Kapitel 12: Adressraumverwaltung

- Motivation
- Basisbegriffe
- Abbildung Logischer Adressraum -> Hauptspeicher
- Adressraumverwaltungen
    - Einprogrammbetrieb
    - Mehrprogrammbetrieb
    - Adressräume größer als der Hauptspeicher

12.1 Motivation und Einführung

Damit eine Anwendung korrekte Resultate liefern kann, sollte sie in einer sicheren Umgebung ablaufen, unbeeinflussbar durch gleichzeitig ablaufende andere Aktivitäten. Der Adressraum liefert uns diesen Schutz (siehe Vergleich zur Wohnung), ein Prozess oder eine Task bewahrt sich somit mit Hilfe ihres Adressraums eine Art "Privatsphäre". Andererseits kann es aus Effizienzgründen lohnend sein, gewisse Dinge gemeinsam zu benutzen (vergleiche Sauna in einem großen Wohnhaus). Typische Beispiele von gemeinsam verwendbaren Programmen stellen die Programmbibliotheken dar.

12.2 Basisbegriffe

Sieht sich ein(e) Anwenderprogrammierer(in) seine(Ihre) Applikation im Hinblick auf Erfordernisse im Zusammenhang mit der Speicherverwaltung an, dann wird er(sie) leicht Softwareabschnitte feststellen können, die aus Sicht des Prozessors unterschiedlich behandelt werden. Codeabschnitte sollen selbstverständlich ausführbar sein, einige Datenabschnitte sollen vielleicht nur lesbar, andere sowohl les- als auch beschreibbar sein. Die jeweiligen Größen dieser verschiedenartigen Softwareabschnitte können von Applikation zu Applikation höchst unterschiedlich sein. Es wird sich zeigen, dass wir die so unterscheidbaren Softwareabschnitte unterschiedlich auf den Hauptspeicher abbilden können.

Sinn und Zweck von Adressräumen ist der Schutzaspekt. Anwenderprogrammierer wollen vor ihresgleichen geschützt sein, wobei hierbei weniger böse Absichten unterstellt werden sollen, sondern die vielen Möglichkeiten, die zu Adressierungsfehler führen können, vergessene Zeigerinitialisierung dienen häufig als Paradebeispiel von verbreiteten Programmierfehler von C/C++-Programmierern.

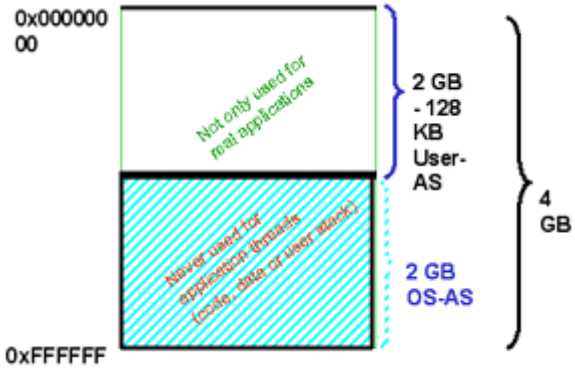
Ein Adressraum schützt eine Applikation vor Einbruch etc., wie auch eine Wohnung einen gewissen Schutz des Wohnungsinhabers vor Diebstahl bietet.

12.2.1 Adressrahmen

Ein logischer Adressraum umfasst also alle zugelassenen logischen Adressen des logischen *Adressrahmens* (address scope). Der Unterschied wird an folgendem Schaubild klar: Ein 32 Bit Rechner hat einen logischen Adressenumfang von Adresse 0 bis 2^{32} = 4 GByte. Der Adressraum einer typischen Anwendung wird aber nicht jede dieser 4 GB Adressen umfassen, denn zum einen benötigt ja auch der Kernadressraum Platz, der von WindowsXP beansprucht beispielsweise die letzten 2 GB (+ ein paar Bytes aus dem ersten beiden GB), der von Linux gibt sich mit nur einem GByte zufrieden.

