Prozesse: Prozesskontrollblock, -zustände, -umschaltung

Vorweg sollte geklärt werden, was Prozess bedeutet, wenn man im Kontext über Betriebssystemen davon redet. Ein Prozess ist ein Programm während der Ausführung und wird oft auch als 'thread' oder 'task' bezeichnet. Es ergibt sich auch oft, dass durch die Ausführung eines Programms mehrere Prozesse gleichzeitig im System erzeugt werden.

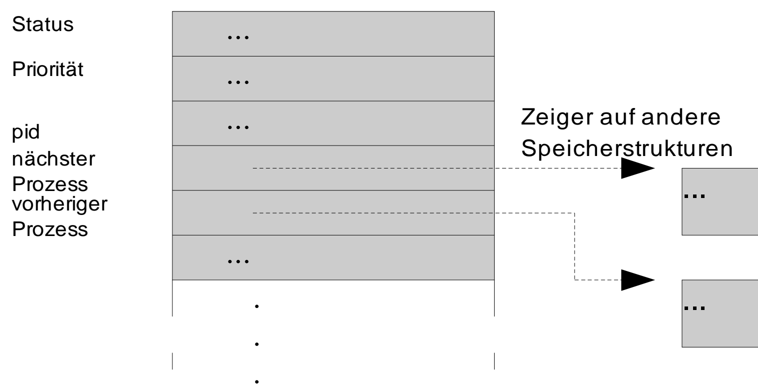
Der Prozesskontrollblock

Der Process Descriptor

Eines der wichtigsten Elemente um mit Prozessen umgehen zu können ist der sogenannte Process Descriptor.

Was ist der Process Descriptor?

Der Process Descriptor ist ist eine Speicherstruktur, dessen Felder eine sehr große Menge an Informationen über einen einzigen Prozess beinhalten. Neben reinen Einträgen in die Felder gibt es auch eine Reihe von Zeigern, die auf weitere seperate Speicherstrukturen deuten. Durch diese Verteilung und Menge an Informationen wird der Process Descriptor zu einem sehr komplexen Gebilde. Der Process Descriptor enthält z.B. Informationen über die Priorität des Prozesses, den Spciherberecih, der dem Prozess zugeordnet ist, Dateien die für den Prozess geöffnet wurden, usw.



Da der Process Descriptor nur Daten über einen einzigen Prozess beinhaltet wird für jeden Prozess im System eine eigener Process Descriptor erstellt.

Und wozu ist der Process Descriptor gut?
Anhand des Process Descriptors sind dem Kernel benötigte Daten bekannt, um zu wissen, wie er die jeweiligen Prozesse behandeln muss.

Zustände von Prozessen

Prozesse können je nach Situation und Bedingungen ihrer Umgebung verschiedene Zustände annehmen. D.h. ihnen wird eine Information zugeordnet die für den Kernel relevant ist um mit dem Prozess entsprechend umzugehen. Grob kann man sich vorstellen, dass Prozesse entweder einfach ablaufen, oder durch irgend einen bestimmten Grund einmal angehalten werden sollen.
In welchen Zustand sich ein Prozess gerade befindet wird im Process Descriptor im Statusfeld vermerkt.

Näheres anhand der folgenden Liste:

- TASK_RUNNING
  In diesem Zustand wird der Prozess ausgeführt, bzw. er wartet darauf ausgeführt zu werden

- TASK_INTERRUPTIBLE
  Der Prozess wurde angehalten (nicht gestoppt/beendet!) um auf eine Bedingung zu warten. Dabei kann es sich z.B. um eine Hardwarekomponente handeln, auf die ein Prozess zugreifen möchte, wenn das Gerät noch nicht bereit ist. Ist das Gerät bereit, wird die Ausführung des Prozesses wieder fortgeführt. Es ist auch möglich den Prozess durch ein Signal von einem anderen Prozess wieder "aufzuwecken".
  In beiden Fällen ist der Folgezustand wieder 'TASK_RUNNING'!

- TASK_UNINTERRUPTIBLE
  Dieser Zustand ist ähnlich dem Vorherigen. Der Unterschied besteht jedoch darin, dass die Ausführung des Prozesses nicht durch ein Signal fortgeführt werden kann, d.h. nur durch eine Hardwarekomponente kann die Ausführung wieder aufgenommen werden! Der Zustand ist z.B. sinnvoll, wenn ein Gerätetreiber auf eine Hardwarekomponente wartet und solange angehalten wird. Schickt nun ein anderer Prozess dem Treiber ein Signal, so darf der Treiber dadurch nicht aufgeweckt werden, sondern sollte erst auf die Hardwarekomponente warten und danach erst auf den anderen Prozess reagieren.

