Konzepte von Betriebssystemkomponenten (KVBK)
Schwerpunkt Linux

Adressräume, Page Faults, Demand Paging, Copy on Write

Seminar am 24.11.2003, Referent: Johannes Werner

Speicherverwaltung ist bei heutigen Rechensystemen ein zentrales Thema, da die Leistung hauptsächlich durch ineffiziente Seitenverwaltungstechniken gebremst wird. Im Folgenden sollen einige auf modernen Rechnern gebräuchliche Methoden vorgestellt werden.

1.) **Adressräume**

1.1) Der ideale Adressraum

Vom Programmierer aus betrachtet, wäre es sicher ideal, wenn sein Programm die alleinige Kontrolle über den gesamten Adressraum besäße, also bei i386 kompatiblen CPUs 4GB, diese vollständig (physikalisch) vorhanden wären und er sie mit niemandem teilen müsste. Auch nicht mit diversen Peripheriegeräten. Eine typische Nutzung des Speichers in idealisierter Form wird in Figur 2.1 dargestellt. Am oberen Ende ist der Stack angesiedelt (Stapel, Keller), auf dem zum Beispiel Funktionsparameter (std-call), Rücksprungadressen und Sicherungsdaten (z.B. in Subroutinen benutzte Register) abgelegt werden. Am Speicheranfang liegen zuerst der Programmcode, dann einige statisch allokierte Daten und anschließend die dynamischen Daten (Heap). Dazwischen ist normalerweise ein sehr großer Bereich an nicht benutztem Speicher.

Fig 2.1

1.2) Der reale (physikalische) Adressraum

In der Realität sieht das System allerdings anders aus: nur ein geringer Teil des adressierbaren Speicher ist auch tatsächlich bestückt, der dann auch noch unter mehreren Prozessen aufgeteilt werden muss. In diesem befinden sich zudem noch E/A-Bereiche, Interruptvektoren und das BIOS, die zusätzlich berücksichtigt werden müssen.

1.3) Das Abbildungsproblem

Da die Programmierer möglichst auf jedem Rechensystem die gleichen Voraussetzungen vorfinden sollen, ist es die Aufgabe des Betriebssystems, jedem Prozeß einen möglichst idealen Adressraum bereitzustellen. Dazu muss es die von den einzelnen Prozessen benutzten (idealen) Adressräume auf den vorhandenen Speicher abbilden. Wenn dabei jedem Prozess von Vornherein 4 GB Speicher zur Verfügung gestellt würden, könnte man den durch Auslagerung der physikalisch nicht vorhandenen Seiten aufkommenden E/A Transfer nicht mehr bewältigen (in vernünftigem Maß). Dieses Problem kann man durch die Tatsache vermeiden, dass die meisten Prozesse im Allgemeinen nur wenige MB an Speicher verbrauchen. Diese liegen an den beiden Enden des (idealen) Adressraums. Der Speicher dazwischen muss (kann) dem Prozess erst dann zur Verfügung gestellt werden, wenn dieser ihn auch benötigt. Ein Adressraum ist in diesem Fall die Menge aller

allokierten Speicherbereiche, also aller Adressen, auf die der Prozess zugreifen darf ohne sie erneut anzufordern. Man nennt ihn auch tatsächlichen Adressraum.

1.4) Ziele bei Mehrprozess- Betriebssystemen

Bei zur Mehrfädigkeit befähigten Betriebssystemen ist es außerdem erforderlich, dass mehrere Prozesse jeweils einen eigenen vollständig unabhängigen Adressraum besitzen. Dieser muss zudem durch jegliche Fremdeinwirkung durch andere Prozesse geschützt werden (Ausnahme: geteilter Speicher, der zur Interprozess – Kommunikation verwendet werden kann). Sollte der physikalische Speicher trotz der Maßnahmen von 2.3 nicht ausreichen, kann der Speicher von „schlafenden“ Prozessen ausgelagert werden.

1.5) Umsetzung in Linux

Um die oben genannten Details umzusetzen, besitzt unter Linux jeder Prozess einen Prozess Memory Descriptor (PMD) vom Typ `mm_struct`. Dieser enthält z.B. die Anzahl der zugehörigen Speicherseiten (4KB) und einen Zeiger auf die erste Speicherregion (SR). Diese werden von einem Memory Region Descriptor (MRD bzw. VMA) des Typs `vm_area_struct` verwaltet und beinhalten, neben anderen Informationen, die Zugriffsrechte des Prozesses auf die Region, jeweils einen Zeiger auf die erste Adresse der SR und einen auf die erste Adresse nach der SR, sowie einen Zeiger auf die nächste SR. Diese Listenverwaltung wird allerdings bei großen Prozessen recht schnell sehr ineffizient. Daher werden die PMDs ab einer bestimmten Anzahl (in der Regel 32) als Red-Black Baum gespeichert (ab Kernel 2.4, davor: AVL Baum). Um gerade bei großen Prozessen den Verwaltungsaufwand zu reduzieren, wird zusätzlich versucht, nebeneinander liegende SR zu verschmelzen. Dies ist nur dann nicht möglich, wenn benachbarte Regionen nicht dieselben Zugriffsrechte haben (vgl. Figur 2.2).

