegnehmend, unterbrechend
- nicht preemtiv (kooperativ)

- Kooperativ: OS lässt so lange arbeiten, bis der
Prozess zurrückgibt
- Preemtiv: OS lässt den Prozess nur für definierte Zeit arbeiten

Dispatcher: Dient dazu bei dienem Kontextwechsel den aktiven Prozess
die CPU zu entziehen und nach den Regeln des
Schedulers einem neuem Prozess zuzuweisen.

Der Scheduler wird immer vom Dispatcher aufgerufen und trifft
kurz- und langfristige Entscheidungen im Rahmen der Warteschlangen-
organisation

Die Anforderungen an den Scheduler variieren je nach System:

- Stapelverarbeitenden Systeme: Einfacher Scheduler - Neue Prozesse werden in der Warteschlange eingereiht und nach Abarbeitung wir ein Prozess aufgerufen (Queu Manager)
- Interaktive Systeme: Anforderung sind kurze Antwortzeiten bei oft vielen Prozessen bzw. Interrupts -> preemtives priorisieren Wartschlagen mit Zeitscheibenverfahren
- Echtzeitsystem: Für jeden Job/Abarbeitung eines Prozess gib es ein Zeitlimit -> Realtime-Scheduling

Um damit Prozesse laufend zu unterbrechen und nach gewisser Zeit wieder anzustoßen kommen zwei Prinzipien in Kombination häufig zum Einsatz:

- Priority Queue / Vorrangwarteschlange:
  Den Elementen in der Warteschlange wird ein zusätzlicher Schlüssel mitgegeben, der die Reihenfolge der Bearbeitung mit Beeinflusst.
- Round Robin / Ringpuffer:
  Gewährt allen Prozessen nacheinander für die Bearbeitung einen Zeitschlitz, danach werden die noch nicht vollständig bearbeiteten Prozesse wieder hinten in der Warteschlange eingereiht.

1.4 Threads

Moderne Konzepte erlauben es den Prozess noch granularer zu betrachten und diesen in kleinere Teileinheiten zu zerlegen und diese auch weitestgehend unabhängig voneinander abarbeiten zu lassen.

Für Threads wird ein eigenes Scheduling verwendet, diese verfügen über eigene Zustandsautomaten und Befehlszähler.

Hinsichtlich der Frage, wo die Handhabung der Threads stattfindet kann man zwei Konzepte unterscheiden:

Benutzerebene (User Thread -> Verwaltung im Programm selbst):
Eine Funktionsbibliothek übernimmt das Scheduling.
Das OS kennt nur den Prozesskontext innerhalb dessen im Usermode die Verarbeitung des Threads erfolgt. Effizient, da kein Wechsel in den Kernelmode notwendig um den Thread/Prozess zu wechseln, allerding blockieren Systemaufrufe alle Threads des Prozesses.

Kernelebene (Kernel Threas -> Verwaltung durch das OS):
- echte Parallelisierung
- keine Blockage des Prozesses durch einzelne Threads
- Wechsel zw. User-Kernel-Mode

1.5 Prozesse in Linux

Unix/Linux ist als Betriebssystem in der Lage mehrere geöffnete Programme gleichzeitig zu verwalten. Diese Programme werden intern als Prozesse bezeichnet. Viele der Betriebssystemprogramme laufen ebenfalls als eigene Prozesse. Prozesse können über das Betriebssystem auf Hardware zugreifen, Zugriffe, zeitliche Verfügbarkeit und Rechte werden dabei immer über den Kernel koordiniert und überprüft.

Unix/Linux organisiert Prozesse als Tree. Die Wurzel ist der init-Prozess. Mit dem Systemcall fork können aus init weitere Prozesse als Kindprozess abgeleitet werden.

Einsicht in die Prozesse:

```
# ps
# ps -l
# ps -ax
```

Hierarchische Struktur der Prozesse:
```
# ps axjf
```

PID -> PPID (Parent Process ID)

Bildung von Prozessgruppen zur Steuerung zusammenhängender Prozesse

Ein Beispiel hierfür:

```
1 cat <datei> | more
```

cat liest dabei die Datei aus und leitet die Ausgabe in more um. Die beiden Prozesse sind durch die Pipe verbunden und z.B. die Unterbrechung eines der Programme hat Auswirkungen auf das jeweils andere. Daher sind diese auf Ebene des Betriebssystems als Prozessgruppe zu behandeln.

Sicherheitshinweis für die Nutzung von Passwörtern:

Auf vielen Unix-Systemen wird bei ps der gesamte Befehl zum Aufruf eines Programms angezeigt. Man sollte deshalb niemals Passwörter beim Aufruf eines Programms als Parameter an das Programm übergeben. Dies kann ggf. durch andere Nutzer bei der Durchsicht der Prozesse mit ps eingesehen werden.

Übung in Linux

1. Mit ps eine Übersicht aller Prozesse erhalten. Mit -u wird zusätzlich der User angezeigt, und mit -w wird am Ende jeder Zeile ein Zeilenumbruch eingefügt.
2. Prozesse im Hintergrund: Durch Anhängen von & Prozess im Hintergrund starten
3. Mit fg und bg unter Angabe der PID kann man Prozesse zwischen Vordergrund (foreground) und Hintergrund (background) verschieben.
4. Beenden eines Prozesses mit kill (sollte nur verwendet werden, wenn ein Programm nicht mehr reagiert): kill <PID>

5. kill -9 sollte alle Prozesse killen, selbst wenn sie widerspenstig sind.

6. Den aktuellen Prozess im Vordergrund beendet man mit CTRL+C Dabei wird eine Nummer angezeigt, die alle Prozesse der aktuellen Shell durchnummeriert nicht die PID

7. Mit jobs kann man sich die aktuellen Prozesse anzeigen lassen.

8. Bei fg, bg und kill kann die jobID verwendet werden, wenn % Zeichen vorangestellt wird. fg %1 holt Hintergrundprozess in den Vordergrund. Mit bg %1 in den Background schicken.

9. Mehrere Prozesse lassen sich nacheinander starten in dem diese durch Semikolon getrennt werden

10. Zwei Prozesse können abhängig voneinander nacheinander gestartet werden mit dem Aufruf: mit programm1 && programm2 Nur wenn der erste Prozess erfolgreich war wird auch der zweite gestartet. Der erste Programmaufruf liefert bei Erfolg Fehlercode 0, damit wird zweiter Prozess angestoßen

11. Mit nice kann die Priorität eines Prozesse geändert werden. Durch Angabe einer Nummer zwischen 19 und -10 wird die Priorität definiert. Nur root darf negative Werte eingeben. Standard ist 10. Je größer der Wert, um so geringer die Prio.

12. Mit renice lassen nachträglich Prioritäten ändern

13. Leiten Sie aus der Prozessliste den Baum der Linux Prozessstruktur ab (beispielhaft an den ca. 10 -15 ersten Prozessen.