- TASK_STOPPED
  In diesem Zustand ist die Ausführung des Prozesses gestoppt. In diesen Zustand lässt sich ein Prozess durch bestimmte Signale versetzen. (SIGSTOP, SIGTSTP, SIGTTOU)
  Dieser Zustand ist z.B. nützlich, wenn man durch einen Debugger einen anderen Prozess überwachen lässt und den zu überwachenden Prozess stoppen muss, damit der Debugger Ausgaben machen kann.

- TASK_ZOMBIE
  Ein Prozess gelangt in diesen Zustand, wenn seine Ausführung beendet ist, aber ein anderer Prozess von dem aus er gestartet wurde noch keinen Systemaufruf (wait()-ähnlich) veranlasst hat, der Informationen über den "toten" Prozess zurückgibt. Der Kernel kann in dieser Situation belegte Resourcen noch nicht freigeben, da gespeicherte Informationen über den Prozess eventuelle vom Elterprozess noch gebraucht werden.

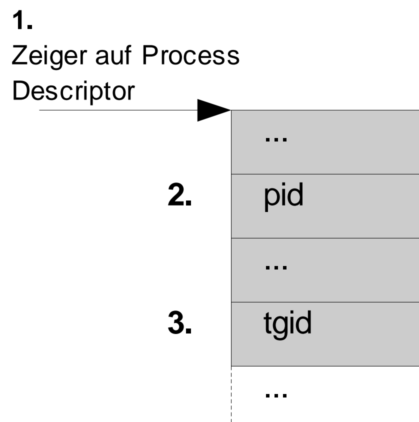
Wie werden Prozesse identifiziert?

Prozesse können auf mehreren Wegen identifiziert werden:

1. Da es zu jedem Prozess die Speicherstruktur des Process Descriptors gibt, muss dieser Speicherbereich auf irgendeine Möglichkeit im Speicher wieder auffindbar sein. Das geschieht anhand von Zeigern auf den Process Descriptor, d.h. auf jeden einzelnen Process Descriptor muss ein seperater Zeiger deuten. Es geht also, Prozesse ausfindig zu machen, über die Zeiger auf deren Process Descriptor und tatsächlich spricht der Linux Kernel in den meisten Fällen Prozesse über diese Möglichkeit an.

2. Da die 1. Möglichkeit eine eher für betriebssysteminterne Aktionen vorgesehene Variante ist, wurde für Benutzer von Unix-Systemen eine weitere Art der Unterscheidung von Prozessen eingführt, nämlich über die Zuweiseung einer Zahl zu jedem Prozess, der sogenannten PID. PID ist die Abkürzung für Process ID. Die Zahlen werden fortlaufend neuen Prozessen zugewiesen, wobei die höchste Zahl für eine PID 32767 ist. Soll der Kernel noch eine weiteren Prozess erzeugen, so muss er mit der Zählung von vorne beginnen und die nächste freie PID dem neuen Prozess zuweisen.

3. Unix-Systeme bieten außerdem die Möglichkeit, Prozesse die zusammen zu einer Gruppe gehören über eine eine gemeinsame PID zu adressieren. In Linux wird das realisiert indem alle Process Descriptors der Prozesse einer Gruppe in einer (doppelt verketteten) Liste vermerkt werden. Die PID des ersten Prozesses in der Gruppe wird zur Identifikation verwendet und in einem Feld der Process Descriptors, dem tgid-Feld (steht für thread group id), der restlichen Prozesse eingetragen. Wobei aber dennoch in jedem pid-Feld jedes Process Descriptors die normale Durchnummerierung für PIDs beibehalten wird!

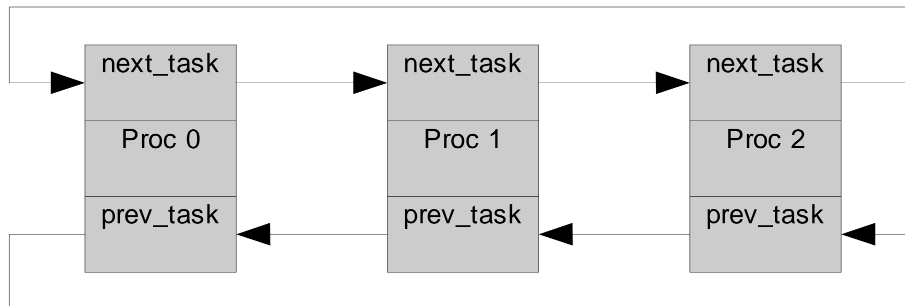


Woher weiß der Kernel wieviele Prozesse es gibt?