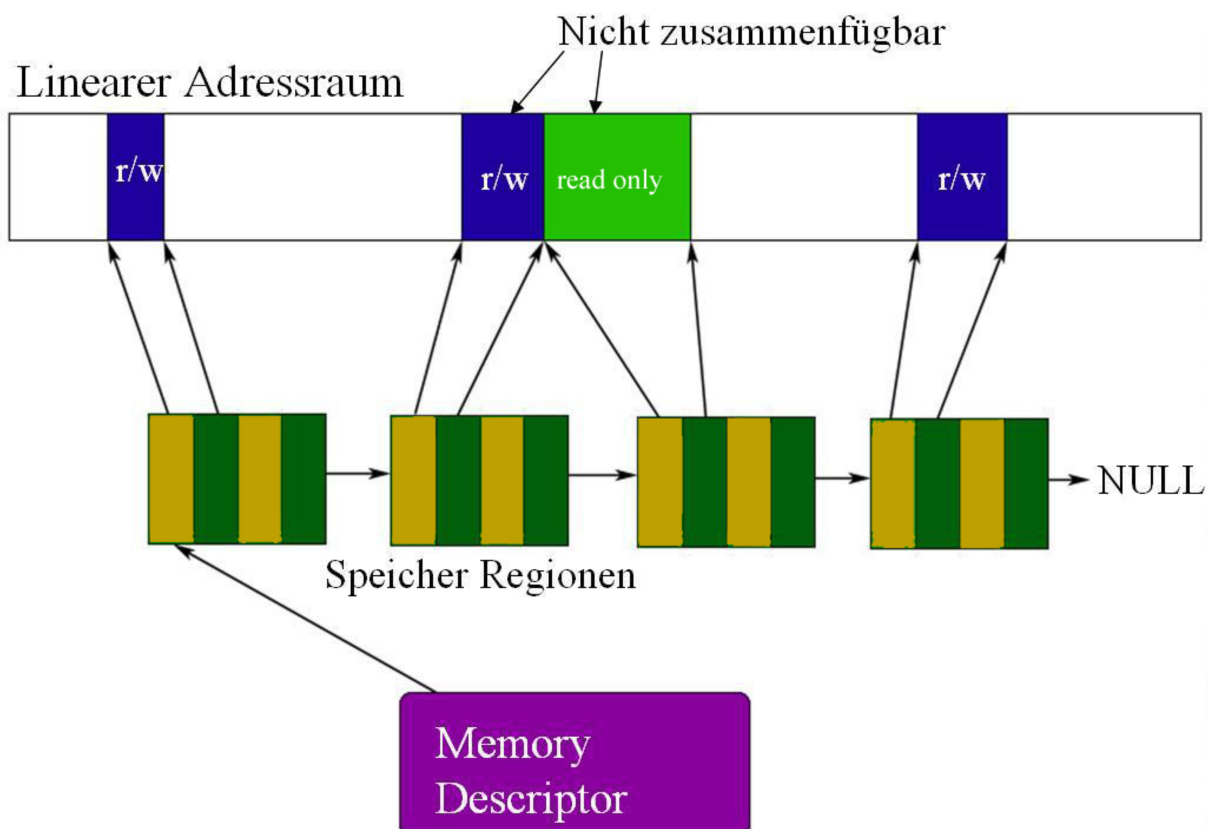

Fig 2.2

2.) **Seitenfehler (Pagefault)**

2.1) Auftreten von Seitenfehlern

Sobald ein Prozess versucht, eine Seite aufzurufen, die nicht im Speicher vorhanden ist, wird von der Speicher- Verwaltungseinheit (MMU) ein, u.a. den auslösenden Befehl beschreibender, Fehlercode auf den Stack geschoben und die Adresse, die den Fehler verursacht hat, ins Register `cr2` geschrieben. Anschließend löst sie einen Interrupt, den sogenannten Pagefault, aus.