Zunächst ist es interessant zu eruieren, wie groß der potentielle Rahmen für die Programmierung von Anwender- und Systemsoftware ist. Der so genannte Adressrahmen (*address scope*) einer 32-Bit Maschine wird durch eben diese 32-Bitadressen limitiert, man spricht in diesem Zusammenhang auch von der Adressbreite des Prozessors. In den heutigen 64-Bit Maschinen wird i.d.R. noch nicht mit den vollen 64-Bit Adressen, sondern i.d.R. nur mit 48-Bit-Adressen operiert. Wie dieser potentielle Adressrahmen unterteilt wird, um beispielsweise System und Anwender voneinander zu trennen, bleibt dem Systemarchitekten

überlassen. Die folgende Abbildung zeigt die mögliche Aufteilung des 4 GB großen Adressrahmen bei einer Intel x86 32-Bit Maschine, wie sie beim Windows NT System verwendet wird.



12.2.2 Adressraum

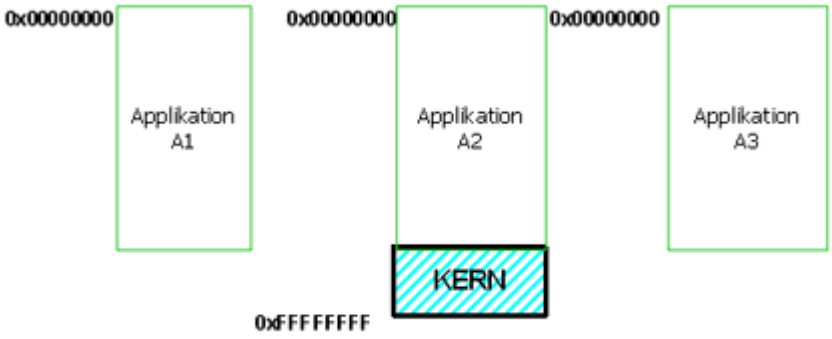
Wie man aus obiger Skizze entnehmen kann, gibt es zumindest zwei logisch getrennte Adressrahmenabschnitte, den so genannten Kernadressraum und den Anwenderadressraum

Definition:

> Ein logischer Adressraum ist der Adressbereich im Adressrahmen, der für die jeweilige Softwareeinheit (Kern oder Anwender) benutzbar ist.

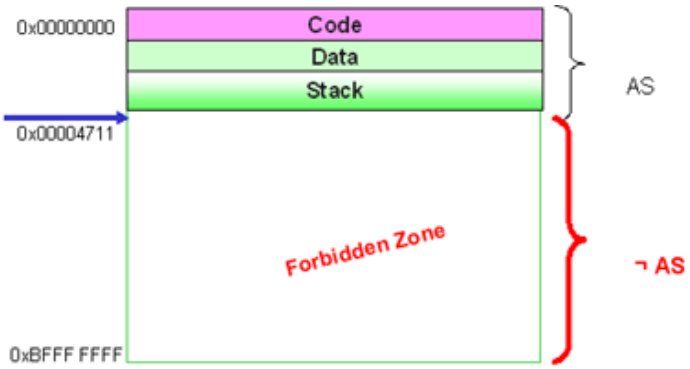
Wie erreicht man jedoch, dass sowohl in Linux als auch in Windows gleichzeitig mehrere Tasks gleichzeitig aktiviert sein können, und trotzdem keine unbeabsichtigte bzw. bösartige Beeinflussung der Anwendertasks untereinander stattfinden kann?

Man definiert für jede Anwendertask bzw. jeden Anwenderprozess einen eigenen *logischen Adressraum* (In Windows von jeweils 2 GB, in Linux von jeweils 3 GB Größe). Beispielsweise würde in einem Linuxsystem mit drei gleichzeitig aktivierten Anwendertasks die Aufteilung wie folgt aussehen:



Man muss nun allerdings noch dafür sorgen, dass die Applikationsadressräume so auf den Hauptspeicher abgebildet werden, dass sie sich nicht stören. Diese Abbildungsvorschriften müssen per Hardware unterstützt werden, da ansonsten zuviel Verwaltungsaufwand (*overhead*) damit verknüpft wäre. Es gilt nun auszuloten, wie Adressräume prinzipiell implementiert werden können.

Ein zusammenhängender logischer Adressraum enthält keine Adressraumlöcher, wird aber i.d.R. trotz allem nicht den gesamten zur Verfügung stehenden Adressrahmen ausschöpfen (siehe folgende Skizze).



Diskutieren Sie Vor- und Nachteile des zusammenhängenden logischen Adressraums.