Dafür gibt es (doppelt verkette) Listen, für Prozesse in verschiedenen Zuständen. D.h. Prozesse deren Zustand TASK_RUNNING ist, werden in einer anderen Liste behandelt, wie Prozesse mit Status TASK_INTERRUPTIBLE.
Der Aufbau solcher Listen erfolgt mittels Zeigern auf Process Descriptors. Die einzelnen Listenelemente werden unterneinader mit Zeigern den Process Descriptor des vorher bzw. nachher gestarteten Prozesses verknüpft. Gespeichert werden die Zeiger im Process Descriptor in den Feldern prev_task und next_task.

Prozesslisten als Warteschlangen

Warteschlangen werden häufig gebraucht, wenn Prozesse z.B. auf ein Hardware Gerät warten müssen, bis es für den Prozess zur Verfügung steht oder ein Prozess muss auf eine Datenübertragung der Festplatten warten. D.h. eine Warteschlange ist eine Sammlung angehaltener Prozesse. Die Verkettung der einzelnen Prozesse geschieht wieder über Zeiger auf Process Descriptors, also wie Prozesslisten.
Muss ein Prozess nun auf eine bestimmte Bedingung warten, so kann er sich durch einen Systemaufruf durch den Kernel in eine Warteschlange eingliedern lassen.



Der Übersicht wegen deuten die Zeiger "irgendwo" auf den Process Descriptor. Normalerweise deuten die Zeiger auf die Speicheradresse, ab der der Process Descriptor beginnt!

Es kommt manchmal vor, dass zwei Prozesse auf die gleiche Resource warten, die aber nur den Zugriff eines einzigen Prozesses erlaubt. In diesem Fall macht es keinen Sinn beide – oder gar mehr – Prozesse aufzuwecken. Daher wurde eine Unterscheidung, bei angehaltenenen Prozessen, zwischen exklusiven und nichexklusiven Prozessen eingeführt. Wartet ein exklusiver Prozess auf eine Bedingung, so wird nur dieser eine Prozess vom Kernel aufgeweckt, sobald die Bedingung wahr wird. Im Gegensatz dazu, werden nichtexklusive Prozesse alle zusammen aufgeweckt, wenn die entsprechende Bedingung wahr wird, auf die die Prozesse zusammen warten. Wartende, nichtexklusive Prozesse treten z.B. bei Festplattenaktionen auf.

Die Prozessumschaltung

Warum die Prozessumschaltung?

Auf der CPU es Rechners kann immer nur ein Prozess bearbeitet werden. Da aber mehrere Prozesse im System parallel existieren können und ablaufen, muss der Kernel dafür sorgen, dass immer nur ein Prozess Zugriff zur CPU erhält und alle anderen Prozesse währenddessen angehalten werden. Wobei angehalten hier nicht bedeutet, dass die Prozesse vom Kernel in einen anderen Zustand versetzt werden, sondern lediglich darauf warten, sobald wie möglich wieder auf der CPU ablaufen zu können, d.h. die angehaltnen Prozesse befinden sich nach wie vor im Zustand TASK_RUNNING.

Soll eine Prozessumschaltung stattfinden, ist zu bedenken, dass mehrere Prozesse zwar ihren eigenen Speicherbereich zugeteilt bekommen, sich in der CPU aber die Register teilen müssen. Da zu jedem Prozess das Muster der Register anders aussieht muss der Kernel dafür Sorge tragen, dass die Zustände der Register vor dem Anhalten eines Prozesses gespeichert werden. Soll ein angehaltener Prozess seine Arbeit wieder aufnehmen, so müssen die Zustände der Register vorher natürlich wieder korrekt in die CPU

geladen werden. Das Speichern und Laden der CPU Register wird als 'hardware context' bezeichnet. Gespeichert wird der Inhalt der CPU Register an zwei verschiedenen Orten – Ein Teil wird im Process Descriptor abgelegt, der Rest befindet sich nach dem Speichervorgang im Kernel Modus Stapelspeicher.