2.2) Behandlung von Seitenfehlern

Tritt ein Seitenfehler auf, muss zuerst überprüft werden, ob die angewählte Seite vom Prozess allokiert worden ist. Ist dies nicht der Fall, muss dem Prozess entweder weiter Speicher zugeteilt werden (z.B. wenn der Stapel über eine Seitengrenze wächst) oder der Fehler anderweitig, etwa durch Terminieren des Prozesses, behandelt werden. Ist hingegen die Seite allokiert, aber nicht im RAM verfügbar, weil sie z.B. ausgelagert wurde, muss sie in eine (gegebenenfalls erst zu erzeugende) freie Speicherseite kopiert oder zumindest zum Kopieren eingeplant werden. Anschließend muss der letzte Befehl des Prozesses wiederholt werden, damit der Prozess ordnungsgemäß weiterlaufen kann.

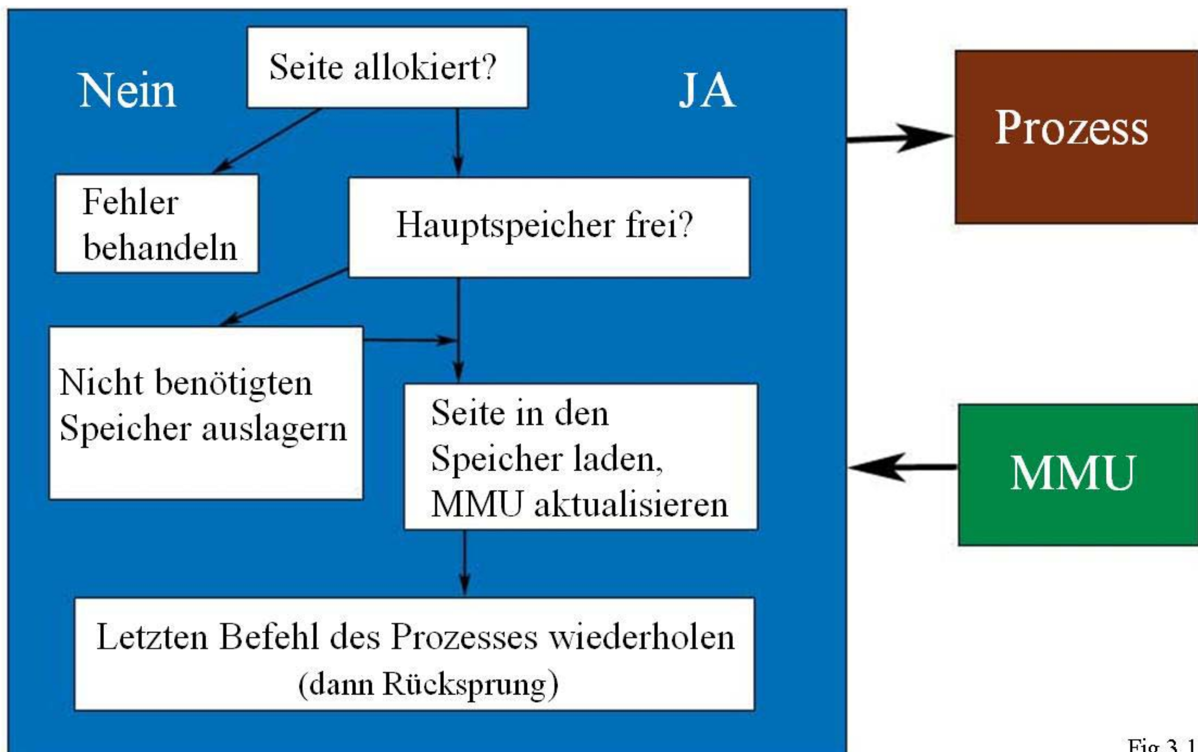

Fig 3.1

2.3) Der Pagefault-Handler in Linux

Der Handler holt sich zunächst die von der MMU bemängelte Adresse aus dem Register `cr2`. Anschließend versucht er, die Adresse einer virtuellen Speicherregion zuzuordnen. Nun sind mehrere Fälle zu unterscheiden:

1. Die Adresse liegt in keiner zur Zeit benutzten virtuellen Speicherregion des Prozesses. In diesem Fall wird geprüft, ob der Prozess auf Nutzer- oder Kernelebene auf die Adresse zugreifen wollte. War es Ersteres, so wird dem Verursacher das Signal `SIGSEGV` (Segmentation Violation) gesendet. War es der Kernel, so wird eine Fehlermeldung ausgegeben und das System über die Funktion `die()` in den Stillstand versetzt.