Man kann sich jedoch auch Platzierungen von Programmabschnitten innerhalb des 3 GB großen Adressrahmens von Linuxanwendungen vorstellen, die sich an anderen Gegebenheiten orientieren.

12.2.3 Regionen

Im Allgemeinen müssen wir somit davon ausgehen, dass es innerhalb eines Adressraumes unterschiedliche Zonen gibt, solche, die definiert sind und somit benutzt werden können, und so genannte *Adressraumlöcher* (*address space holes*), die somit nicht benutzt werden dürfen. Auch in den definierten Zonen mag es noch unterschiedlich benutzbare Adressregionen geben, z.B. bildet Linux seine Bibliotheken und den Anwendercode auf die niedersten Adressen ab, während initialisierte Daten und der Heap abgesetzt und auf Megabytegrenzen ausgerichtet abgebildet werden, und schließlich der Stapel (*stack*) auf die obersten gültigen Anwenderadressen abgebildet wird (s.u.). Zusammenhängende Adressbereiche, die gleichartig benutzt werden können, nennt man *Regionen* (*regions*), oder mit anderen Worten, innerhalb einer Region gibt es keine Adressraumlöcher.

Ferner muss natürlich angemerkt werden, dass nur wenige Anwendungen wirklich 3 GB Platz benötigen, vielmehr wird es häufig genug so sein, dass diese Anwenderadressraume nur zu einem Bruchteil mit wirklicher Information gefüllt sind, also z.B. mit einer Coderegion von zig KByte, einer Stackregion von einigen KByte und den globalen Daten von vielleicht 1 MByte. Der Rest sind Adressraumlöcher, die von der Anwendung nie benötigt werden.

12.3 Abbildung von logischen Adressräumen auf den Hauptspeicher

Es gibt ein Fülle von Kombinationen wie man logische Adressräume auf den physischen Adressraum (=Hauptspeicher) abbilden kann, u.a.:

Zusammenhängender AR	->	zusammenhängender Speicherblock (Partition)
Nicht zusammenhängender AR	->	zusammenhängender Speicherblock (Partition)
Zusammenhängender AR	->	nicht zusammenhängende Speicherblöcke
Regionen	->	nicht zusammenhängende Speicherblöcke
Seiten	->	nicht zusammenhängende Kacheln

Zusätzliche kann man noch unterscheiden, wie häufig man diese Abbildungen im Laufe der Abarbeitung der Task verrichten muss, einmalig oder u.U. auch öfters.

12.3.1 Einmaliges Transferieren des Taskraums in den Hauptspeichers

Diese Methode ist insbesondere dann angebracht, wenn nur eine statische Anzahl von Prozessen vorliegt oder in besonders einfach und robust zu haltenden Systemen (*embedded devices* etc.).

12.3.2 Mehrmaliges Ein- und Auslagern des Taskadressraums

Je komfortabler die Abwicklung von Prozessen und Tasks ist, desto wünschenswerter mag es auch sein, dass die Benutzer oder auch das System zwischendurch eine Task oder einen Prozess mittel- oder auch langfristig aus dem System auslagern wollen, damit andere dringlichere Aufgaben zwischendurch erledigt werden können. Hierbei wird es außerdem eine Rolle spielen, ob dabei die Task oder der Prozess nur komplett oder auch teilweise ausgelagert sein dürfen.

12.4 Adressraumverwaltung

12.4.1 Aufspaltung des Hauptspeichers zwischen Kern und Anwendung(en)

So genannte Einprogrammsysteme sind relativ einfach zu verwalten, es mag nur noch darum gehen, an welchen physischen Adressen bestimmte Systemtabellen (z.B. die Unterbrechungsvektortabelle oder bestimmte Geräteregister) aus technischen Gründen im Hauptspeicher abgebildet sein müssen, dementsprechend wird man dann in deren Umgebung auch die zugehörigen Systemkomponenten (Kern bzw. Gerätetreiber platzieren).

In Mehrprogammbetriebssystemen geht es darum, möglichst effektiv und effizient eine Platzierungsvorschrift für die n>1 gleichzeitig im System vorhandenen Anwendungen zu finden.