Obwohl bei der Intelarchitektur in der Hardware ein Mechanismus implementiert ist, um die Umschaltung zwischen Prozessen zu einem großen Teil der Einfachkeit halber der Hardware zu überlassen, geht Linux hier seinen eigenen Weg und übernimmt die Umschaltung nur auf Software-Ebene. Da das nicht merklich langsamer ist, bietet diese Variante dabei zusätzlich noch die folgenden entscheidenden Vorteile:

1. Der Vorgang wird transparenter und es lässt sich deshalb besser kontrollieren, dass die Daten fehlerfrei in die Register geladen werden.

2. Die Implementierrung auf Hardwareseite ist fest durch Gatter verankert und kann nicht modifiziert werden. Auf Softwareebene besteht natürlich die Möglichkeit eventuelle Optimierungen an der Umschaltprozedur vorzunehmen.

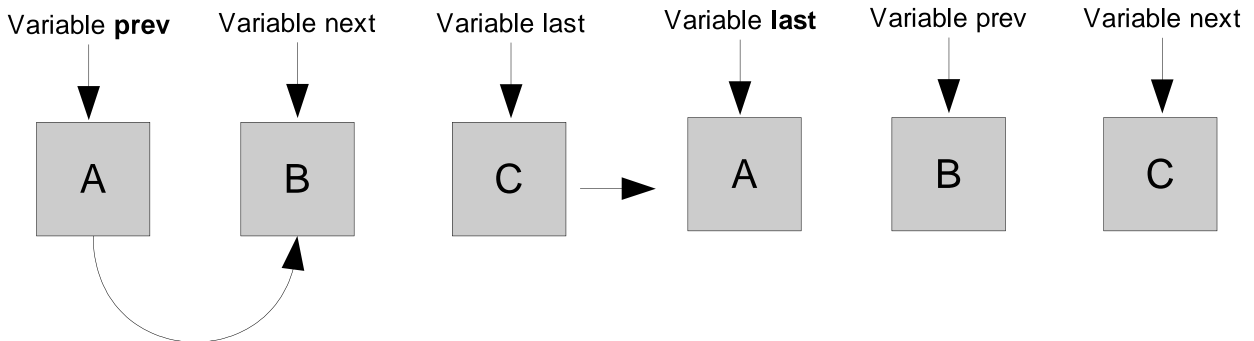
Wie läuft die Prozessumschaltung ab?

Die Umschaltung zwischen Prozessen wird ausschließlich vom Scheduler aufgerufen. Dazu müssen zwei Variablen des Schedulers bekannt sein, nämlich prev und next, wobei prev einen Zeiger auf den Process Descriptor des Prozesses darstellt, der durch einen anderen ersetzte werden soll. Next stellt einen Zeiger auf den Process Descriptor des Prozesses dar, der als nächster ausgeführt wird.

Um die Umschaltprozedur anzustoßen wird ein Makro aufgerufen, das drei Parameter benötigt. Zwei davon sind Zeiger auf die Process Descriptors des zu ersetzenden und des wieder auszuführenden Prozesses. Der dritte Parameter ist ebenfalls ein Zeiger auf den Process Descriptors eines dirtten Prozesses.

Warum ein dritter Prozess?
Wenn der Kernel einen Prozess A anhalten und einen Prozess B ausführen will, so zeigt in der schedule()-Funktion der Paramter prev auf den Descriptor von A und der Paramter next auf den Descriptor von B. Sobald das switch_to() Makro abläuft, wird A angehalten.
Will der Kernel zu einem späteren Zeitpunkt A erneut ausführen, so muss er einen Prozess C durch die Ausführung des switch_to() Makros anhalten, wobei der Platzhalter prev auf C und der Platzhalter next auf A zeigt. A findet durch den Zeiger auf dessen Process Descriptors seine relevanten Daten wieder. Die Variable next wird mit dem Zeiger überschrieben, der auf den Descriptor des als nächstes auszuführenden Prozesses (in diesem Fall B) zeigt und die Variable prev würde mit dem Zeiger auf den Descriptor von Prozess A überschrieben. Die Folge wäre, dass der Zeiger auf den Descriptor von C und damit keine Informationen mehr über diesen Prozess verfügbar sind. Um dieses Problem aber zu umgehen wird der dritte Parameter 'last' benötigt, der den Zeiger auf den Descriptor von C speichert damit dieser Zeiger wieder in die Variable prev geschrieben werden kann.

Von Prozess A soll zu B umgeschalten werden.



switch_to(prev, next, last)
A B C

Hier sind prev, next und last Parameter!!