2. Die Startadresse der virtuellen Speicherregion, in dem sich die angeforderte Adresse befindet, ist größer als die Adresse selbst. Dies deutet auf eine Stack-Adresse hin, da der Stack nach „unten“ wächst (vgl. Fig 2.1). Ist für den Bereich das Flag `VM_GROWSDOWN` gesetzt, was für den Stack-Bereich der Fall ist, so wird eine neue Stack-Seite angelegt, falls noch genügend Platz dafür ist. Anderenfalls erhält der verursachende Prozess ebenfalls ein `SIGSEGV`. Sollte das Flag `VM_GROWSDOWN` nicht gesetzt sein, so handelt es sich auch nicht um einen Stack. Der Adresswiderspruch wird in diesem Fall nicht weiter behandelt, sondern ein `SIGSEGV` an den Verursacherprozess gesendet.
3. Ist keine der oben genannten Situationen zutreffend, wird geprüft, ob die Operation die Zugriffsrechte (Lesen, Schreiben, Ausführen) der Seite verletzt. Ist dies der Fall, wird der Prozess – ebenfalls mit dem Signal `SIGSEGV` – terminiert. Ist der Zugriff vom Typ „Schreiben“ und darf die Seite nur gelesen werden, wird zusätzlich abgeprüft, ob es sich um eine zurückgestellte Seite handelt, und dies behandelt (vgl. Copy on Write). Stellt sich der Zugriff als legal heraus, wird geprüft, ob die Seite ausgelagert oder noch nicht erstellt wurde (vgl. Demand Paging). In diesen Fällen wird die Seite zum in den Speicher holen eingeplant.

3.) **Demand Paging**

3.1) Das Konzept

Viele Prozesse besitzen mehr Speicherseiten, als sie im momentanen Zustand benötigen, da normalerweise nicht ständig die benutzte Seite gewechselt wird (werden sollte :-). Daher ist es unnötig, beim Taskwechsel alle Speicherseiten eines Prozesses in den Speicher zu laden. Eine bessere Strategie ist es, die Seiten erst dann zu holen bzw. bei nie zuvor benutzten Seiten Speicher (wirklich) zu allokieren, wenn diese auch wirklich benutzt werden (Englisch: „on demand“). Diese Verfahrensweise ist insofern effizienter, da insbesondere bei großen Prozessen eine Menge ausgelagerter Seiten erst einmal von der Festplatte geladen werden müssten, was sehr lange dauert. Dadurch verkürzt sich auch die zum Prozesswechsel benötigte Zeit.

3.2) Anwendung unter Linux

Linux unterscheidet Demand Paging in 2 Fälle, eine ausgelagerte Seite in den Speicher wieder einlesen und eine nie benutzte Seite neu anlegen:

1. Demand Allocation
   Wenn eine noch nicht existierende Seite zum allerersten Mal gelesen wird, wird diese, da ja noch keine Daten vorhanden sind, einfach mit der schreibgeschützten Seite `empty_zero_page` verknüpft. Sollte versucht werden, die nun ja vorhandene Seite zu beschreiben, wird ein Fehler ausgelöst und der zweite Fall ausgelöst:
   Wenn eine Seite zum ersten Mal beschrieben wird, wird die Funktion `alloc_page()` aufgerufen, um eine neue Seite zu allokieren. Anschließend wird die MMU aktualisiert und der schreibende Befehl wiederholt.
2. Behandlung ausgelagerter Seiten
   Ausgelagerte Seiten werden von der Funktion `do_swap_page()` in den Speicher zurückgeladen.

4.) **Copy on Write**

4.1) Copy on Write im Allgemeinen

In älteren Systemen wurde, wann immer `fork()` aufgerufen wurde, der Adressraum des Eltern-Prozesses vollständig kopiert. Da dies recht zeitaufwendig und möglicherweise nutzlos (da der Kind-Prozess die Daten evtl. gar nicht benötigt oder beide Prozesse sie nicht mehr ändern) ist, ist es effizienter, die Daten erst dann zu kopieren, wenn sie tatsächlich verändert werden sollen. Ansonsten können sich beide Prozesse die Daten teilen.

4.1) Verwendung unter Linux

Während des `fork()` Systemaufrufs werden die Seiten des Eltern-Prozesses als nur lesbar markiert und beiden Prozessen zugeordnet. Wenn nun ein Schreibzugriff auf eine der (geteilten) Seiten geschieht, tritt ein Seitenfehler auf. Da im zugehörigen MRD allerdings der Schreibzugriff erlaubt wird, ruft der Pagefault Handler die Funktion `do_wp_page()` auf, um die betroffene Seite zu kopieren. Auf diese Weise müssen nur die Seitentabellen-Einträge bei einer `fork()`-Anforderung kopiert werden.

5.) **Literaturhinweise**

Bovet D.P.: Understanding the Linux Kernel, Second Edition, O'Reilly, 2002

Mel Gorman: Understanding The Linux Virtual Memory Manager
http://www.skynet.ie/~mel/projects/vm/guide/html/understand/

Podschun T.E.: Das Assembler- Buch, Addison-Wesley, 2002