12.4.2 Fixe Partitionierung

Robuster und einfacher ist es, eine fixe Partitionierung des Hauptspeichers einzuführen, andererseits wird hierbei u.U. viel Hauptspeicher effektiv überhaupt nicht genutzt, wenn zufällig gleichzeitig nur sehr kleine Anwendungen mit wenig Platzbedarf geladen worden sind. Man spricht in diesem Fall vom internen Speicherverschnitt (*internal fragmentation*).

12.4.3 Dynamische Partitionierung

Demgegenüber ist die dynamische Partitionierung flexibler, andererseits aber auch aufwendiger zu implementieren. Hierbei kann es zu einer anderen Art von Speicherverschnitt kommen, wenn nämlich in der Summe zwar genügend viel freier Speicher vorhanden ist, jedoch kein einziges dieser freien Stücke groß genug ist, um die aktuelle Speicheranforderung zu erfüllen (*external fragmentation*).

12.4.4 Speicherallokation für dynamisch große Partitionen

12.4.4.1 First Fit

Im Mittel wird nur der vordere Speicher benutzt, d.h. insbesondere dass am Ende des Speichers relativ große freie Stücke übrig bleiben.

12.4.4.2 Next Fit

Next-Fit ist langsamer als First-Fit ist, obwohl es eine etwas gleichmäßigere Speicherallokation durchführt. Der Grund hierfür ist, dass mit großer Wahrscheinlichkeit vor dem Next-Fit-Zeiger sein relativ großes freies Speicherstück liegt, das dann auch bei kleinen Anforderungen aufgeteilt wird, so dass bei höherer Speicherauslastung relativ schnell keine großen freien Stücke mehr übrig sind.

12.4.4.3 Best Fit

Auf den ersten Blick das beste Verfahren, aber man sucht u.U. jedes Mal den ganzen Freispeichervektor ab, ehe man das passendste freie Stück gefunden hat. Darüber hinaus tendiert dieses Verfahren dazu, dass häufig

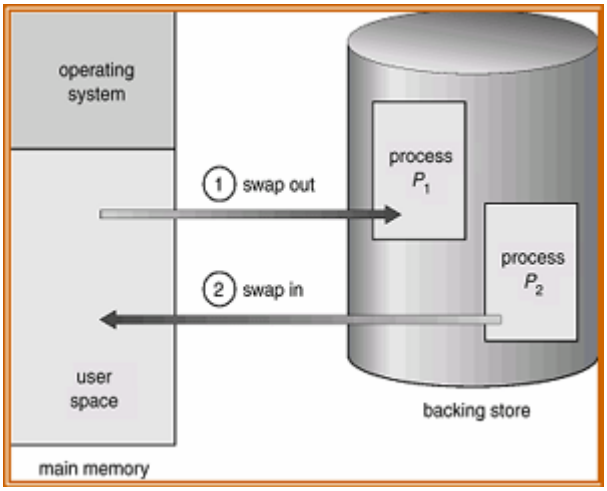
relativ kleine Speicherstückchen noch übrig bleiben, die zu klein für jegliche neue Anforderung sind, was u.U. zu einer großen Speicherzerstückelung (à la Schweizerkäse) führen kann.

12.4.4.4 Nearest Fit

Hierbei wird vom aktuellen Nearest-Fit Zeiger in beiden Richtungen nach dem nächst gelegenen freien Stück gesucht, dass größer als die Anforderung ist.

12.4.5 Swapping und Overlay

Wenn also Tasks bzw. Prozesse bzw. Teile von ihnen zwischen Platte und Hauptspeicher ein- und ausgelagert werden sollen, dann braucht man eine Verwaltung im Hauptspeicher und auf Platte.



Wenn man also schon Teile von Adressräumen ein- und auslagern kann, dann könnte man diesen Mechanismus auch darauf anwenden, dass man Teile von zu großen Adressräumen je nach Bedarf aus- und einlagert. Dies wurde in Systemen, die noch ohne virtuellen Speicher gearbeitet haben, mit Erfolg mittels der Überlagerungstechnik (*overlay*) durchgeführt. In den -zu großen- Anwenderprogrammen musste dem Binder (*linker*) mitgeteilt werden, welche Stücke der Anwendung resident bleiben sollten, und welche bei Bedarf ein- und wieder ausgelagert werden konnten.